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 DeMarco[1] 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.

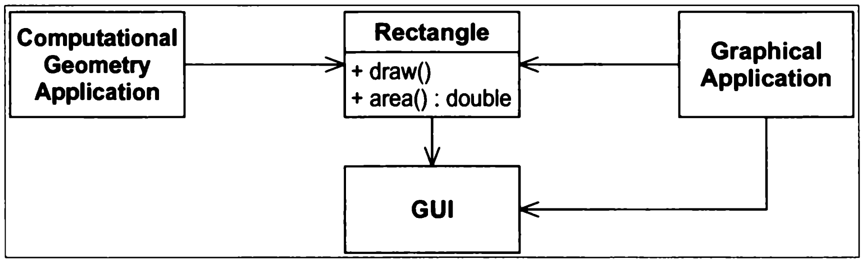


Figure 8-1 More than one responsibility

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`.

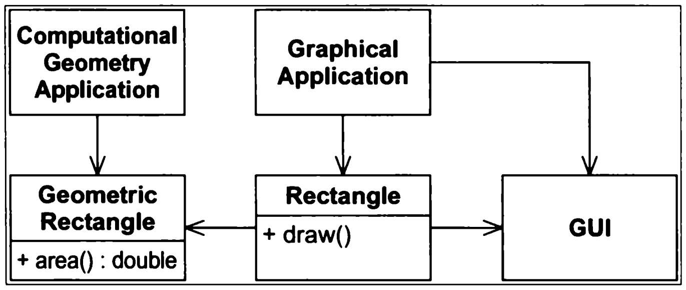


Figure 8-2 Separated Responsibilities

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 Rigidity 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.

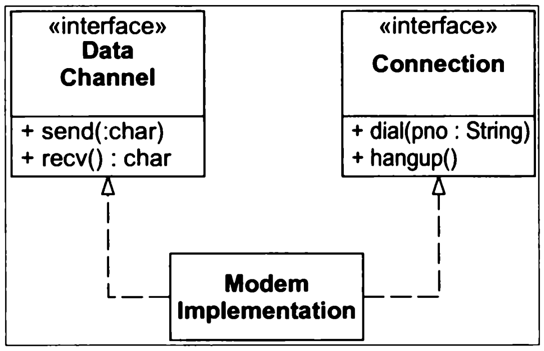


Figure 8-3 Separated Modem Interface

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.

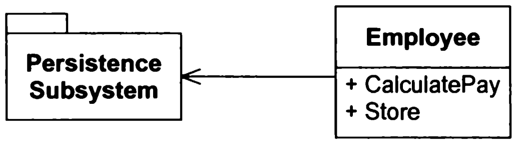


Figure 8-4 Coupled Persistence

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 Rigidity and Fragility become strong, the design should be refactored using the FACADE or PROXY patterns to separate the two responsibilities.

Conclusion

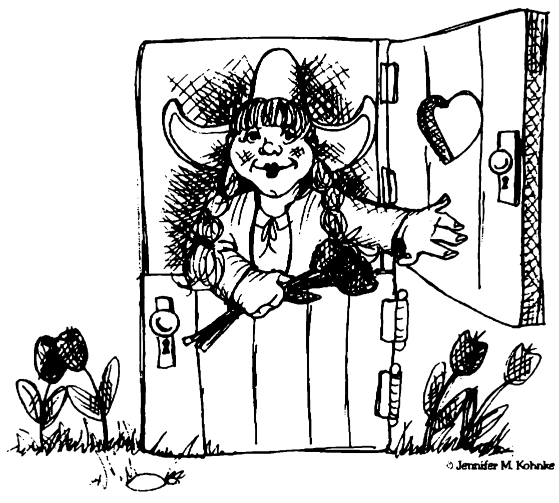
The SRP is one of the simplest of the principles, and one of the hardest to get right. Conjoining responsibilities is something that we do naturally. Finding and separating those responsibilities from one another is much of what software design is really about. Indeed, the rest of the principles we will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

9

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.

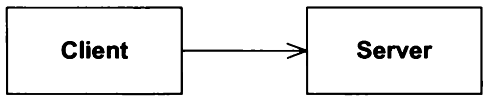


Figure 9-1 Client is not open and closed

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.

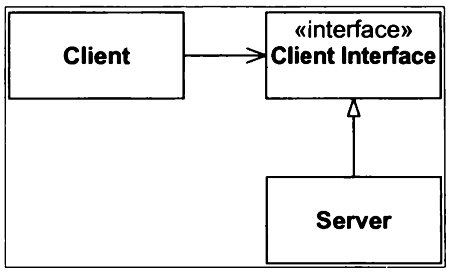


Figure 9-2 STRATEGY pattern: Client is both open and closed

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.

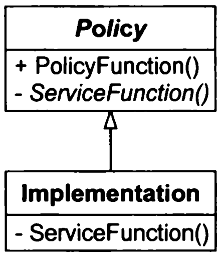


Figure 9-3 Template Method Pattern: Base class is open and closed

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 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`.

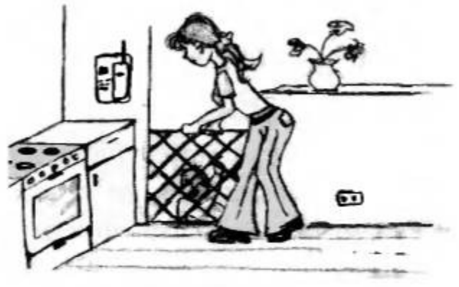
5. Such objects are known as *factories*, and we'll have more to say about them in Chapter 21 on page 269.

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;
    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 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.