
8

SRP: The Single-Responsibility Principle



None but Buddha himself must take the responsibility of giving out occult secrets...

—E. Cobham Brewer 1810–1897.
Dictionary of Phrase and Fable. 1898.

This principle was described in the work of Tom DeMarco¹ and Meilir Page-Jones.² They called it *cohesion*. They defined cohesion as the functional relatedness of the elements of a module. In this chapter we'll shift that meaning a bit and relate cohesion to the forces that cause a module, or a class, to change.

SRP: The Single-Responsibility Principle

A class should have only one reason to change.

Consider the bowling game from Chapter 6. For most of its development, the `Game` class was handling two separate responsibilities. It was keeping track of the current frame, and it was calculating the score. In the end, `RCM` and `RSK` separated these two responsibilities into two classes. The `Game` kept the responsibility to keep track of frames, and the `Scorer` got the responsibility to calculate the score. (See page 78.)

1. [DeMarco79], p. 310.
2. [Page-Jones88], Chapter 6, p. 82.

Why was it important to separate these two responsibilities into separate classes? Because each responsibility is an axis of change. When the requirements change, that change will be manifest through a change in responsibility amongst the classes. If a class assumes more than one responsibility, then there will be more than one reason for it to change.

If a class has more than one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the ability of the class to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when changed.

For example, consider the design in Figure 8-1. The `Rectangle` class has two methods shown. One draws the rectangle on the screen, the other computes the area of the rectangle.

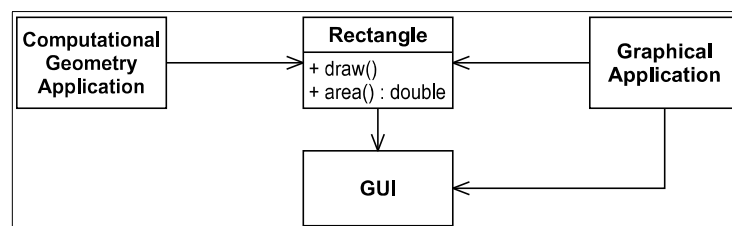


Figure 8-1 More than one responsibility

Two different applications use the `Rectangle` class. One application does computational geometry. It uses `Rectangle` to help it with the mathematics of geometric shapes. It never draws the rectangle on the screen. The other application is graphical in nature. It may also do some computational geometry, but it definitely draws the rectangle on the screen.

This design violates the Single-Responsibility Principle (SRP). The `Rectangle` class has two responsibilities. The first responsibility is to provide a mathematical model of the geometry of a rectangle. The second responsibility is to render the rectangle on a graphical user interface.

This violation of the SRP causes several nasty problems. First, we must include the `GUI` in the computational geometry application. If this were a C++ application, the `GUI` would have to be linked in, consuming link time, compile time, and memory footprint. In a Java application, the `.class` files for the `GUI` have to be deployed to the target platform.

Second, if a change to the `GraphicalApplication` causes the `Rectangle` to change for some reason, that change may force us to rebuild, retest, and redeploy the `ComputationalGeometryApplication`. If we forget to do this, that application may break in unpredictable ways.

A better design is to separate the two responsibilities into two completely different classes as shown in Figure 8-2. This design moves the computational portions of `Rectangle` into the `GeometricRectangle` class. Now changes made to the way rectangles are rendered cannot affect the `ComputationalGeometryApplication`.

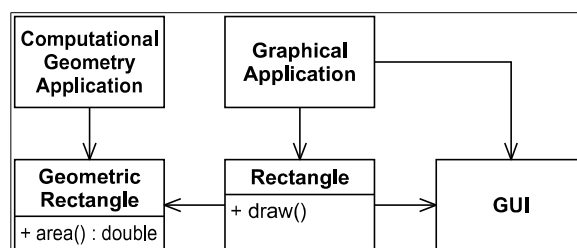


Figure 8-2 Separated Responsibilities

What Is a Responsibility?

In the context of the SRP, we define a responsibility to be “a reason for change.” If you can think of more than one motive for changing a class, then that class has more than one responsibility. This is sometimes hard to see. We are accustomed to thinking of responsibility in groups. For example, consider the `Modem` interface in Listing 8-1. Most of us will agree that this interface looks perfectly reasonable. The four functions it declares are certainly functions belonging to a modem.

Listing 8-1

Modem.java -- SRP Violation

```
interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
}
```

However, there are two responsibilities being shown here. The first responsibility is connection management. The second is data communication. The `dial` and `hangup` functions manage the connection of the modem, while the `send` and `recv` functions communicate data.

Should these two responsibilities be separated? That depends on how the application is changing. If the application changes in ways that affect the signature of the connection functions, then the design will smell of Ridity because the classes that call `send` and `recv` will have to be recompiled and redeployed more often than we like. In that case the two responsibilities should be separated as shown in Figure 8-3. This keeps the client applications from coupling the two responsibilities.

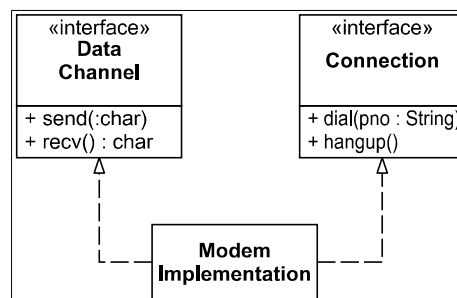


Figure 8-3 Separated Modem Interface

If, on the other hand, the application is not changing in ways that cause the the two responsibilities to change at different times, then there is no need to separate them. Indeed, separating them would smell of Needless Complexity.

There is a corollary here. *An axis of change is an axis of change only if the changes actually occur.* It is not wise to apply the SRP, or any other principle for that matter, if there is no symptom.

Separating Coupled Responsibilities

Notice that in Figure 8-3 I kept both responsibilities coupled in the `ModemImplementation` class. This is not desirable, but it may be necessary. There are often reasons, having to do with the details of the hardware or OS, that force us to couple things that we’d rather not couple. However, by separating their interfaces we have decoupled the concepts as far as the rest of the application is concerned.

We may view the `ModemImplementation` class as a kludge, or a wart; however, notice that all dependencies flow *away* from it. Nobody needs to depend on this class. Nobody except `main` needs to know that it exists. Thus, we've put the ugly bit behind a fence. Its ugliness need not leak out and pollute the rest of the application.

Persistence

Figure 8-4 shows a common violation of the SRP. The `Employee` class contains business rules and persistence control. These two responsibilities should almost never be mixed. Business rules tend to change frequently, and though persistence may not change as frequently, it changes for completely different reasons. Binding business rules to the persistence subsystem is asking for trouble.

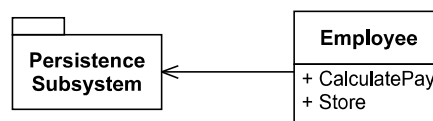


Figure 8-4 Coupled Persistence

Fortunately, as we saw in Chapter 4, the practice of test-driven development will usually force these two responsibilities to be separated long before the design begins to smell. However, in cases where the tests did not force the separation, and the smells of Ridity and Fragility become strong, the design should be refactored using the FACADE or PROXY patterns to separate the two responsibilities.

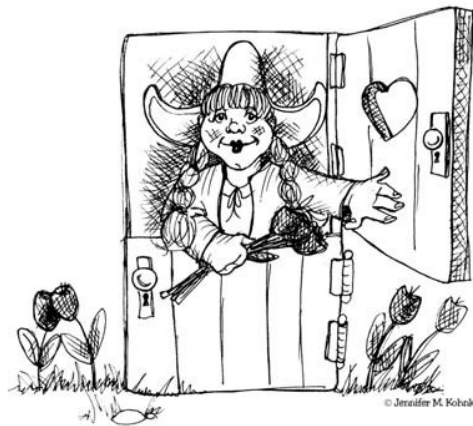
Conclusion

The SRP is one of the simplest of the principles, and one of the hardest to get right. Conjoining responsibilities is something that we do naturally. Finding and separating those responsibilities from one another is much of what software design is really about. Indeed, the rest of the principles we will discuss come back to this issue in one way or another.

Bibliography

1. DeMarco, Tom. *Structured Analysis and System Specification*. Yourdon Press Computing Series. Englewood Cliff, NJ: 1979.
2. Page-Jones, Meilir. *The Practical Guide to Structured Systems Design*, 2d ed. Englewood Cliff, NJ: Yourdon Press Computing Series, 1988.

OCP: The Open–Closed Principle



Dutch Door—(Noun) *A door divided in two horizontally so that either part can be left open or closed.*

—The American Heritage® Dictionary of the English Language: Fourth Edition. 2000.

As Ivar Jacobson has said, “All systems change during their life cycles. This must be born in mind when developing systems are expected to last longer than the first version.”¹ How can we create designs that are stable in the face of change and that will last longer than the first version? Bertrand Meyer gave us guidance as long ago as 1988 when he coined the now famous Open–Closed Principle.²

OCP: The Open–Closed Principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

When a single change to a program results in a cascade of changes to dependent modules, the design smells of Rigidity. The OCP advises us to refactor the system so that further changes of that kind will not cause more

1. [Jacobson92], p. 21.

2. [Meyer97], p. 57.

modifications. If the OCP is applied well, then further changes of that kind are achieved by adding new code, not by changing old code that already works.

This may seem like motherhood and apple pie—the golden unachievable ideal—but in fact there are some relatively simple and effective strategies for *approaching* that ideal.

Description

Modules that conform to the Open–Closed Principle have two primary attributes. They are

1. “Open for extension.”

This means that the behavior of the module can be extended. As the requirements of the application change, we are able to extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does.

2. “Closed for modification.”

Extending the behavior of a module does not result in changes to the source or binary code of the module. The binary executable version of the module, whether in a linkable library, a DLL, or a Java `.jar`, remains untouched.

It would seem that these two attributes are at odds with each other. The normal way to extend the behavior of a module is to make changes to the source code of that module. A module that cannot be changed is normally thought to have a fixed behavior.

How is it possible that the behaviors of a module can be modified without changing its source code? How can we change what a module does, without changing the module?

Abstraction Is the Key

In C++, Java, or any other OOPL,³ it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors. The abstractions are abstract base classes, and the unbounded group of possible behaviors is represented by all the possible derivative classes.

It is possible for a module to manipulate an abstraction. Such a module can be closed for modification since it depends upon an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction.

Figure 9–1 shows a simple design that does not conform to the OCP. Both the `Client` and `Server` classes are concrete. The `Client` class *uses* the `Server` class. If we wish for a `Client` object to use a different server object, then the `Client` class must be changed to name the new server class.



Figure 9-1 Client is not open and closed

Figure 9–2 shows the corresponding design that conforms to the OCP. In this case, the `ClientInterface` class is an abstract class with abstract member functions. The `Client` class uses this abstraction; however, objects of the `Client` class will be using objects of the derivative `Server` class. If we want `Client` objects to use a different server class, then a new derivative of the `ClientInterface` class can be created. The `Client` class can remain unchanged.

The `Client` has some work that it needs to get done, and it can describe that work in terms of the abstract interface presented by `ClientInterface`. Subtypes of `ClientInterface` can implement that interface in any

3. Object-oriented programming language.

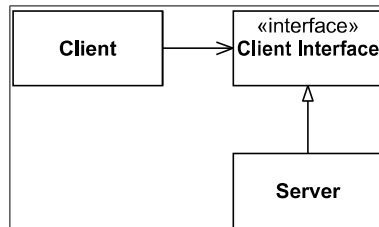


Figure 9-2 STRATEGY pattern: Client is both open and closed

manner they choose. Thus, the behavior specified in `Client` can be extended and modified by creating new subtypes of `ClientInterface`.

You may wonder why I named `ClientInterface` the way I did. Why didn't I call it `AbstractServer` instead? The reason, as we will see later, is that *abstract classes are more closely associated to their clients than to the classes that implement them.*

Figure 9-3 shows an alternative structure. The `Policy` class has a set of concrete public functions that implements a policy of some kind. Similar to the functions of the `Client` in Figure 9-2. As before, those policy functions describe some work that needs to be done in terms of some abstract interfaces. However, in this case, the abstract interfaces are part of the `Policy` class itself. In C++ they would be pure virtual functions, and in Java they would be abstract methods. Those functions are implemented in the subtypes of `Policy`. Thus, the behaviors specified within `Policy` can be extended or modified by creating new derivatives of the `Policy` class.

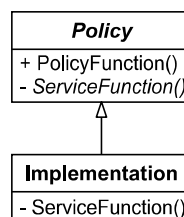


Figure 9-3 Template Method Pattern: Base class is open and closed

These two patterns are the most common ways of satisfying the OCP. They represent a clear separation of generic functionality from the detailed implementation of that functionality.

The Shape Application

The following example has been shown in many books on OOD. It is the infamous “Shape” example. It is normally used to show how polymorphism works. However, this time we will use it to elucidate the OCP.

We have an application that must be able to draw circles and squares on a standard GUI. The circles and squares must be drawn in a particular order. A list of the circles and squares will be created in the appropriate order, and the program must walk the list in that order and draw each circle or square.

Violating the OCP

In C, using procedural techniques that do not conform to the OCP, we might solve this problem as shown in Listing 9-1. Here we see a set of data structures that has the same first element, but is different beyond that. The first element of each is a type code that identifies the data structure as either a circle or a square. The function `DrawAllShapes` walks an array of pointers to these data structures, examining the type code and then calling the appropriate function (either `DrawCircle` or `DrawSquare`).

Listing 9-1**Procedural Solution to the Square/Circle Problem**

```

--shape.h-----
enum ShapeType {circle, square};

struct Shape
{
    ShapeType itsType;
};

--circle.h-----
struct Circle
{
    ShapeType itsType;
    double itsRadius;
    Point itsCenter;
};

void DrawCircle(struct Circle*);

--square.h-----
struct Square
{
    ShapeType itsType;
    double itsSide;
    Point itsTopLeft;
};

void DrawSquare(struct Square*);

--drawAllShapes.cc-----
typedef struct Shape *ShapePointer;

void DrawAllShapes(ShapePointer list[], int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        struct Shape* s = list[i];
        switch (s->itsType)
        {
            case square:
                DrawSquare((struct Square*)s);
                break;

            case circle:
                DrawCircle((struct Circle*)s);
                break;
        }
    }
}

```

The function `DrawAllShapes` does not conform to the OCP because it cannot be closed against new kinds of shapes. If I wanted to extend this function to be able to draw a list of shapes that included triangles, I would

The Shape Application

have to modify the function. In fact, I would have to modify the function for any new type of shape that I needed to draw.

Of course this program is only a simple example. In real life, the `switch` statement in the `DrawAllShapes` function would be repeated over and over again in various functions all through the application, each one doing something a little different. There might be functions for dragging shapes, stretching shapes, moving shapes, deleting shapes, etc. Adding a new shape to such an application means hunting for every place that such `switch` statements (or `if/else` chains) exist and adding the new shape to each.

Moreover, it is very unlikely that all the `switch` statements and `if/else` chains would be as nicely structured as the one in `DrawAllShapes`. It is much more likely that the predicates of the `if` statements would be combined with logical operators or that the `case` clauses of the `switch` statements would be combined so as to “simplify” the local decision making. In some pathological situations, there may be functions that do precisely the same things to `Squares` that they do to `Circles`. Such functions would not even have the `switch/case` statements or `if/else` chains. Thus, the problem of finding and understanding all the places where the new shape needs to be added can be nontrivial.

Also, consider the kind of changes that would have to be made. We’d have to add a new member to the `ShapeType` enum. Since all the different shapes depend on the declaration of this enum, we’d have to recompile them all.⁴ And we’d also have to recompile all the modules that depend on `Shape`.

So, not only must we change the source code of all `switch/case` statements or `if/else` chains, but we also must alter the binary files (via recompilation) of all the modules that use any of the `Shape` data structures. Changing the binary files means that any DLLs, shared libraries, or other kinds of binary components must be redeployed. The simple act of adding a new shape to the application causes a cascade of subsequent changes to many source modules and to even more binary modules and binary components. Clearly, the impact of adding a new shape is very large.

Bad Design. Let’s run through this again. The solution in Listing 9-1 is *Rigid* because the addition of `Triangle` causes `Shape`, `Square`, `Circle`, and `DrawAllShapes` to be recompiled and redeployed. It is *Fragile* because there will be many other `switch/case` or `if/else` statements that are both hard to find and hard to decipher. It is *Immobile* because anyone attempting to reuse `DrawAllShapes` in another program is required to bring along `Square` and `Circle`, even if that new program does not need them. Thus, Listing 9-1 exhibits many of the smells of bad design.



Conforming to the OCP

Listing 9-2 shows the code for a solution to the `square/circle` problem that conforms to the OCP. In this case, we have written an abstract class named `Shape`. This abstract class has a single abstract method named `Draw`. Both `Circle` and `Square` are derivatives of the `Shape` class.

Listing 9-2

OOD solution to Square/Circle problem.

```
class Shape
{
    public:
        virtual void Draw() const = 0;
};
```

4. Changes to enums can cause a change in the size of the variable used to hold the enum. So, great care must be taken if you decide that you don’t really need to recompile the other shape declarations.

```

class Square : public Shape
{
    public:
        virtual void Draw() const;
};

class Circle : public Shape
{
    public:
        virtual void Draw() const;
};

void DrawAllShapes(vector<Shape*>& list)
{
    vector<Shape*>::iterator i;
    for (i=list.begin(); i != list.end(); i++)
        (*i)->Draw();
}

```

Note that if we want to extend the behavior of the `DrawAllShapes` function in Listing 9-2 to draw a new kind of shape, all we need do is add a new derivative of the `Shape` class. The `DrawAllShapes` function does not need to change. Thus `DrawAllShapes` conforms to the OCP. Its behavior can be extended without modifying it. Indeed, adding a `Triangle` class has *absolutely no effect* on any of the modules shown here. Clearly some part of the system must change in order to deal with the `Triangle` class, but all of the code shown here is immune to the change.

In a real application, the `Shape` class would have many more methods. Yet adding a new shape to the application is still quite simple since all that is required is to create the new derivative and implement all its functions. There is no need to hunt through all of the application looking for places that require changes. This solution is not *Fragile*.

Nor is the solution *Rigid*. No existing source modules need to be modified, and with one exception, no existing binary modules need to be rebuilt. The module that actually creates instances of the new derivative of `Shape` must be modified. Typically, this is either done by `main`, in some function called by `main`, or in the method of some object created by `main`.⁵

Finally, the solution is not *Immobile*. `DrawAllShapes` can be reused by any application without the need to bring `Square` or `Circle` along for the ride. Thus, the solution exhibits none of the attributes of bad design mentioned previously.

This program conforms to the OCP. *It is changed by adding new code rather than by changing existing code.* Therefore, it does not experience the cascade of changes exhibited by nonconforming programs. The only changes required are the addition of the new module and the change related to `main` that allows the new objects to be instantiated.

OK, I Lied

The previous example was blue sky and apple pie! Consider what would happen to the `DrawAllShapes` function from Listing 9-2 if we decided that *all Circles should be drawn before any Squares*. The `DrawAllShapes` function is not closed against a change like this. To implement that change, we'll have to go into `DrawAllShapes` and scan the list first for `Circles` and then again for `Squares`.

5. Such objects are known as *factories*, and we'll have more to say about them in Chapter 21 on page 269.

Anticipation and “Natural” Structure

Had we anticipated this kind of change, then we could have invented an abstraction that protected us from it. The abstractions we chose in Listing 9-2 are more of a hindrance to this kind of change than a help. You may find this surprising. After all, what could be more natural than a `Shape` base class with `Square` and `Circle` derivatives? Why isn't that natural model the best one to use? Clearly the answer is that the model is *not* natural in a system where ordering is more significant than shape type.

This leads us to a disturbing conclusion. In general, no matter how “closed” a module is, there will always be some kind of change against which it is not closed. *There is no model that is natural to all contexts!*

Since closure cannot be complete, it must be strategic. That is, the designer must choose the kinds of changes against which to close his design. He must guess at the most likely kinds of changes, and then construct abstractions to protect him from those changes.

This takes a certain amount of prescience derived from experience. The experienced designer hopes he knows the users and the industry well enough to judge the probability of different kinds of changes. He then invokes the OCP against the most probable changes.

This is not easy. It amounts to making educated guesses about the likely kinds of changes that the application will suffer over time. When the developers guess right, they win. When they guess wrong, they lose. And they will certainly guess wrong much of the time.

Also, conforming to the OCP is expensive. It takes development time and effort to create the appropriate abstractions. Those abstractions also increase the complexity of the software design. There is a limit to the amount of abstraction that the developers can afford. Clearly, we want to limit the application of the OCP to changes that are likely.

How do we know which changes are likely? We do the appropriate research, we ask the appropriate questions, and we use our experience and common sense. And after all that, *we wait until the changes happen!*



Putting the “Hooks” In

How do we protect ourselves from changes? In the previous century, we had a saying. We'd “put the hooks in” for changes that we thought might take place. We felt that this would make our software flexible.

However, the hooks we put in were often incorrect. Worse, they smelled of *Needless Complexity* that had to be supported and maintained, even though they weren't used. This is not a good thing. We don't want to load the design with lots of unnecessary abstraction. Rather, we often wait until we actually need the abstraction, and then we put it in.

Fool Me Once... There is an old saying: “Fool me once, shame on you. Fool me twice, shame on me.” This is a powerful attitude in software design. To keep from loading our software with *Needless Complexity*, we may permit ourselves to be fooled *once*. This means we initially write our code expecting it not to change. When a change occurs, we implement the abstractions that protect us from future changes *of that kind*. In short, we *take the first bullet*, and then we make sure we are protected from any more bullets coming from that gun.

Stimulating Change. If we decide to take the first bullet, then it is to our advantage to get the bullets flying early and frequently. We want to know what kinds of changes are likely before we are very far down the development path. The longer we wait to find out what kinds of changes are likely, the harder it will be to create the appropriate abstractions.

Therefore, we need to stimulate the changes. We do this through several of the means we discussed in Chapter 2.

- We write tests first. Testing is one kind of usage of the system. By writing tests first we force the system to be testable. Therefore changes in testability will not surprise us later. We will have built the abstractions that make the system testable. We are likely to find that many of these abstractions will protect us from other kinds of changes later.
- We develop using very short cycles—days instead of weeks.
- We develop features before infrastructure and frequently show those features to stakeholders.
- We develop the most important features first.
- We release the software early and often. We get it in front of our customers and users as quickly and as often as possible.

Using Abstraction to Gain Explicit Closure

OK, so we've taken the first bullet. The user wants us to draw all `Circles` before any `Squares`. Now we want to protect ourselves from any future changes of that kind.

How can we close the `DrawAllShapes` function against changes in the ordering of drawing? Remember that closure is based upon abstraction. Thus, in order to close `DrawAllShapes` against ordering, we need some kind of “ordering abstraction.” This abstraction would provide an abstract interface through which any possible ordering policy could be expressed.

An ordering policy implies that, given any two objects, it is possible to discover which ought to be drawn first. We can define an abstract method of `Shape` named `Precedes`. This function takes another `Shape` as an argument and returns a `bool` result. The result is `true` if the `Shape` object that receives the message should be drawn before the `Shape` object passed as the argument.

In C++, this function could be represented by an overloaded `operator<` function. Listing 9-3 shows what the `Shape` class might look like with the ordering methods in place.

Now that we have a way to determine the relative ordering of two `Shape` objects, we can sort them and then draw them in order. Listing 9-4 shows the C++ code that does this.

Listing 9-3

Shape with ordering methods

```
class Shape
{
public:
    virtual void Draw() const = 0;
    virtual bool Precedes(const Shape&) const = 0;

    bool operator<(const Shape& s) {return Precedes(s);}
};
```

Listing 9-4

DrawAllShapes with Ordering

```
template <typename P>
class Lessp // utility for sorting containers of pointers.
{
public:
    bool operator()(const P p, const P q) {return (*p) < (*q);}
};

void DrawAllShapes(vector<Shape*>& list)
{
    vector<Shape*> orderedList = list;
```

The Shape Application

```
    sort(orderedList.begin(),
         orderedList.end(),
         Lessp<Shape*>());

    vector<Shape*>::const_iterator i;
    for (i=orderedList.begin(); i != orderedList.end(); i++)
        (*i)->Draw();
}
```

This gives us a means for ordering `Shape` objects and for drawing them in the appropriate order. But we still do not have a decent ordering abstraction. As it stands, the individual `Shape` objects will have to override the `Precedes` method in order to specify ordering. How would this work? What kind of code would we write in `Circle::Precedes` to ensure that `Circles` were drawn before `Squares`? Consider Listing 9–5.

Listing 9-5

Ordering a Circle

```
bool Circle::Precedes(const Shape& s) const
{
    if (dynamic_cast<Square*>(s))
        return true;
    else
        return false;
}
```

It should be very clear that this function, and all its siblings in the other derivatives of `Shape`, do not conform to the OCP. There is no way to close them against new derivatives of `Shape`. Every time a new derivative of `Shape` is created, all the `Precedes()` functions will need to be changed.⁶

Of course this doesn't matter if no new derivatives of `Shape` are ever created. On the other hand, if they are created frequently, this design would cause a significant amount of thrashing. Again, we'd take the first bullet.

Using a “Data-Driven” Approach to Achieve Closure

If we must close the derivatives of `Shape` from knowledge of each other, we can use a table-driven approach. Listing 9-6 shows one possibility.

Listing 9-6

Table driven type ordering mechanism

```
#include <typeinfo>
#include <string>
#include <iostream>

using namespace std;

class Shape
{
public:
    virtual void Draw() const = 0;
    bool Precedes(const Shape&) const;
```

6. It is possible to solve this problem by using the ACYCLIC VISITOR pattern described in Chapter 29. Showing that solution now would be getting ahead of ourselves a bit. I'll remind you to come back here at the end of that chapter.

```

    bool operator<(const Shape& s) const
    {return Precedes(s);}
private:
    static const char* typeOrderTable[];
};

const char* Shape::typeOrderTable[] =
{
    typeid(Circle).name(),
    typeid(Square).name(),
    0
};

// This function searches a table for the class names.
// The table defines the order in which the
// shapes are to be drawn. Shapes that are not
// found always precede shapes that are found.
//
bool Shape::Precedes(const Shape& s) const
{
    const char* thisType = typeid(*this).name();
    const char* argType = typeid(s).name();
    bool done = false;
    int thisOrd = -1;
    int argOrd = -1;
    for (int i=0; !done; i++)
    {
        const char* tableEntry = typeOrderTable[i];
        if (tableEntry != 0)
        {
            if (strcmp(tableEntry, thisType) == 0)
                thisOrd = i;
            if (strcmp(tableEntry, argType) == 0)
                argOrd = i;
            if ((argOrd >= 0) && (thisOrd >= 0))
                done = true;
        }
        else // table entry == 0
            done = true;
    }
    return thisOrd < argOrd;
}

```

By taking this approach, we have successfully closed the `DrawAllShapes` function against ordering issues in general and each of the `Shape` derivatives against the creation of new `Shape` derivatives or a change in policy that reorders the `Shape` objects by their type. (e.g., changing the ordering so that `Squares` are drawn first.)

The only item that is not closed against the order of the various `Shapes` is the table itself. That table can be placed in its own module, separate from all the other modules, so that changes to it do not affect any of the other modules. Indeed, in C++, we can choose which table to use at link time.

Conclusion

In many ways, the OCP is at the heart of object-oriented design. Conformance to this principle is what yields the greatest benefits claimed for object oriented technology (i.e., flexibility, reusability, and maintainability). Yet conformance to this principle is not achieved simply by using an object-oriented programming language. Nor is it a

Conclusion

good idea to apply rampant abstraction to every part of the application. Rather, it requires a dedication on the part of the developers to apply abstraction only to those parts of the program that exhibit frequent change. *Resisting premature abstraction is as important as abstraction itself.*

Bibliography

1. Jacobson, Ivar, et al. *Object-Oriented Software Engineering*. Reading, MA: Addison–Wesley, 1992.
2. Meyer, Bertrand. *Object-Oriented Software Construction*. 2d ed. Upper Saddle River, NJ: Prentice Hall, 1997.

LSP: The Liskov Substitution Principle



The primary mechanisms behind the OCP are abstraction and polymorphism. In statically typed languages like C++ and Java, one of the key mechanisms that supports abstraction and polymorphism is inheritance. It is by using inheritance that we can create derived classes that implement abstract methods in base classes.

What are the design rules that govern this particular use of inheritance? What are the characteristics of the best inheritance hierarchies? What are the traps that will cause us to create hierarchies that do not conform to the OCP? These are the questions that are addressed by the Liskov Substitution Principle (LSP).

LSP: The Liskov Substitution Principle

The LSP can be paraphrased as follows:

SUBTYPES MUST BE SUBSTITUTABLE FOR THEIR BASE TYPES.

Barbara Liskov first wrote this principle in 1988.¹ She said,

What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T .

1. [Liskov88].

The importance of this principle becomes obvious when you consider the consequences of violating it. Presume that we have a function f that takes, as its argument, a pointer or reference to some base class B . Presume also that there is some derivative D of B which, when passed to f in the guise of B , causes f to misbehave. Then D violates the LSP. Clearly D is Fragile in the presence of f .

The authors of f will be tempted to put in some kind of test for D so that f can behave properly when a D is passed to it. This test violates the OCP because now f is not closed to all the various derivatives of B . Such tests are a code smell that are the result of inexperienced developers (or, what's worse, developers in a hurry) reacting to LSP violations.

A Simple Example of a Violation of the LSP

Violating the LSP often results in the use of Run-Time Type Information (RTTI) in a manner that grossly violates the OCP. Frequently, an explicit `if` statement or `if/else` chain is used to determine the type of an object so that the behavior appropriate to that type can be selected. Consider Listing 10-1.

Listing 10-1

A violation of LSP causing a violation of OCP.

```
struct Point {double x,y;};

struct Shape {
    enum ShapeType {square, circle} itsType;
    Shape(ShapeType t) : itsType(t) {}
};

struct Circle : public Shape
{
    Circle() : Shape(circle) {};
    void Draw() const;
    Point itsCenter;
    double itsRadius;
};

struct Square : public Shape
{
    Square() : Shape(square) {};
    void Draw() const;
    Point itsTopLeft;
    double itsSide;
};

void DrawShape(const Shape& s)
{
    if (s.itsType == Shape::square)
        static_cast<const Square&>(s).Draw();
    else if (s.itsType == Shape::circle)
        static_cast<const Circle&>(s).Draw();
}
```

Clearly, the `DrawShape` function in Listing 10-1 violates the OCP. It must know about every possible derivative of the `Shape` class, and it must be changed whenever new derivatives of `Shape` are created. Indeed, many rightly view the structure of this function as anathema to good design. What would drive a programmer to write a function like this?

Square and Rectangle, a More Subtle Violation

Consider Joe the Engineer. Joe has studied object-oriented technology and has come to the conclusion that the overhead of polymorphism is too high to pay.² Therefore, he defined class `Shape` without any virtual functions. The classes (structs) `Square` and `Circle` derive from `Shape` and have `Draw()` functions, but they don't override a function in `Shape`. Since `Circle` and `Square` are not substitutable for `Shape`, `DrawShape` must inspect its incoming `Shape`, determine its type, and then call the appropriate `Draw` function.

The fact that `Square` and `Circle` cannot be substituted for `Shape` is a violation of the LSP. This violation forced the violation of the OCP by `DrawShape`. Thus, *a violation of LSP is a latent violation of OCP*.

Square and Rectangle, a More Subtle Violation

Of course, there are other, far more subtle, ways of violating the LSP. Consider an application which uses the `Rectangle` class as described in Listing 10-2.

Listing 10-2

Rectangle class

```
class Rectangle
{
    public:
        void    SetWidth(double w)  {itsWidth=w;}
        void    SetHeight(double h) {itsHeight=w;}
        double GetHeight() const   {return itsHeight;}
        double GetWidth() const    {return itsWidth;}
    private:
        Point  itsTopLeft;
        double itsWidth;
        double itsHeight;
};
```

Imagine that this application works well and is installed in many sites. As is the case with all successful software, its users demand changes from time to time. One day, the users demand the ability to manipulate *squares* in addition to rectangles.

It is often said that inheritance is the *IS-A* relationship. In other words, if a new kind of object can be said to fulfill the *IS-A* relationship with an old kind of object, then the class of the new object should be derived from the class of the old object.

For all normal intents and purposes, a square *is a* rectangle. Thus, it is logical to view the `Square` class as being derived from the `Rectangle` class. (See Figure 10-1.)

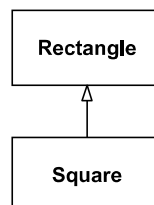


Figure 10-1 Square inherits from Rectangle

This use of the *IS-A* relationship is sometimes thought to be one of the fundamental techniques of object-oriented analysis.³ A square is a rectangle, and so the `Square` class should be derived from the `Rectangle` class.

2. On a reasonably fast machine, that overhead is on the order of `Ins` per method invocation, so it's hard to see Joe's point.
3. A term that is frequently used but seldom defined.

However, this kind of thinking can lead to some subtle, yet significant, problems. Generally, these problems are not foreseen until we see them in code.

Our first clue that something has gone wrong might be the fact that a `Square` does not need both `itsHeight` and `itsWidth` member variables. Yet it will inherit them from `Rectangle`. Clearly, this is wasteful. In many cases, such waste is insignificant. But if we must create hundreds of thousands of `Square` objects (e.g., a CAD/CAE program in which every pin of every component of a complex circuit is drawn as a square), this waste could be significant.

Let's assume, for the moment, that we are not very concerned with memory efficiency. There are other problems that ensue from deriving `Square` from `Rectangle`. `Square` will inherit the `setWidth` and `setHeight` functions. These functions are inappropriate for a `Square`, since the width and height of a square are identical. This is a strong indication that there is a problem. However, there is a way to sidestep the problem. We could override `setWidth` and `setHeight` as follows:

```
void Square::setWidth(double w)
{
    Rectangle::setWidth(w);
    Rectangle::setHeight(w);
}

void Square::setHeight(double h)
{
    Rectangle::setHeight(h);
    Rectangle::setWidth(h);
}
```

Now, when someone sets the width of a `Square` object, its height will change correspondingly. And when someone sets the height, its width will change with it. Thus, the invariants⁴ of the `Square` remain intact. The `Square` object will remain a mathematically proper square.

```
Square s;
s.setWidth(1); // Fortunately sets the height to 1 too.
s.setHeight(2); // sets width and height to 2. Good thing.
```

But consider the following function:

```
void f(Rectangle& r)
{
    r.setWidth(32); // calls Rectangle::setWidth
}
```

If we pass a reference to a `Square` object into this function, the `Square` object will be corrupted because the height won't be changed. This is a clear violation of LSP. The `f` function does not work for derivatives of its arguments. The reason for the failure is that `setWidth` and `setHeight` were not declared `virtual` in `Rectangle`; therefore, they are not polymorphic.

We can fix this easily. However, when the creation of a derived class causes us to make changes to the base class, it often implies that the design is faulty. Certainly it violates the OCP. We might counter this by saying that forgetting to make `setWidth` and `setHeight` `virtual` was the real design flaw, and we are just fixing it now. However, this is hard to justify since setting the height and width of a rectangle are exceedingly primitive operations. By what reasoning would we make them `virtual` if we did not anticipate the existence of `Square`.

4. Those properties that must always be true regardless of state.

Still, let's assume that we accept the argument and fix the classes. We wind up with the code in Listing 10-3.

Listing 10-3

Rectangle and Square that are Self-Consistent.

```
class Rectangle
{
    public:
        virtual void SetWidth(double w) {itsWidth=w;}
        virtual void SetHeight(double h) {itsHeight=h;}
        double      GetHeight() const {return itsHeight;}
        double      GetWidth() const  {return itsWidth;}
    private:
        Point  itsTopLeft
        double itsHeight;
        double itsWidth;
};

class Square : public Rectangle
{
    public:
        virtual void SetWidth(double w);
        virtual void SetHeight(double h);
};

void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
}
```

The Real Problem

Square and Rectangle now appear to work. No matter what you do to a Square object, it will remain consistent with a mathematical square. And regardless of what you do to a Rectangle object, it will remain a mathematical rectangle. Moreover, you can pass a Square into a function that accepts a pointer or reference to a Rectangle, and the Square will still act like a square and will remain consistent.

Thus, we might conclude that the design is now self-consistent and correct. However, this conclusion would be amiss. A design that is self-consistent is not necessarily consistent with all its users! Consider the following function *g*:

```
void g(Rectangle& r)
{
    r.SetWidth(5);
    r.SetHeight(4);
    assert(r.Area() == 20);
}
```

This function invokes the `setWidth` and `setHeight` members of what it believes to be a `Rectangle`. The function works just fine for a `Rectangle`, but it declares an assertion error if passed a `Square`. So here is the real problem: *The author of `g` assumed that changing the width of a `Rectangle` leaves its height unchanged.*

Clearly, it is reasonable to assume that changing the width of a rectangle does not affect its height! However, not all objects that can be passed as `Rectangles` satisfy that assumption. If you pass an instance of a `Square` to a function like `g`, whose author made that assumption, then that function will malfunction. Function `g` is Fragile with respect to the `Square/Rectangle` hierarchy.

Function `g` shows that there exist functions that take pointers or references to `Rectangle` objects, but that cannot operate properly on `Square` objects. Since, for these functions, `Square` is not substitutable for `Rectangle`, the relationship between `Square` and `Rectangle` violates the LSP.

One might contend that the problem lay in function `g`—that the author had no right to make the assumption that width and height were independent. The author of `g` would disagree. The function `g` takes a `Rectangle` as its argument. There are invariants, statements of truth, that obviously apply to a class named `Rectangle`, and one of those invariants is that height and width are independent. The author of `g` had every right to assert this invariant. It is the author of `Square` that has violated the invariant.

Interestingly enough, the author of `Square` did not violate an invariant of `Square`. By deriving `Square` from `Rectangle`, the author of `Square` violated an invariant of `Rectangle`!

Validity Is Not Intrinsic

The LSP leads us to a very important conclusion: *A model, viewed in isolation, cannot be meaningfully validated.* The validity of a model can only be expressed in terms of its clients. For example, when we examined the final version of the `Square` and `Rectangle` classes in isolation, we found that they were self-consistent and valid. Yet when we looked at them from the viewpoint of a programmer who made reasonable assumptions about the base class, the model broke down.

When considering whether a particular design is appropriate or not, one cannot simply view the solution in isolation. One must view it in terms of the reasonable assumptions made by the users of that design.⁵

Who knows what reasonable assumptions the users of a design are going to make? Most such assumptions are not easy to anticipate. Indeed, if we tried to anticipate them all, we'd likely wind up imbuing our system with the smell of Needless Complexity. Therefore, like all other principles, it is often best to defer all but the most obvious LSP violations until the related Fragility has been smelled.

ISA Is about Behavior

So what happened? Why did the apparently reasonable model of the `Square` and `Rectangle` go bad? After all, isn't a `Square` a `Rectangle`? Doesn't the IS-A relationship hold?

Not as far as the author of `g` is concerned! A square might be a rectangle, but from `g`'s point of view, a `Square` object is definitely *not* a `Rectangle` object. Why? Because the *behavior* of a `Square` object is not consistent with `g`'s expectation of the behavior of a `Rectangle` object. Behaviorally, a `Square` is not a `Rectangle`, and it is *behavior* that software is really all about. The LSP makes it clear that in OOD, the IS-A relationship pertains to *behavior* that can be reasonably assumed and that clients depend on.

5. Often you will find that those reasonable assumptions are asserted in the unit tests written for the base class. Yet another good reason to practice test-driven development.



A Real Example

Design by Contract

Many developers may feel uncomfortable with the notion of behavior that is “reasonably assumed.” How do you know what your clients will really expect? There is a technique for making those reasonable assumptions explicit, thereby enforcing the LSP. The technique is called design by contract (DBC) and is expounded by Bertrand Meyer.⁶

Using DBC, the author of a class explicitly states the contract for that class. The contract informs the author of any client code about the behaviors that can be relied on. The contract is specified by declaring preconditions and postconditions for each method. The preconditions must be true in order for the method to execute. On completion, the method guarantees that the postconditions are true.

We can view the postcondition of `Rectangle::SetWidth(double w)` as follows:

```
assert((itsWidth == w) && (itsHeight == old.itsHeight));
```

In this example, `old` is the value of the `Rectangle` before `SetWidth` is called. Now the rule for preconditions and postconditions of derivatives, as stated by Meyer, is:

A routine redeclaration [in a derivative] may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger.⁷

In other words, when using an object through its base-class interface, the user knows only the preconditions and postconditions of the base class. Thus, derived objects must not expect such users to obey preconditions that are stronger than those required by the base class. That is, they must accept anything that the base class could accept. Also, derived classes must conform to all the postconditions of the base. That is, their behaviors and outputs must not violate any of the constraints established for the base class. Users of the base class must not be confused by the output of the derived class.

Clearly, the postcondition of `Square::SetWidth(double w)` is weaker⁸ than the postcondition of `Rectangle::SetWidth(double w)`, since it does not enforce the constraint, `(itsHeight == old.itsHeight)`. Thus, the `SetWidth` method of `Square` violates the contract of the base class.

Certain languages, like Eiffel, have direct support for preconditions and postconditions. You can declare them and have the runtime system verify them for you. Neither C++ nor Java has such a feature. In these languages, we must manually consider the preconditions and postcondition of each method and make sure that Meyer’s rule is not violated. Moreover, it can be very helpful to document these preconditions and postconditions in the comments for each method.

Specifying Contracts in Unit Tests

Contracts can also be specified by writing unit tests. By thoroughly testing the behavior of a class, the unit tests make the behavior of the class clear. Authors of client code will want to review the unit tests so that they know what to reasonably assume about the classes they are using.

A Real Example

Enough of squares and rectangles! Does the LSP have a bearing on real software? Let’s look at a case study that comes from a project that I worked on a few years ago.

6. [Meyer97], Chapter 11, p. 331.

7. [Meyer97], p. 573, Assertion Redeclaration rule (1).

8. The term “weaker” can be confusing. X is weaker than Y if X does not enforce all the constraints of Y. It does not matter how many new constraints X enforces.

Motivation

In the early 1990s, I purchased a third-party class library that had some container classes. The containers were roughly related to the `Bags` and `Sets` of Smalltalk. There were two varieties of `Set` and two similar varieties of `Bag`. The first variety was called “bounded” and was based on an array. The second was called “unbounded” and was based on a linked list.

The constructor for `BoundedSet` specified the maximum number of elements the set could hold. The space for these elements was preallocated as an array within the `BoundedSet`. Thus, if the creation of the `BoundedSet` succeeded, we could be sure that it had enough memory. Since it was based on an array, it was very fast. There were no memory allocations performed during normal operation. And since the memory was preallocated, we could be sure that operating the `BoundedSet` would not exhaust the heap. On the other hand, it was wasteful of memory since it would seldom completely utilize all the space that it had preallocated.

`UnboundedSet`, on the other hand, had no declared limit on the number of elements it could hold. So long as there was heap memory available, the `UnboundedSet` would continue to accept elements. Therefore, it was very flexible. It was also economical in that it only used the memory necessary to hold the elements that it currently contained. It was also slow because it had to allocate and deallocate memory as part of its normal operation. Finally, there was a danger that its normal operation could exhaust the heap.

I was unhappy with the interfaces of these third-party classes. I did not want my application code to be dependent on them because I felt that I would want to replace them with better classes later. Thus, I wrapped the third-party containers in my own abstract interface as shown in Figure 10-2.

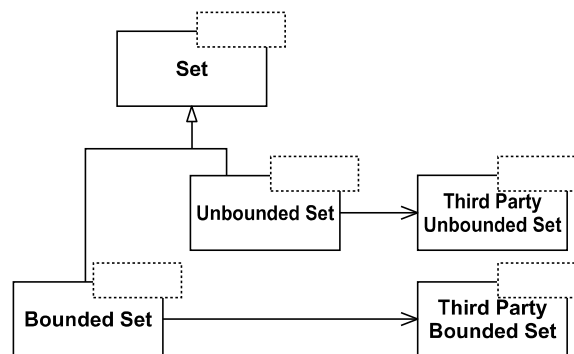


Figure 10-2 Container class adapter layer

I created an abstract class called `Set` that presented pure virtual `Add`, `Delete`, and `IsMember` functions, as shown in Listing 10-4. This structure unified the unbounded and bounded varieties of the two third-party sets and allowed them to be accessed through a common interface. Thus, some client could accept an argument of type `Set<T>&` and would not care whether the actual `Set` it worked on was of the bounded or unbounded variety. (See the `PrintSet` function in Listing 10-5.)

Listing 10-4

Abstract Set Class

```

template <class T>
class Set
{
public:
    virtual void Add(const T&) = 0;
    virtual void Delete(const T&) = 0;
    virtual bool IsMember(const T&) const = 0;
};
  
```

A Real Example

Listing 10-5

PrintSet

```
template <class T>
void PrintSet(const Set<T>& s)
{
    for (Iterator<T>i(s); i; i++)
        cout << (*i) << endl;
}
```

It is a big advantage not to have to know or care what kind of `Set` you are using. It means that the programmer can decide which kind of `Set` is needed in each particular instance, and none of the client functions will be affected by that decision. The programmer may choose an `UnboundedSet` when memory is tight and speed is not critical, or the programmer may choose an `BoundedSet` when memory is plentiful and speed is critical. The client functions will manipulate these objects through the interface of the base class `Set` and will therefore not know or care which kind of `Set` they are using.

Problem

I wanted to add a `PersistentSet` to this hierarchy. A persistent set is a set that can be written out to a stream and then read back in later, possibly by a different application. Unfortunately, the only third-party container that I had access to, that also offered persistence, was not a `template` class. Instead, it accepted objects that were derived from the abstract base class `PersistentObject`. I created the hierarchy shown in Figure 10-3.

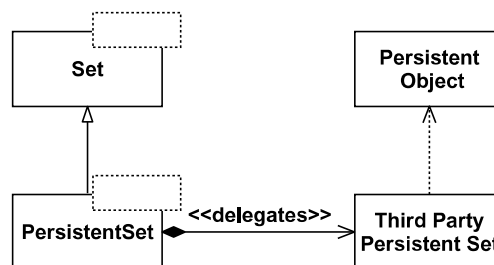


Figure 10-3 Persistent Set Hierarchy

Note that `PersistentSet` contains an instance of the third-party persistent set, to which it delegates all its methods. Thus, if you call `Add` on the `PersistentSet`, it simply delegates that to the appropriate method of the contained third-party persistent set.

On the surface of it, this might look all right. However, there is an ugly implication. Elements that are added to the third-party persistent set must be derived from `PersistentObject`. Since `PersistentSet` simply delegates to the third-party persistent set, any element added to `PersistentSet` must therefore derive from `PersistentObject`. Yet the interface of `Set` has no such constraint.

When a client is adding members to the base class `Set`, that client cannot be sure whether or not the `Set` might actually be a `PersistentSet`. Thus, the client has no way of knowing whether or not the elements it adds ought to be derived from `PersistentObject`.

Consider the code for `PersistentSet::Add()` in Listing 10-6.

Listing 10-6

```

template <typename T>
void PersistentSet::Add(const T& t)
{
    PersistentObject& p =
        dynamic_cast<PersistentObject&>(t);
    itsThirdPartyPersistentSet.Add(p);
}

```

This code makes it clear that if any client tries to add an object that is not derived from the class `PersistentObject` to my `PersistentSet`, a runtime error will ensue. The `dynamic_cast` will throw `bad_cast`. None of the existing clients of the abstract base class `Set` expects exceptions to be thrown on `Add`. Since these functions will be confused by a derivative of `Set`, this change to the hierarchy violates the LSP.

Is this a problem? Certainly. Functions that never before failed when passed a derivative of `Set` may now cause runtime errors when passed a `PersistentSet`. Debugging this kind of problem is relatively difficult since the runtime error occurs very far away from the actual logic flaw. The logic flaw is either the decision to pass a `PersistentSet` into a function or it is the decision to add an object to the `PersistentSet` that is not derived from `PersistentObject`. In either case, the actual decision might be millions of instructions away from the actual invocation of the `Add` method. Finding it can be a bear. Fixing it can be worse.

A Solution That Does *Not* Conform to the LSP

How do we solve this problem? Several years ago, I solved it by convention. Which is to say that I did not solve it in source code. Rather, I established a convention whereby `PersistentSet` and `PersistentObject` were not known to the application as a whole. They were only known to one particular module. This module was responsible for reading and writing all the containers to and from the persistent store. When a container needed to be written, its contents were copied into appropriate derivatives of `PersistentObject` and then added to `PersistentSets`, which were then saved on a stream. When a container needed to be read from a stream, the process was inverted. A `PersistentSet` was read from the stream, and then the `PersistentObjects` were removed from the `PersistentSet` and copied into regular (nonpersistent) objects, which were then added to a regular `Set`.

This solution may seem overly restrictive, but it was the only way I could think of to prevent `PersistentSet` objects from appearing at the interface of functions that would want to add nonpersistent objects to them. Moreover, it broke the dependency of the rest of the application on the whole notion of persistence.

Did this solution work? Not really. The convention was violated in several parts of the application by developers who did not understand the necessity for it. That is the problem with conventions—they have to be continually resold to each developer. If the developer has not learned the convention, or does not agree with it, then the convention will be violated. And one violation can compromise the whole structure.

An LSP-Compliant Solution

How would I solve this now? I would acknowledge that a `PersistentSet` does not have an IS-A relationship with `Set`, that it is not a proper derivative of `Set`. Thus, I would separate the hierarchies, but not completely. There are features that `Set` and `PersistentSet` have in common. In fact, it is only the `Add` method that causes the difficulty with LSP. Consequently, I would create a hierarchy in which both `Set` and `PersistentSet` were siblings beneath an abstract interface that allowed for membership testing, iteration, etc. (See Figure 10-4.) This would allow `PersistentSet` objects to be iterated and tested for membership, etc. But it would not afford the ability to add objects that were not derived from `PersistentObject` to a `PersistentSet`.

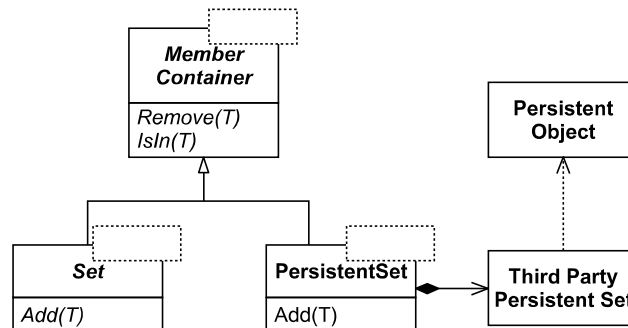


Figure 10-4 A solution that is LSP compliant

Factoring instead of Deriving

Another interesting and puzzling case of inheritance is the case of the `Line` and the `LineSegment`.⁹ Consider Listings 10-7 and 10-8. These two classes appear, at first, to be natural candidates for public inheritance. `LineSegment` needs every member variable and every member function declared in `Line`. Moreover, `LineSegment` adds a new member function of its own, `GetLength`, and overrides the meaning of the `IsOn` function. Yet these two classes violate the LSP in a subtle way.

Listing 10-7

geometry/line.h

```

#ifndef GEOMETRY_LINE_H
#define GEOMETRY_LINE_H
#include "geometry/point.h"

class Line
{
public:
    Line(const Point& p1, const Point& p2);

    double    GetSlope()          const;
    double    GetIntercept()      const; // Y Intercept
    Point     GetP1()             const {return itsP1;};
    Point     GetP2()             const {return itsP2;};
    virtual bool IsOn(const Point&) const;

private:
    Point itsP1;
    Point itsP2;
};
#endif
  
```

Listing 10-8

geometry/lineseg.h

```

#ifndef GEOMETRY_LINESEGMENT_H
#define GEOMETRY_LINESEGMENT_H
class LineSegment : public Line
  
```

9. Despite the similarity that this example has to the `Square/Rectangle` example, it comes from a real application and was subject to the real problems discussed.

```

{
    public:
        LineSegment(const Point& p1, const Point& p2);
        double      GetLength()      const;
        virtual bool IsOn(const Point&) const;
};
#endif

```

A user of `Line` has a right to expect that all points that are colinear with it are on it. For example, the point returned by the `Intercept` function is the point at which the line intersects the y-axis. Since this point is collinear with the line, users of `Line` have a right to expect that `IsOn(Intercept()) == true`. In many instances of `LineSegment`, however, this statement will fail.

Why is this an important issue? Why not simply derive `LineSegment` from `Line` and live with the subtle problems? This is a judgment call. There are *rare* occasions when it is more expedient to accept a subtle flaw in polymorphic behavior than to attempt to manipulate the design into complete LSP compliance. Accepting compromise instead of pursuing perfection is an engineering trade-off. A good engineer learns when compromise is more *profitable* than perfection. However, conformance to the LSP *should not be surrendered lightly*. The guarantee that a subclass will always work where its base classes are used is a powerful way to manage complexity. Once it is forsaken, we must consider each subclass individually.

In the case of the `Line` and `LineSegment`, there is a simple solution that illustrates an important tool of OOD. If we have access to both the `Line` and `LineSegment` classes, then we can *factor* the common elements of both into an abstract base class. Listings 10-9 through 10-11 show the factoring of `Line` and `LineSegment` into the base class `LinearObject`.

Listing 10-9

geometry/linearobj.h

```

#ifndef GEOMETRY_LINEAR_OBJECT_H
#define GEOMETRY_LINEAR_OBJECT_H

#include "geometry/point.h"

class LinearObject
{
    public:
        LinearObject(const Point& p1, const Point& p2);

        double  GetSlope() const;
        double  GetIntercept() const;

        Point GetP1() const {return itsP1;};
        Point GetP2() const {return itsP2;};
        virtual int  IsOn(const Point&) const = 0; // abstract.

    private:
        Point itsP1;
        Point itsP2;
};
#endif

```

Listing 10-10

geometry/line.h

```
#ifndef GEOMETRY_LINE_H
#define GEOMETRY_LINE_H
#include "geometry/linearobj.h"

class Line : public LinearObject
{
public:
    Line(const Point& p1, const Point& p2);
    virtual bool IsOn(const Point&) const;
};
#endif
```

Listing 10-11

geometry/lineseg.h

```
#ifndef GEOMETRY_LINESEGMENT_H
#define GEOMETRY_LINESEGMENT_H
#include "geometry/linearobj.h"

class LineSegment : public LinearObject
{
public:
    LineSegment(const Point& p1, const Point& p2);

    double      GetLength()      const;
    virtual bool IsOn(const Point&) const;
};
#endif
```

LinearObject represents both Line and LineSegment. It provides most of the functionality and data members for both subclasses, with the exception of the IsOn method, which is pure virtual. Users of LinearObject are not allowed to assume that they understand the extent of the object they are using. Thus, they can accept either a Line or a LineSegment with no problem. Moreover, users of Line will never have to deal with a LineSegment.

Factoring is a design tool that is best applied before there is much code written. Certainly, if there were dozens of clients of the Line class shown in Listing 10-7, we would not have had an easy time of factoring out the LinearObject class. When factoring is possible, however, it is a powerful tool. If qualities can be factored out of two subclasses, there is the distinct possibility that other classes will show up later that need those qualities, too. Of factoring, Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener say:

We can state that if a set of classes all support a common responsibility, they should inherit that responsibility from a common superclass.

If a common superclass does not already exist, create one, and move the common responsibilities to it. After all, such a class is demonstrably useful—you have already shown that the responsibilities will be inherited by some classes. Isn't it conceivable that a later extension of your system might add a new subclass that will support those same responsibilities in a new way? This new superclass will probably be an abstract class.¹⁰

10. [WirfsBrock90], p. 113.

Listing 10-12 shows how the attributes of `LinearObject` can be used by an unanticipated class, `Ray`. A `Ray` is substitutable for a `LinearObject`, and no user of `LinearObject` would have any trouble dealing with it.

Listing 10-12

geometry/ray.h

```
#ifndef GEOMETRY_RAY_H
#define GEOMETRY_RAY_H

class Ray : public LinearObject
{
public:
    Ray(const Point& p1, const Point& p2);
    virtual bool IsOn(const Point&) const;
};
#endif
```

Heuristics and Conventions

There are some simple heuristics that can give you some clues about LSP violations. They all have to do with derivative classes that somehow *remove* functionality from their base classes. A derivative that does less than its base is usually not substitutable for that base, and therefore violates the LSP.

Degenerate Functions in Derivatives

Consider Listing 10-13. The `f` function in `Base` is implemented. However, in `Derived` it is degenerate. Presumably, the author of `Derived` found that function `f` had no useful purpose in a `Derived`. Unfortunately, the users of `Base` don't know that they shouldn't call `f`, so there is a substitution violation.

Listing 10-13

A degenerate function in a derivative

```
public class Base
{
    public void f() { /*some code*/ }
}

public class Derived extends Base
{
    public void f() {}
}
```

The presence of degenerate functions in derivatives is not always indicative of an LSP violation, but it's worth looking at them when they occur.

Throwing Exceptions from Derivatives

Another form of violation is the addition of exceptions to methods of derived classes whose bases don't throw them. If the users of the base classes don't expect exceptions, then adding them to the methods of derivatives is not substitutable. Either the expectations of the users must be altered or the derived classes should not throw the exceptions.

Conclusion

Conclusion

The OCP is at the heart of many of the claims made for OOD. When this principle is in effect, applications are more maintainable, reusable, and robust. The LSP is one of the prime enablers of the OCP. It is the substitutability of subtypes that allows a module, expressed in terms of a base type, to be extensible without modification. That substitutability must be something that developers can depend on implicitly. Thus, the contract of the base type has to be well and prominently understood, if not explicitly enforced, by the code.

The term “IS-A” is too broad to act as a definition of a subtype. The true definition of a subtype is “substitutable,” where substitutability is defined by either an explicit or implicit contract.

Bibliography

1. Meyer, Bertrand. *Object-Oriented Software Construction*, 2d ed. Upper Saddle River, NJ: Prentice Hall, 1997.
2. Wirfs-Brock, Rebecca, et al. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall, 1990.
3. Liskov, Barbara. Data Abstraction and Hierarchy. *SIGPLAN Notices*, 23,5 (May 1988).

DIP: The Dependency-Inversion Principle



*Nevermore
Let the great interests of the State depend
Upon the thousand chances that may sway
A piece of human frailty*
—Sir Thomas Noon Talfourd (1795–1854)

DIP: The Dependency-Inversion Principle

- a. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
- b. Abstractions should not depend on details. Details should depend on abstractions.*

Over the years, many have questioned why I use the word “inversion” in the name of this principle. It is because more traditional software development methods, such as Structured Analysis and Design, tend to create software structures in which high-level modules depend on low-level modules, and in which policy depends on detail. Indeed one of the goals of these methods is to define the subprogram hierarchy that describes how the high-level modules make calls to the low-level modules. The initial design of the `COPY` program in Figure 7-1 on page 90 is a good example of such a hierarchy. The dependency structure of a well-designed, object-oriented program is “inverted” with respect to the dependency structure that normally results from traditional procedural methods.

Consider the implications of high-level modules that depend on low-level modules. It is the high-level modules that contain the important policy decisions and business models of an application. These modules

contain the identity of the application. Yet, when these modules depend on the lower level modules, changes to the lower level modules can have direct effects on the higher level modules and can force them to change in turn.

This predicament is absurd! It is the high-level, policy-setting modules that ought to be influencing the low-level, detailed modules. The modules that contain the high-level business rules should take precedence over, and be independent of, the modules that contain the implementation details. High-level modules simply should not depend on low-level modules in any way.

Moreover, it is high-level, policy-setting modules that we want to be able to reuse. We are already quite good at reusing low-level modules in the form of subroutine libraries. When high-level modules depend on low-level modules, it becomes very difficult to reuse those high-level modules in different contexts. However, when the high-level modules are independent of the low-level modules, then the high-level modules can be reused quite simply. This principle is at the very heart of framework design.

Layering

According to Booch, “...all well-structured object-oriented architectures have clearly defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface.”¹ A naive interpretation of this statement might lead a designer to produce a structure similar to Figure 11-1. In this diagram, the high-level `Policy` layer uses a lower-level `Mechanism` layer, which in turn uses a detailed-level `Utility` layer. While this may look appropriate, it has the insidious characteristic that the `Policy` layer is sensitive to changes all the way down in the `Utility` layer. *Dependency is transitive*. The `Policy` layer depends on something that depends on the `Utility` layer; thus, the `Policy` layer transitively depends on the `Utility` layer. This is very unfortunate.

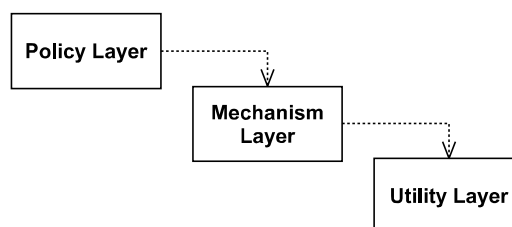


Figure 11-1 Naive layering scheme

Figure 11-2 shows a more appropriate model. Each of the upper-level layers declares an abstract interface for the services that it needs. The lower-level layers are then realized from these abstract interfaces. Each higher-level class uses the next-lowest layer through the abstract interface. Thus, the upper layers do not depend on the lower layers. Instead, the lower layers depend on abstract service interfaces *declared in* the upper layers. Not only is the transitive dependency of `PolicyLayer` on `UtilityLayer` broken, but even the direct dependency of the `PolicyLayer` on `MechanismLayer` is broken.

An Inversion of Ownership

Notice that the inversion here is not just one of dependencies, it is also one of interface ownership. We often think of utility libraries as owning their own interfaces. But when the DIP is applied, we find that the clients tend to own the abstract interfaces and that their servers derive from them.

1. [Booch96], p. 54.

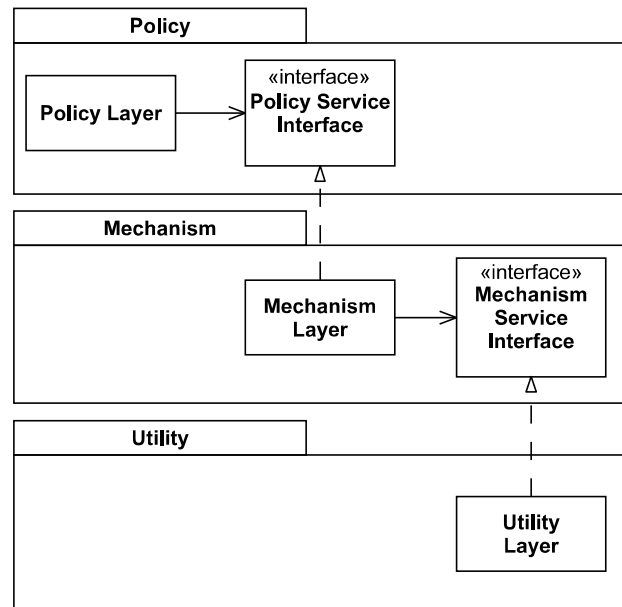


Figure 11-2 Inverted Layers

This is sometimes known as the Hollywood principle: “Don’t call us, we’ll call you.”² The lower-level modules provide the implementation for interfaces that are declared within, and called by, the upper-level modules.

Using this inversion of ownership, `PolicyLayer` is unaffected by any changes to `MechanismLayer` or `UtilityLayer`. Moreover, `PolicyLayer` can be reused in any context that defines lower-level modules that conform to the `PolicyServiceInterface`. Thus, by inverting the dependencies, we have created a structure, which is simultaneously more flexible, durable, and mobile.

Depend On Abstractions

A somewhat more naive, yet still very powerful, interpretation of the DIP is the simple heuristic: “Depend on abstractions.” Simply stated, this heuristic recommends that you should not depend on a concrete class—that all relationships in a program should terminate on an abstract class or an interface.

According to this heuristic,

- No variable should hold a pointer or reference to a concrete class.
- No class should derive from a concrete class.
- No method should override an implemented method of any of its base classes.

Certainly this heuristic is usually violated at least once in every program. Somebody has to create the instances of the concrete classes, and whatever module does that will depend on them.³ Moreover, there seems no reason to follow this heuristic for classes that are concrete but nonvolatile. If a concrete class is not going to change very much, and no other similar derivatives are going to be created, then it does very little harm to depend on it.

2. [Sweet85].

3. Actually, there are ways around this if you can use strings to create classes. Java allows this. So do several other languages. In such languages, the names of the concrete classes can be passed into the program as configuration data.

For example, in most systems the class that describes a string is concrete. In Java, for example, it is the concrete class `String`. This class is not volatile. That is, it does not change very often. Therefore it does no harm to depend directly on it.

However, most concrete classes that *we* write as part of an application program are volatile. It is *those* concrete classes that we do not want to depend directly on. Their volatility can be isolated by keeping them behind an abstract interface.

This is not a complete solution. There are times when the interface of a volatile class must change, and this change must be propagated to the abstract interface that represents the class. Such changes break through the isolation of the abstract interface.

This is the reason that the heuristic is a bit naive. If, on the other hand, we take the longer view that the client classes declare the service interfaces that they need, then the only time the interface will change is when the *client* needs the change. Changes to the classes that implement the abstract interface will not affect the client.

A Simple Example

Dependency inversion can be applied wherever one class sends a message to another. For example, consider the case of the `Button` object and the `Lamp` object.

The `Button` object senses the external environment. On receiving the `Poll` message, it determines whether or not a user has “pressed” it. It doesn’t matter what the sensing mechanism is. It could be a button icon on a GUI, a physical button being pressed by a human finger, or even a motion detector in a home security system. The `Button` object detects that a user has either activated or deactivated it.

The `Lamp` object affects the external environment. On receiving a `TurnOn` message, it illuminates a light of some kind. On receiving a `TurnOff` message, it extinguishes that light. The physical mechanism is unimportant. It could be an LED on a computer console, a mercury vapor lamp in a parking lot, or even the laser in a laser printer.

How can we design a system such that the `Button` object controls the `Lamp` object? Figure 11-3 shows a naive design. The `Button` object receives `Poll` messages, determines if the button has been pressed, and then simply sends the `TurnOn` or `TurnOff` message to the `Lamp`.

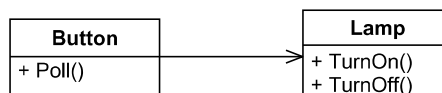


Figure 11-3 Naive Model of a Button and a Lamp

Why is this naive? Consider the Java code that is implied by this model (Listing 11-1). Note that the `Button` class depends directly on the `Lamp` class. This dependency implies that `Button` will be affected by changes to `Lamp`. Moreover, it will not be possible to reuse `Button` to control a `Motor` object. In this design, `Button` objects control `Lamp` objects, and *only* `Lamp` objects.

Listing 11-1

Button.java

```

public class Button
{
    private Lamp itsLamp;
    public void poll()
    {
        if (/*some condition*/)
            itsLamp.turnOn();
    }
}
  
```

A Simple Example

This solution violates the DIP. The high-level policy of the application has not been separated from the low-level implementation. The abstractions have not been separated from the details. Without such a separation, the high-level policy automatically depends on the low-level modules, and the abstractions automatically depend on the details.

Finding the Underlying Abstraction

What is the high-level policy? It is the abstraction that underlies the application, the truths that do not vary when the details are changed. It is the system *inside* the system—it is the metaphor. In the `Button/Lamp` example, the underlying abstraction is to detect an on/off gesture from a user and relay that gesture to a target object. What mechanism is used to detect the user gesture? Irrelevant! What is the target object? Irrelevant! These are details that do not impact the abstraction.

The design in Figure 11-3 can be improved by inverting the dependency on the `Lamp` object. In Figure 11-4, we see that the `Button` now holds an association to something called a `ButtonServer`. `ButtonServer` provides abstract methods that `Button` can use to turn something on or off. `Lamp` implements the `ButtonServer` interface. Thus, `Lamp` is now doing the depending, rather than being depended on.

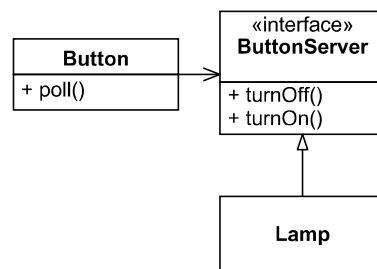


Figure 11-4 Dependency Inversion Applied to the Lamp

The design in Figure 11-4 allows a `Button` to control any device that is willing to implement the `ButtonServer` interface. This gives us a great deal of flexibility. It also means that `Button` objects will be able to control objects that have not yet been invented.

However, this solution also puts a constraint on any object that needs to be controlled by a `Button`. Such an object *must* implement the `ButtonServer` interface. This is unfortunate because these objects may also want to be controlled by a `Switch` object or some object other than a `Button`.

By inverting the direction of the dependency and making the `Lamp` do the depending instead of being depended on, we have made `Lamp` depend on a different detail—`Button`. Or have we?

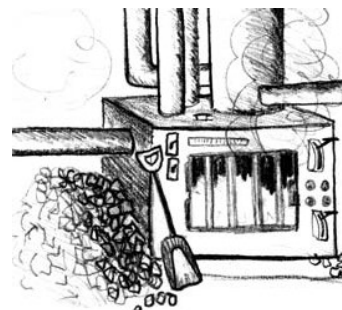
`Lamp` certainly depends on `ButtonServer`, but `ButtonServer` does not depend on `Button`. Any kind of object that knows how to manipulate the `ButtonServer` interface will be able to control a `Lamp`. Thus, the dependency is in name only. And we can fix that by changing the name of `ButtonServer` to something a bit more generic like `SwitchableDevice`. We can also ensure that `Button` and `SwitchableDevice` are kept in separate libraries, so that the use of `SwitchableDevice` does not imply the use of `Button`.

In this case, nobody owns the interface. We have the interesting situation where the interface can be used by lots of different clients and implemented by lots of different servers. Thus, the interface needs to stand alone without belonging to either group. In C++, we would put it in a separate namespace and library. In Java we would put it in a separate package.⁴

4. In dynamic languages like Smalltalk, Python, or Ruby, the interface simply wouldn't exist as an explicit source-code entity.

The Furnace Example

Let's look at a more interesting example. Consider the software that might control the regulator of a furnace. The software can read the current temperature from an IO channel and instruct the furnace to turn on or off by sending commands to a different IO channel. The structure of the algorithm might look something like Listing 11-2.



Listing 11-2

Simple algorithm for a thermostat

```
#define THERMOMETER 0x86
#define FURNACE     0x87
#define ENGAGE      1
#define DISENGAGE   0

void Regulate(double minTemp, double maxTemp)
{
    for(;;)
    {
        while (in(THERMOMETER) > minTemp)
            wait(1);
        out(FURNACE, ENGAGE);

        while (in(THERMOMETER) < maxTemp)
            wait(1);
        out(FURNACE, DISENGAGE);
    }
}
```

The high-level intent of the algorithm is clear, but the code is cluttered with lots of low-level details. This code could never be reused with different control hardware.

This may not be much of a loss since the code is very small. But even so, it is a shame to have the algorithm lost for reuse. We'd rather invert the dependencies and see something like Figure 11-5.

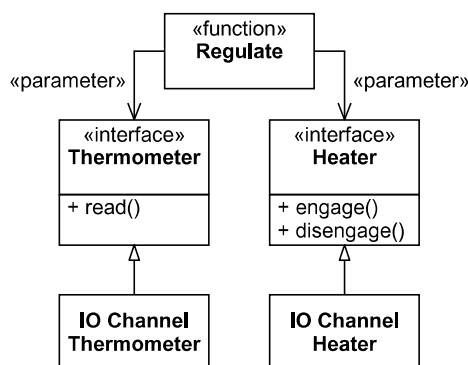


Figure 11-5 Generic Regulator

This shows that the regulate function takes two arguments that are both interfaces. The `Thermometer` interface can be read, and the `Heater` interface can be engaged and disengaged. This is all the `Regulate` algorithm needs. Now it can be written as shown in Listing 11-3.

Listing 11-3

Generic Regulator

```
void Regulate(Thermometer& t, Heater& h,
              double minTemp, double maxTemp)
{
    for(;;)
    {
        while (t.Read() > minTemp)
            wait(1);
        h.Engage();

        while (t.Read() < maxTemp)
            wait(1);
        h.Disengage();
    }
}
```

This has inverted the dependencies such that the high-level regulation policy does not depend on any of the specific details of the thermometer or the furnace. The algorithm is nicely reusable.

Dynamic v. Static Polymorphism

We have achieved the inversion of the dependencies, and made `Regulate` generic, through the use of dynamic polymorphism (i.e., abstract classes or interfaces). However, there is another way. We could have used the static form of polymorphism afforded by C++ templates. Consider Listing 11-4.

Listing 11-4

```
template <typename THERMOMETER, typename HEATER>
class Regulate(THERMOMETER& t, HEATER& h,
              double minTemp, double maxTemp)
{
    for(;;)
    {
        while (t.Read() > minTemp)
            wait(1);
        h.Engage();

        while (t.Read() < maxTemp)
            wait(1);
        h.Disengage();
    }
}
```

This achieves the same inversion of dependencies without the overhead (or flexibility) of dynamic polymorphism. In C++, the `Read`, `Engage`, and `Disengage` methods could all be nonvirtual. Moreover, any class that declares these methods can be used by the template. They do not have to inherit from a common base.

As a template, `Regulate` does not depend on any particular implementation of these functions. All that is required is that the class substituted for `HEATER` have an `Engage` and a `Disengage` method and that the class substituted for `THERMOMETER` have a `Read` function. Thus, those classes must implement the interface defined by the template. In other words, both `Regulate`, and the classes that `Regulate` uses, must agree on the same interface, and they both depend on that agreement.

Static polymorphism breaks the source-code dependency nicely, but it does not solve as many problems as does dynamic polymorphism. The disadvantages of the template approach are (1) The types of HEATER and THERMOMETER cannot be changed at runtime; and (2) The use of a new kind of HEATER or THERMOMETER will force recompilation and redeployment. So unless you have an extremely stringent requirement for speed, dynamic polymorphism should be preferred.

Conclusion

Traditional procedural programming creates a dependency structure in which policy depends on detail. This is unfortunate since the policies are then vulnerable to changes in the details. Object-oriented programming inverts that dependency structure such that both details and policies depend on abstraction, and service interfaces are often owned by their clients.

Indeed, it is this inversion of dependencies that is the hallmark of good object-oriented design. It doesn't matter what language a program is written in. If its dependencies are inverted, it has an OO design. If its dependencies are not inverted, it has a procedural design.

The principle of dependency inversion is the fundamental low-level mechanism behind many of the benefits claimed for object-oriented technology. Its proper application is necessary for the creation of reusable frameworks. It is also critically important for the construction of code that is resilient to change. Since the abstractions and details are all isolated from each other, the code is much easier to maintain.

Bibliography

1. Booch, Grady. *Object Solutions*. Menlo Park, CA: Addison-Wesley, 1996.
2. Gamma, et al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
3. Sweet, Richard E. The Mesa Programming Environment. *SIGPLAN Notices*, 20(7) (July 1985): 216–229.

12

ISP: The Interface-Segregation Principle

This principle deals with the disadvantages of “fat” interfaces. Classes that have “fat” interfaces are classes whose interfaces are not cohesive. In other words, the interfaces of the class can be broken up into groups of methods. Each group serves a different set of clients. Thus, some clients use one group of member functions, and other clients use the other groups.

The ISP acknowledges that there are objects that require noncohesive interfaces; however, it suggests that clients should not know about them as a single class. Instead, clients should know about abstract base classes that have cohesive interfaces.

Interface Pollution

Consider a security system. In this system, there are `Door` objects that can be locked and unlocked, and which know whether they are open or closed. (See Listing 12-1.)

Listing 12-1

```
Security Door
class Door
{
    public:
        virtual void Lock()    = 0;
        virtual void Unlock() = 0;
        virtual bool IsDoorOpen() = 0;
};
```

This class is abstract so that clients can use objects that conform to the `Door` interface, without having to depend on particular implementations of `Door`.

Now consider that one such implementation, `TimedDoor`, needs to sound an alarm when the door has been left open for too long. In order to do this, the `TimedDoor` object communicates with another object called a `Timer`. (See Listing 12-2.)

Listing 12-2

```

class Timer
{
    public:
        void Register(int timeout, TimerClient* client);
};

class TimerClient
{
    public:
        virtual void TimeOut() = 0;
};

```

When an object wishes to be informed about a time-out, it calls the `Register` function of the `Timer`. The arguments of this function are the time of the time-out, and a pointer to a `TimerClient` object whose `TimeOut` function will be called when the time-out expires.

How can we get the `TimerClient` class to communicate with the `TimedDoor` class so that the code in the `TimedDoor` can be notified of the time-out? There are several alternatives. Figure 12-1 shows a naive solution. We force `Door`, and therefore `TimedDoor`, to inherit from `TimerClient`. This ensures that `TimerClient` can register itself with the `Timer` and receive the `TimeOut` message.

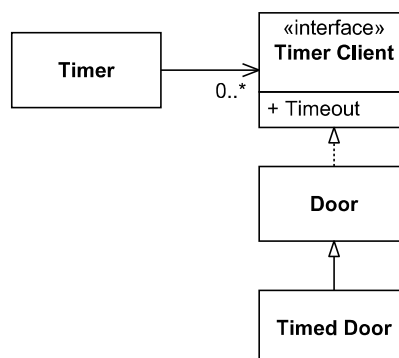


Figure 12-1 Timer Client at Top of Hierarchy

Although this solution is common, it is not without problems. Chief among these is that the `Door` class now depends on `TimerClient`. Not all varieties of `Door` need timing. Indeed, the original `Door` abstraction had nothing whatever to do with timing. If timing-free derivatives of `Door` are created, those derivatives will have to provide degenerate implementations for the `TimeOut` method—a potential violation of the LSP. Moreover, the applications that use those derivatives will have to import the definition of the `TimerClient` class, even though it is not used. That smells of *Needless Complexity* and *Needless Redundancy*.

This is an example of interface pollution, a syndrome that is common in statically typed languages like C++ and Java. The interface of `Door` has been polluted with a method that it does not require. It has been forced to incorporate this method solely for the benefit of one of its subclasses. If this practice is pursued, then every time a derivative needs a new method, that method will be added to the base class. This will further pollute the interface of the base class, making it “fat.”

Moreover, each time a new method is added to the base class, that method must be implemented (or allowed to default) in derived classes. Indeed, an associated practice is to add these interfaces to the base class giving them degenerate implementations, specifically so that derived classes are not burdened with the need to implement them. As we learned previously, such a practice can violate the LSP, leading to maintenance and reusability problems.

Separate Clients Mean Separate Interfaces

`Door` and `TimerClient` represent interfaces that are used by completely different clients. `Timer` uses `TimerClient`, and classes that manipulate doors use `Door`. Since the clients are separate, the interfaces should remain separate, too. Why? Because clients exert forces on the interfaces they use.

The Backwards Force Applied by Clients On Interfaces

When we think of forces that cause changes in software, we normally think about how changes to interfaces will affect their users. For example, we would be concerned about the changes to all the users of `TimerClient` if the `TimerClient` interface changed. However, there is a force that operates in the other direction. Sometimes it is the *user* that forces a change to the interface.

For example, some users of `Timer` will register more than one time-out request. Consider the `TimedDoor`. When it detects that the `Door` has been opened, it sends the `Register` message to the `Timer`, requesting a time-out. However, before that time-out expires, the door closes, remains closed for a while, and then opens again. This causes us to register a *new* time-out request before the old one has expired. Finally, the first time-out request expires and the `Timeout` function of the `TimedDoor` is invoked. The `Door` alarms falsely.

We can correct this situation by using the convention shown in Listing 12-3. We include a unique `timeOutId` code in each time-out registration, and we repeat that code in the `Timeout` call to the `TimerClient`. This allows each derivative of `TimerClient` to know which time-out request is being responded to.

Listing 12-3

```
Timer with ID
class Timer
{
    public:
        void Register(int timeout,
                    int timeOutId,
                    TimerClient* client);
};

class TimerClient
{
    public:
        virtual void Timeout(int timeOutId) = 0;
};
```

Clearly this change will affect all the users of `TimerClient`. We accept this since the lack of the `timeOutId` is an oversight that needs correction. However, the design in Figure 12-1 will also cause `Door`, and all clients of `Door` to be affected by this fix! This smells of *Rigidity* and *Viscosity*. Why should a bug in `TimerClient` have *any* affect on clients of `Door` derivatives that do not require timing? When a change in one part of the program affects other completely unrelated parts of the program, the cost and repercussions of changes become unpredictable, and the risk of fallout from the change increases dramatically.

ISP: The Interface-Segregation Principle

Clients should not be forced to depend on methods that they do not use.

When clients are forced to depend on methods that they don't use, then those clients are subject to changes to those methods. This results in an inadvertent coupling between all the clients. Said another way, when a client depends on a class that contains methods that the client does not use, but that other clients *do* use, then that client will be

affected by the changes that those other clients force upon the class. We would like to avoid such couplings where possible, and so we want to separate the interfaces.

Class Interfaces v. Object Interfaces

Consider the `TimedDoor` again. Here is an object which has two separate interfaces used by two separate clients—`Timer` and the users of `Door`. These two interfaces *must* be implemented in the same object, since the implementation of both interfaces manipulates the same data. So how can we conform to the ISP? How can we separate the interfaces when they must remain together?

The answer to this lies in the fact that clients of an object do not need to access it through the interface of the object. Rather, they can access it through delegation or through a base class of the object.

Separation through Delegation

One solution is to create an object that derives from `TimerClient` and delegates to the `TimedDoor`. Figure 12-2 shows this solution.

When the `TimedDoor` wants to register a time-out request with the `Timer`, it creates a `DoorTimerAdapter` and registers it with the `Timer`. When the `Timer` sends the `Timeout` message to the `DoorTimerAdapter`, the `DoorTimerAdapter` delegates the message back to the `TimedDoor`.

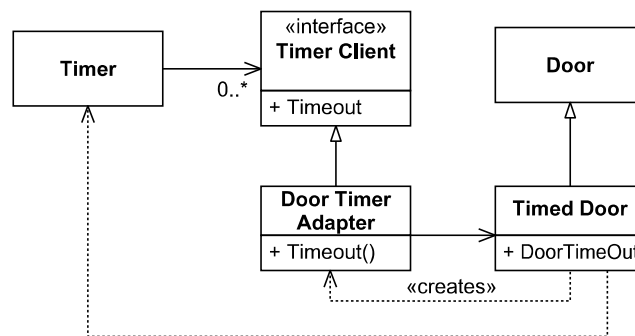


Figure 12-2 Door Timer Adapter

This solution conforms to the ISP and prevents the coupling of `Door` clients to `Timer`. Even if the change to `Timer` shown in Listing 12-3 were to be made, none of the users of `Door` would be affected. Moreover, `TimedDoor` does not have to have the exact same interface as `TimerClient`. The `DoorTimerAdapter` can *translate* the `TimerClient` interface into the `TimedDoor` interface. Thus, this is a very general purpose solution. (See Listing 12-4.)

Listing 12-4

TimedDoor.cpp

```

class TimedDoor : public Door
{
public:
    virtual void DoorTimeOut(int timeOutId);
};

class DoorTimerAdapter : public TimerClient
{
public:
    DoorTimerAdapter(TimedDoor& theDoor)
  
```

The ATM User Interface Example

```
    : itsTimedDoor(theDoor)
    {}

    virtual void TimeOut(int timeOutId)
    {itsTimedDoor.DoorTimeOut(timeOutId);}

private:
    TimedDoor& itsTimedDoor;
};
```

However, this solution is also somewhat inelegant. It involves the creation of a new object every time we wish to register a time-out. Moreover, the delegation requires a very small, but still nonzero, amount of runtime and memory. There are application domains, such as embedded real-time control systems, in which runtime and memory are scarce enough to make this a concern.

Separation through Multiple Inheritance

Figure 12-3 and Listing 12-5 show how multiple inheritance can be used to achieve the ISP. In this model, `TimedDoor` inherits from both `Door` and `TimerClient`. Although clients of both base classes can make use of `TimedDoor`, neither actually depends on the `TimedDoor` class. Thus, they use the same object through separate interfaces.

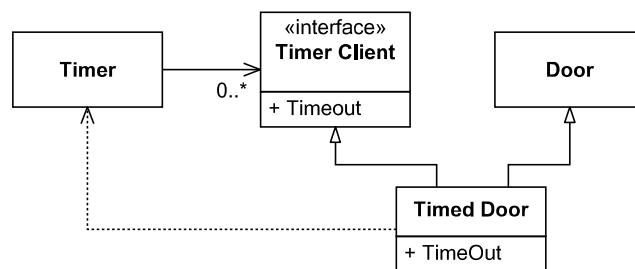


Figure 12-3 Multiply inherited Timed Door

Listing 12-5

TimedDoor.cpp

```
class TimedDoor : public Door, public TimerClient
{
public:
    virtual void TimeOut(int timeOutId);
};
```

This solution is my normal preference. The only time I would choose the solution in Figure 12-2 over Figure 12-3 is if the translation performed by the `DoorTimerAdapter` object were necessary, or if different translations were needed at different times.

The ATM User Interface Example

Now let's consider a slightly more significant example. The traditional automated teller machine (ATM) problem. The user interface of an ATM machine needs to be very flexible. The output may need to be translated into many different languages. It may need to be presented on a screen, or on a braille tablet, or spoken out a speech synthesizer. Clearly this can be achieved by creating an abstract base class that has abstract methods for all the different messages that need to be presented by the interface.



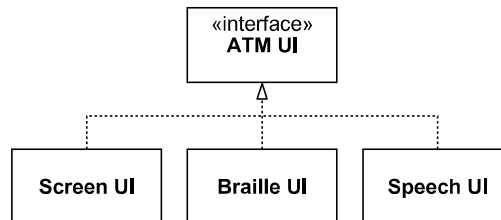


Figure 12-4

Consider also that each different transaction that the ATM can perform is encapsulated as a derivative of the class `Transaction`. Thus, we might have classes such as `DepositTransaction`, `WithdrawalTransaction`, and `TransferTransaction`. Each class invokes methods of the UI. For example, in order to ask the user to enter the amount he wishes to deposit, the `DepositTransaction` object invokes the `RequestDepositAmount` method of the UI class. Likewise, in order to ask the user how much money he wants to transfer between accounts, the `TransferTransaction` object calls the `RequestTransferAmount` method of UI. This corresponds to the diagram in Figure 12-5.

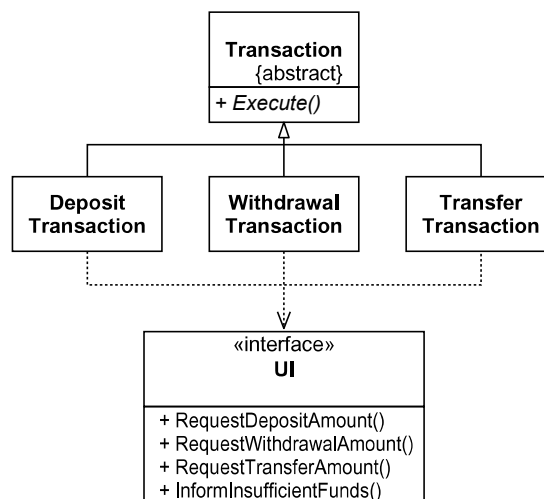


Figure 12-5 ATM Transaction Hierarchy

Notice that this is precisely the situation that the ISP tells us to avoid. Each transaction is using methods of the UI that no other class uses. This creates the possibility that changes to one of the derivatives of `Transaction` will force corresponding change to the UI, thereby affecting all the other derivatives of `Transaction` and every other class that depends on the UI interface. Something smells like Rigidity and Fragility around here.

For example, if we were to add a `PayGasBillTransaction`, we would have to add new methods to UI in order to deal with the unique messages that this transaction would want to display. Unfortunately, since `DepositTransaction`, `WithdrawalTransaction`, and `TransferTransaction` all depend on the UI interface, they must all be recompiled. Worse, if the transactions were all deployed as components in separate DLLs or shared libraries, then those components would have to be redeployed, even though none of their logic was changed. Can you smell the Viscosity?

The ATM User Interface Example

This unfortunate coupling can be avoided by segregating the UI interface into individual interfaces such as `DepositUI`, `WithdrawUI`, and `TransferUI`. These separate interfaces can then be multiply inherited into the final UI interface. Figure 12-6 and Listing 12-6 show this model.

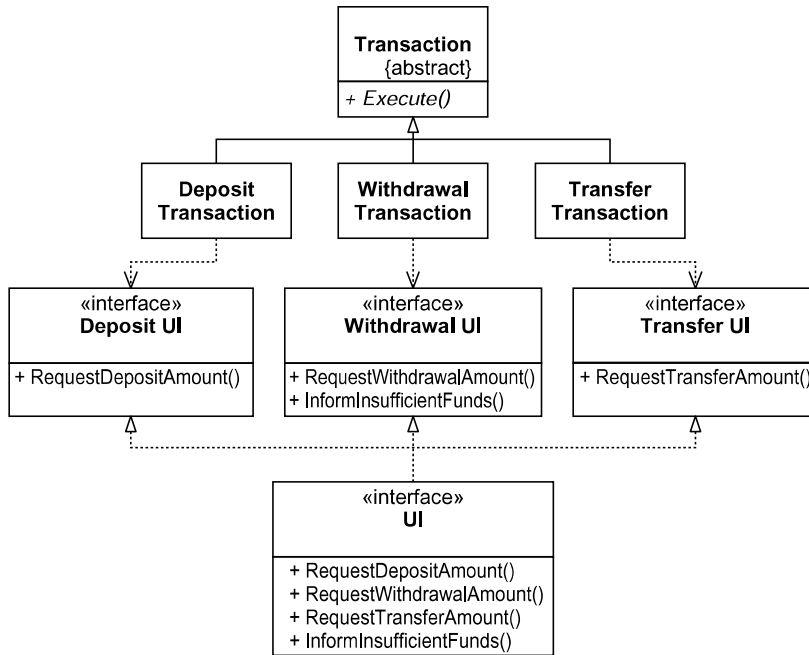


Figure 12-6 Segregated ATM UI Interface

Whenever a new derivative of the `Transaction` class is created, a corresponding base class for the abstract UI interface will be needed, and so the UI interface and all its derivatives must change. However, these classes are not widely used. Indeed, they are probably only used by `main` or whatever process boots the system and creates the concrete UI instance. So the impact of adding new UI base classes is minimized.

Listing 12-6

Segregated ATM UI Interface

```
class DepositUI
{
    public:
        virtual void RequestDepositAmount() = 0;
};

class DepositTransaction : public Transaction
{
    public:
        DepositTransaction(DepositUI& ui)
        : itsDepositUI(ui)
        {}

        virtual void Execute()
        {
            ...
            itsDepositUI.RequestDepositAmount();
        }
};
```

```
        ...
    }
private:
    DepositUI& itsDepositUI;
};

class WithdrawalUI
{
public:
    virtual void RequestWithdrawalAmount() = 0;
};

class WithdrawalTransaction : public Transaction
{
public:
    WithdrawalTransaction(WithdrawalUI& ui)
        : itsWithdrawalUI(ui)
    {}

    virtual void Execute()
    {
        ...
        itsWithdrawalUI.RequestWithdrawalAmount();
        ...
    }
private:
    WithdrawalUI& itsWithdrawalUI;
};

class TransferUI
{
public:
    virtual void RequestTransferAmount() = 0;
};

class TransferTransaction : public Transaction
{
public:
    TransferTransaction(TransferUI& ui)
        : itsTransferUI(ui)
    {}

    virtual void Execute()
    {
        ...
        itsTransferUI.RequestTransferAmount();
        ...
    }
private:
    TransferUI& itsTransferUI;
};

class UI : public DepositUI
        , public WithdrawalUI
        , public TransferUI
```

The ATM User Interface Example

```
{
    public:
        virtual void RequestDepositAmount();
        virtual void RequestWithdrawalAmount();
        virtual void RequestTransferAmount();
};
```

A careful examination of Listing 12-6 will show one of the issues with ISP conformance that was not obvious from the `TimedDoor` example. Note that each transaction must somehow know about its particular version of the UI. `DepositTransaction` must know about `DepositUI`, `WithdrawTransaction` must know about `WithdrawUI`, etc. In Listing 12-6, I have addressed this issue by forcing each transaction to be constructed with a reference to its particular UI. Note that this allows me to employ the idiom in Listing 12-7.

Listing 12-7

Interface Initialization Idiom

```
UI Gui; // global object;

void f()
{
    DepositTransaction dt(Gui);
}
```

This is handy, but it also forces each transaction to contain a reference member to its UI. Another way to address this issue is to create a set of global constants as shown in Listing 12-8. Global variables are not always a symptom of a poor design. In this case they provide the distinct advantage of easy access. Since they are references, it is impossible to change them in any way. Therefore they cannot be manipulated in a way that would surprise other users.

Listing 12-8

Seperate Global Pointers

```
// in some module that gets linked in
// to the rest of the app.

static UI Lui; // non-global object;
DepositUI& GdepositUI = Lui;
WithdrawalUI& GwithdrawalUI = Lui;
TransferUI& GtransferUI = Lui;

// In the depositTransaction.h module

class WithdrawalTransaction : public Transaction
{
    public:

        virtual void Execute()
        {
            ...
            GwithdrawalUI.RequestWithdrawalAmount();
            ...
        }
};
```

In C++, one might be tempted to put all the globals in Listing 12-8 into a single class in order to prevent pollution of the global namespace. Listing 12-9 shows such an approach. This, however, has an unfortunate effect. In order to use `UIGlobals`, you must `#include ui_globals.h`. This, in turn, `#includes` `depositUI.h`, `withdrawalUI.h`, and `transferUI.h`. This means that any module wishing to use any of the UI interfaces transitively depends on all of them— exactly the situation that the ISP warns us to avoid. If a change is made to any of the UI interfaces, all modules that `#include "ui_globals.h"` are forced to recompile. The `UIGlobals` class has recombined the interfaces that we had worked so hard to segregate!

Listing 12-9

Wrapping the Globals in a class

```
// in ui_globals.h

#include "depositUI.h"
#include "withdrawalUI.h"
#include "transferUI.h"

class UIGlobals
{
public:
    static WithdrawalUI& withdrawal;
    static DepositUI& deposit;
    static TransferUI& transfer
};

// in ui_globals.cc

static UI Lui; // non-global object;
DepositUI& UIGlobals::deposit = Lui;
WithdrawalUI& UIGlobals::withdrawal = Lui;
TransferUI& UIGlobals::transfer = Lui;
```

The Polyad v. the Monad

Consider a function `g` that needs access to both the `DepositUI` and the `TransferUI`. Consider also that we wish to pass the UIs into this function. Should we write the function prototype like this?

```
void g(DepositUI&, TransferUI&);
```

Or should we write it like this?

```
void g(UI&);
```

The temptation to write the latter (monadic) form is strong. After all, we know that in the former (polyadic) form, both arguments will refer to the *same object*. Moreover, if we were to use the polyadic form, its invocation might look like this:

```
g(ui, ui);
```

Somehow, this seems perverse.

Perverse or not, the polyadic form is often preferable to the monadic form. The monadic form forces `g` to depend on every interface included in `UI`. Thus, when `WithdrawalUI` changes, `g` and all clients of `g` could be affected. This is more perverse than `g(ui, ui)`! Moreover, we cannot be sure that both arguments of `g` will *always*

Conclusion

refer to the same object! In the future, it may be that the interface objects are separated for some reason. The fact that all interfaces are combined into a single object is information that \mathcal{G} does not need to know. Thus, I prefer the polyadic form for such functions.

Grouping Clients. Clients can often be grouped together by the service methods they call. Such groupings allow segregated interfaces to be created for each group instead of each client. This greatly reduces the number of interfaces that the service has to implement, and it also prevents the service from depending on each client type.

Sometimes, the methods invoked by different groups of clients will overlap. If the overlap is small, then the interfaces for the groups should remain separate. The common functions should be declared in all the overlapping interfaces. The server class will inherit the common functions from each of those interfaces, but it will implement them only once.

Changing Interfaces. When object-oriented applications are maintained, the interfaces to existing classes and components often change. There are times when these changes have a huge impact and force the recompilation and redeployment of a very large part of the system. This impact can be mitigated by adding new interfaces to existing objects, rather than changing the existing interface. Clients of the old interface that wish to access methods of the new interface can query the object for that interface, as shown in Listing 12-10.

Listing 12-10

```
void Client(Service* s)
{
    if (NewService* ns = dynamic_cast<NewService*>(s))
    {
        // use the new service interface
    }
}
```

As with all principles, care must be taken not to overdo it. The spectre of a class with hundreds of different interfaces, some segregated by client and others segregated by version, would be frightening indeed.

Conclusion

Fat classes cause bizarre and harmful couplings between their clients. When one client forces a change on the fat class, all the other clients are affected. Thus, clients should only have to depend on methods that they actually call. This can be achieved by breaking the interface of the fat class into many client-specific interfaces. Each client-specific interface declares only those functions that its particular client, or client group, invoke. The fat class can then inherit all the client-specific interfaces and implement them. This breaks the dependence of the clients on methods that they don't invoke, and it allows the clients to be independent of each other.

Bibliography

1. Gamma, et al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

