9

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.* As we'll see in Chapter 21, we have a more specific definition of cohesion at the package level. However, at the class level the definition is similar.

SRP: The Single Responsibility Principle

> ***THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS 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 85.)

1. [DeMarco79], p310
2. [PageJones88], Chapter 6, p82.

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 then one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability 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 9-1. The `Rectangle` class has two methods shown. One draws the rectangle on the screen, the other computes the area of the rectangle.

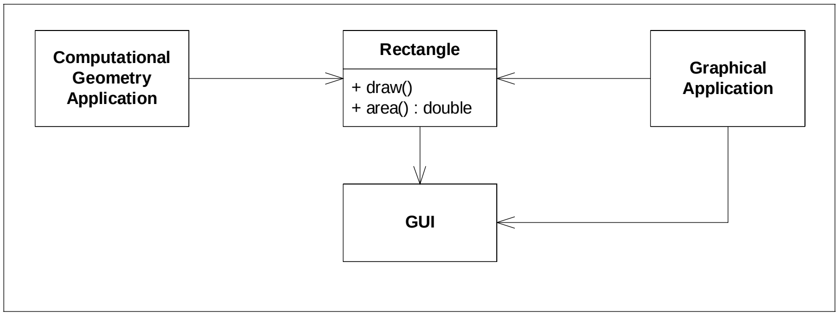


Figure 9-1
More than one responsibility

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 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.

The violation of SRP causes several nasty problems. Firstly, 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.

Secondly, 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 9-2. 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`.

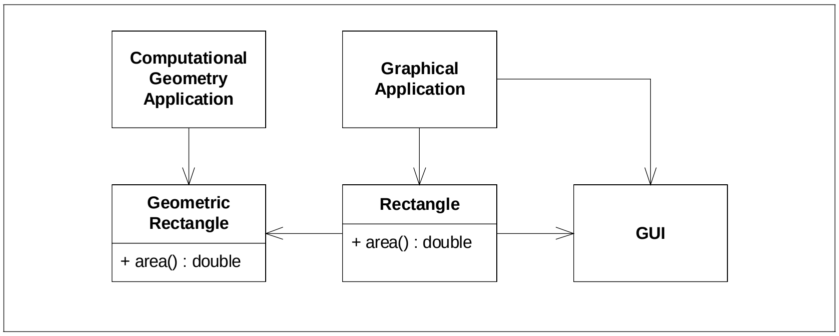


Figure 9-2
Separated Responsibilities

What is a Responsibility?

In the context of the Single Responsibility Principle (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 9-1. Most of us will agree that this interface looks perfectly reasonable. The four functions it declares are certainly functions belonging to a modem.

Listing 9-1
`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? Almost certainly they should. The two sets of functions have almost nothing in common. They'll certainly change for different reasons. Moreover, they will be called from completely different parts of the applications that use them. Those different parts will change for different reasons as well.

Therefore the design in Figure 9-3 is probably better. It separates the two responsibilities into two separate interfaces[3]. This, at least, keeps the client applications from coupling the two responsibilities.

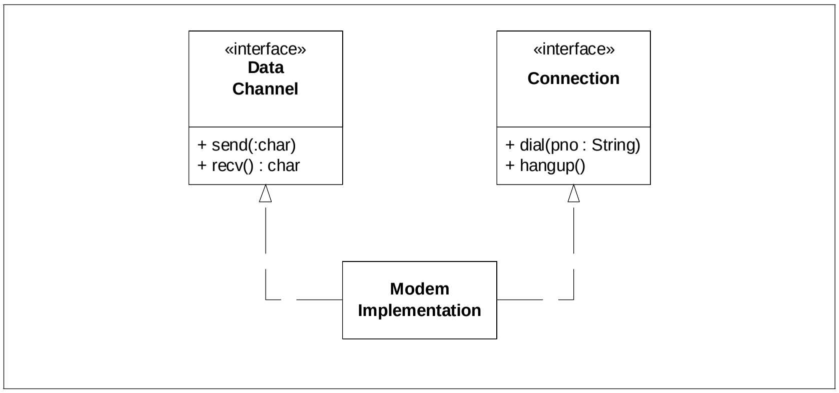


Figure 9-3
Separated Modem Interface

However, notice that I have recoupled the two responsibilities into a single `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 is a kludge, or a wart; however, notice that all dependencies flow *away* from it. Nobody need depend upon this class. Nobody except `main` needs to know that it exists. Thus, we've put the ugly bit behind a fence. It's ugliness need not leak out and pollute the rest of the application.

Conclusion

The SRP is one of the simplest of the principle, 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.

3. We'll see more of this in Chapter 13, when we study the Interface Segregation Principle (ISP).

Bibliography

[DeMarco79]: *Structured Analysis and System Specification,* Tom DeMarco, Yourdon Press Computing Series, 1979

[PageJones88]: *The Practical Guide to Structured Systems Design*, 2d. ed., Meilir Page-Jones, Yourdon Press Computing Series, 1988

The Open-Closed Principle

This is the first of my *Engineering Notebook* columns for *The C++ Report*. The articles that will appear in this column will focus on the use of C++ and OOD, and will address issues of software engineering. I will strive for articles that are pragmatic and directly useful to the software engineer in the trenches. In these articles I will make use of Booch's notation for documenting object oriented designs. The sidebar provides a brief lexicon of Booch's notation.

There are many heuristics associated with object oriented design. For example, "all member variables should be private", or "global variables should be avoided", or "using run time type identification (RTTI) is dangerous". What is the source of these heuristics? What makes them true? Are they *always* true? This column investigates the design principle that underlies these heuristics -- the open-closed principle.

As Ivar Jacobson said: "All systems change during their life cycles. This must be borne in mind when developing systems 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[2] gave us guidance as long ago as 1988 when he coined the now famous open-closed principle. To paraphrase him:

> ***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, that program exhibits the undesirable attributes that we have come to associate with "bad" design. The program becomes fragile, rigid, unpredictable and unreusable. The open-closed principle attacks this in a very straightforward way. It says that you should design modules that *never change*. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

Description

Modules that conform to the open-closed principle have two primary attributes.

1. Object Oriented Software Engineering a Use Case Driven Approach, Ivar Jacobson, Addison Wesley, 1992, p 21.
2. Object Oriented Software Construction, Bertrand Meyer, Prentice Hall, 1988, p 23

1. They are “Open For Extension”.

   This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.

2. They are “Closed for Modification”.

   The source code of such a module is inviolate. No one is allowed to make source code changes to it.

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 that module. A module that cannot be changed is normally thought to have a fixed behavior. How can these two opposing attributes be resolved?

Abstraction is the Key.

In C++, using the principles of object oriented design, 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 1 shows a simple design that does not conform to the open-closed principle. Both the `Client` and `Server` classes are concrete. There is no guarantee that the member functions of the `Server` class are virtual. 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.

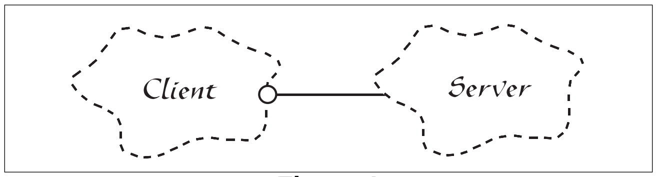


Figure 1
Closed Client

Figure 2 shows the corresponding design that conforms to the open-closed principle. In this case, the `AbstractServer` class is an abstract class with pure-virtual 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 `AbstractServer` class can be created. The `Client` class can remain unchanged.

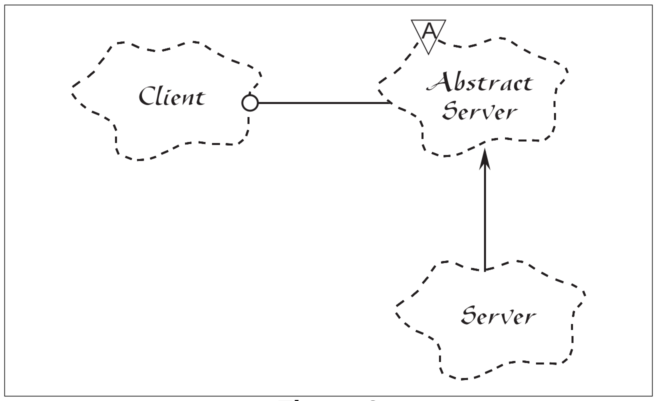


Figure 2
Open Client

The Shape Abstraction

Consider the following example. 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.

In C, using procedural techniques that do not conform to the open-closed principle, we might solve this problem as shown in Listing 1. Here we see a set of data structures that have the same first element, but are 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 1
Procedural Solution to the Square/Circle Problem

```
enum ShapeType {circle, square};

struct Shape
{
  ShapeType itsType;
};

struct Circle
{
  ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};
```

Listing 1 (Continued)
Procedural Solution to the Square/Circle Problem

```
struct Square
{
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};

//
// These functions are implemented elsewhere
//
void DrawSquare(struct Square*)
void DrawCircle(struct Circle*);

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 open-closed principle 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 over the application; each one doing something a little different. 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. Thus the problem of finding and understanding all the places where the new shape needs to be added can be non-trivial.

Listing 2 shows the code for a solution to the square/circle problem that conforms to the open-closed principle. In this case an abstract `Shape` class is created. This abstract class has a single pure-virtual function called `Draw`. Both `Circle` and `Square` are derivatives of the `Shape` class.

Listing 2

OOD solution to Square/Circle problem.

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

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

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

void DrawAllShapes(Set<Shape*>& list)
{
  for (Iterator<Shape*>i(list); i; i++)
     (*i)->Draw();
}
```

Note that if we want to extend the behavior of the `DrawAllShapes` function in Listing 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 open-closed principle. Its behavior can be extended without modifying it.

In the real world 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.

Since programs that conform to the open-closed principle are changed by adding new code, rather than by changing existing code, they do not experience the cascade of changes exhibited by non-conforming programs.

Strategic Closure

It should be clear that no significant program can be 100% closed. For example, consider what would happen to the `DrawAllShapes` function from Listing 2 if we decided that all `Circles` should be drawn before any `Squares`. The `DrawAllShapes` function is not closed against a change like this. In general, no matter how "closed" a module is, there will always be some kind of change against which it is not closed.

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. This takes a certain amount of prescience derived from experience. The experienced designer knows the users and the industry well enough to judge the probability of different kinds of changes. He then makes sure that the open-closed principle is invoked for the most probable changes.

Using Abstraction to Gain Explicit Closure.

How could 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". The specific case of ordering above had to do with drawing certain types of shapes before other types of shapes.

An ordering policy implies that, given any two objects, it is possible to discover which ought to be drawn first. Thus, we can define a method of `Shape` named `Precedes` that 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 ordered before the `Shape` object passed as the argument.

In C++ this function could be represented by an overloaded `operator<` function. Listing 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 4 shows the C++ code that does this. This code uses the `Set`, `OrderedSet` and `Iterator` classes from the `Components` category developed in my book[3] (if you would like a free copy of the source code of the `Components` category, send email to `rmartin@oma.com`).

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 5.

3. *Designing Object Oriented C++ Applications using the Booch Method*, Robert C. Martin, Prentice Hall, 1995.

Listing 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 4

DrawAllShapes with Ordering

```
void DrawAllShapes(Set<Shape*>& list)
{
    // copy elements into OrderedSet and then sort.
    OrderedSet<Shape*> orderedList = list;
    orderedList.Sort();

    for (Iterator<Shape*> i(orderedList); i; i++)
        (*i)->Draw();
}
```

Listing 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 does not conform to the open-closed principle. There is no way to close it against new derivatives of `Shape`. Every time a new derivative of `Shape` is created, this function will need to be changed.

Using a "Data Driven" Approach to Achieve Closure.

Closure of the derivatives of `Shape` can be achieved by using a table driven approach that does not force changes in every derived class. Listing 6 shows one possibility.

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.)

Listing 6

Table driven type ordering mechanism

```
#include <typeinfo.h>
#include <string.h>
enum {false, true};
typedef int bool;

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

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

char* Shape::typeOrderTable[] =
{
    “Circle”,
    “Square”,
    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;
```

Listing 6 (Continued)

Table driven type ordering mechanism

```
            if ((argOrd > 0) && (thisOrd > 0))
                done = true;
        }
        else // table entry == 0
            done = true;
    }
    return thisOrd < argOrd;
}
```

The only item that is not closed against the order of the various `Shapes` is the table itself. And 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.

Extending Closure Even Further.

This isn't the end of the story. We have managed to close the `Shape` hierarchy, and the `DrawAllShapes` function against ordering that is dependent upon the type of the shape. However, the `Shape` derivatives are not closed against ordering policies that have nothing to do with shape types. It seems likely that we will want to order the drawing of shapes according to some higher level structure. A complete exploration of these issues is beyond the scope of this article; however the ambitious reader might consider how to address this issue using an abstract `OrderedObject` class contained by the class `OrderedShape`, which is derived from both `Shape` and `OrderedObject`.

Heuristics and Conventions

As mentioned at the begining of this article, the open-closed principle is the root motivation behind many of the heuristics and conventions that have been published regarding OOD over the years. Here are some of the more important of them.

Make all Member Variables Private.

This is one of the most commonly held of all the conventions of OOD. Member variables of classes should be known only to the methods of the class that defines them. Member variables should never be known to any other class, including derived classes. Thus they should be declared `private`, rather than `public` or `protected`.

In light of the open-closed principle, the reason for this convention ought to be clear. When the member variables of a class change, every function that depends upon those variables must be changed. Thus, no function that depends upon a variable can be closed with respect to that variable.

In OOD, we expect that the methods of a class are not closed to changes in the member variables of that class. However we *do* expect that any other class, including subclasses *are closed* against changes to those variables. We have a name for this expectation, we call it: *encapsulation*.

Now, what if you had a member variable that you knew would never change? Is there any reason to make it `private`? For example, Listing 7 shows a class `Device` that has a `bool status` variable. This variable contains the status of the last operation. If that operation succeeded, then `status` will be `true`; otherwise it will be `false`.

Listing 7

non-const public variable

```
class Device
{
  public:
    bool status;
};
```

We know that the type or meaning of this variable is never going to change. So why not make it `public` and let client code simply examine its contents? If this variable really never changes, and if all other clients obey the rules and only query the contents of `status`, then the fact that the variable is `public` does no harm at all. However, consider what happens if even one client takes advantage of the writable nature of `status`, and changes its value. Suddenly, this one client could affect every other client of `Device`. This means that it is impossible to close any client of `Device` against changes to this one misbehaving module. This is probably far too big a risk to take.

On the other hand, suppose we have the `Time` class as shown in Listing 8. What is the harm done by the public member variables in this class? Certainly they are very unlikely to change. Moreover, it does not matter if any of the client modules make changes to the variables, the variables are supposed to be changed by clients. It is also very unlikely that a derived class might want to trap the setting of a particular member variable. So is any harm done?

Listing 8

```
class Time
{
  public:
    int hours, minutes, seconds;
    Time& operator-=(int seconds);
    Time& operator+=(int seconds);
    bool  operator< (const Time&);
    bool  operator> (const Time&);
    bool  operator==(const Time&);
    bool  operator!=(const Time&);
};
```

One complaint I could make about Listing 8 is that the modification of the time is not atomic. That is, a client can change the `minutes` variable without changing the `hours` variable. This may result in inconsistent values for a `Time` object. I would prefer it if there were a single function to set the time that took three arguments, thus making the setting of the time atomic. But this is a very weak argument.

It would not be hard to think of other conditions for which the `public` nature of these variables causes some problems. In the long run, however, there is no *overriding* reason to make these variables `private`. I still consider it bad *style* to make them `public`, but it is probably not bad *design.* I consider it bad style because it is very cheap to create the appropriate inline member functions; and the cheap cost is almost certainly worth the protection against the slight risk that issues of closure will crop up.

Thus, in those rare cases where the open-closed principle is not violated, the proscription of `public` and `protected` variables depends more upon style than on substance.

No Global Variables -- Ever.

The argument against global variables is similar to the argument against `pubic` member variables. No module that depends upon a global variable can be closed against any other module that might write to that variable. Any module that uses the variable in a way that the other modules don't expect, will break those other modules. It is too risky to have many modules be subject to the whim of one badly behaved one.

On the other hand, in cases where a global variable has very few dependents, or cannot be used in an inconsistent way, they do little harm. The designer must assess how much closure is sacrificed to a global and determine if the convenience offered by the global is worth the cost.

Again, there are issues of style that come into play. The alternatives to using globals are usually very inexpensive. In those cases it is bad style to use a technique that risks even a tiny amount of closure over one that does not carry such a risk. However, there are cases where the convenience of a global is significant. The global variables `cout` and `cin` are common examples. In such cases, if the open-closed principle is not violated, then the convenience may be worth the style violation.

RTTI is Dangerous.

Another very common proscription is the one against `dynamic_cast`. It is often claimed that `dynamic_cast`, or any form of run time type identification (RTTI) is intrinsically dangerous and should be avoided. The case that is often cited is similar to Listing 9 which clearly violates the open-closed principle. However Listing 10 shows a similar program that uses `dynamic_cast`, but does not violate the open-closed principle.

The difference between these two is that the first, Listing 9, *must* be changed whenever a new type of `Shape` is derived. (Not to mention that it is just downright silly). How-

Listing 9

RTTI violating the open-closed principle.

```
class Shape {};

class Square : public Shape
{
  private:
    Point  itsTopLeft;
    double itsSide;
  friend DrawSquare(Square*);
};

class Circle : public Shape
{
  private:
    Point  itsCenter;
    double itsRadius;
  friend DrawCircle(Circle*);
};

void DrawAllShapes(Set<Shape*>& ss)
{
    for (Iterator<Shape*>i(ss); i; i++)
    {
        Circle* c = dynamic_cast<Circle*>(*i);
        Square* s = dynamic_cast<Square*>(*i);
        if (c)
            DrawCircle(c);
        else if (s)
            DrawSquare(s);
    }
}
```

Listing 10

RTTI that does not violate the open-closed Principle.

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

class Square : public Shape
{
    // as expected.
};

void DrawSquaresOnly(Set<Shape*>& ss)
```

Listing 10 (Continued)

RTTI that does not violate the open-closed Principle.

```
{
    for (Iterator<Shape*>i(ss); i; i++)
    {
        Square* s = dynamic_cast<Square*>(*i);
        if (s)
            s->Draw();
    }
}
```

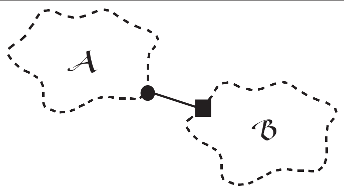
ever, nothing changes in Listing 10 when a new derivative of `Shape` is created. Thus, Listing 10 does not violate the open-closed principle.

As a general rule of thumb, if a use of RTTI does not violate the open-closed principle, it is safe.

Conclusion

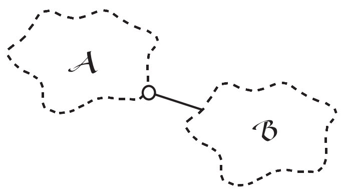
There is much more that could be said about the open-closed principle. In many ways this principle is at the heart of object oriented design. Conformance to this principle is what yeilds the greatest benefits claimed for object oriented technology; i.e. reusability and maintainability. Yet conformance to this principle is not achieved simply by using an object oriented programming language. Rather, it requires a dedication on the part of the designer to apply abstraction to those parts of the program that the designer feels are going to be subject to change.

This article is an extremely condensed version of a chapter from my new book: *The Principles and Patterns of OOD*, to be published soon by Prentice Hall. In subsequent articles we will explore many of the other principles of object oriented design. We will also study various design patterns, and their strenghts and weaknesses with regard to implementation in C++. We will study the role of Booch's class categories in C++, and their applicability as C++ namespaces. We will define what "cohesion" and "coupling" mean in an object oriented design, and we will develop metrics for measuring the quality of an object oriented design. And, after that, many other interesting topics.



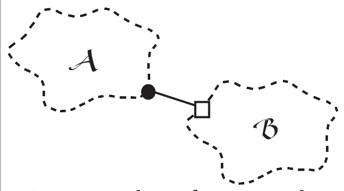
Contains - by value. This indicates that A and B have identical lifetimes. When A is destroyed, B will be destroyed too.

```
class A
{
  private:
    B itsB;
};
```



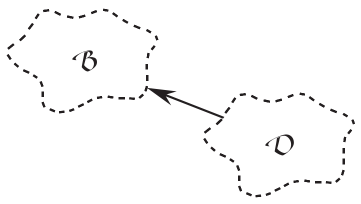
Uses. This indicates that the name of class B is used within the source code of class A.

```
class A
{
  public:
    void F(const B&);
};
```



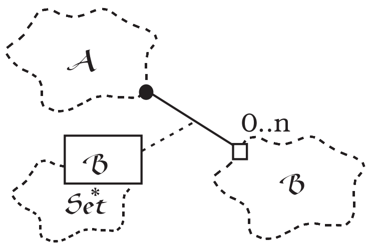
Contains - by reference. This is used to indicate that A and B have dissimilar lifetimes. B may outlive A.

```
class A
{
  private:
    B* itsB;
};
```



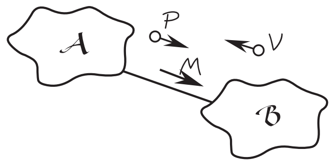
Inheritance. This indicates that B is a public base class of D.

```
class D : public B {...};
```



Using a container class. In this case a template for a "Set".

```
class A
{
  private:
    Set<B*> itsBs;
};
```

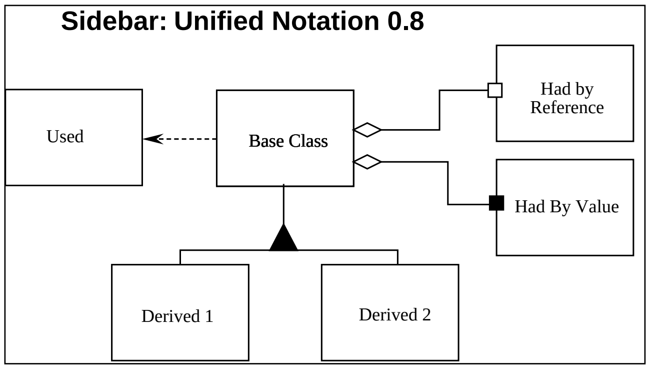


Invoking a member function.

```
void A::F(B& theB)
{
    P p;
    V v = theB.M(p);
}
```

The Liskov Substitution Principle

This is the second of my *Engineering Notebook* columns for *The C++ Report*. The articles that will appear in this column will focus on the use of C++ and OOD, and will address issues of software engineering. I will strive for articles that are pragmatic and directly useful to the software engineer in the trenches. In these articles I will make use of Booch's and Rumbaugh's new *unified* notation (Version 0.8) for documenting object oriented designs. The sidebar provides a brief lexicon of this notation.



Introduction

My last column (Jan, 96) talked about the Open-Closed principle. This principle is the foundation for building code that is maintainable and reusable. It states that well designed code can be extended without modification; that in a well designed program new features are added by adding new code, rather than by changing old, already working, code.

The primary mechanisms behind the Open-Closed principle are abstraction and polymorphism. In statically typed languages like C++, one of the key mechanisms that supports abstraction and polymorphism is inheritance. It is by using inheritance that we can create derived classes that conform to the abstract polymorphic interfaces defined by pure virtual functions in abstract 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 Open-Closed principle? These are the questions that this article will address.

The Liskov Substitution Principle

> ***FUNCTIONS THAT USE POINTERS OR REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.***

The above is a paraphrase of the Liskov Substitution Principle (LSP). Barbara Liskov first wrote it as follows nearly 8 years ago[1]:

> *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.*

The importance of this principle becomes obvious when you consider the consequences of violating it. If there is a function which does not conform to the LSP, then that function uses a pointer or reference to a base class, but must *know* about all the derivatives of that base class. Such a function violates the Open-Closed principle because it must be modified whenever a new derivative of the base class is created.

A Simple Example of a Violation of LSP

One of the most glaring violations of this principle is the use of C++ Run-Time Type Information (RTTI) to select a function based upon the type of an object. i.e.:

```
void DrawShape(const Shape& s)
{
  if (typeid(s) == typeid(Square))
    DrawSquare(static_cast<Square&>(s));
  else if (typeid(s) == typeid(Circle))
    DrawCircle(static_cast<Circle&>(s));
}
```

[Note: `static_cast` is one of the new cast operators. In this example it works exactly like a regular cast. i.e. `DrawSquare((Square&)s);`. However the new syntax has more stringent rules that make is safer to use, and is easier to locate with tools such as grep. It is therefore preferred.]

Clearly the `DrawShape` function is badly formed. 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 view the structure of this function as anathema to Object Oriented Design.

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

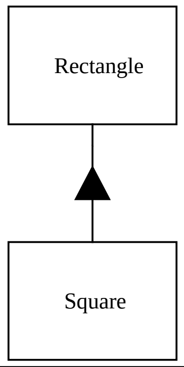
Square and Rectangle, a More Subtle Violation.

However, there are other, far more subtle, ways of violating the LSP. Consider an application which uses the `Rectangle` class as described below:

```
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:
    double itsWidth;
    double itsHeight;
};
```

Imagine that this application works well, and is installed in many sites. As is the case with all successful software, as its users' needs change, new functions are needed. Imagine that one day the users demand the ability to manipulate squares in addition to rectangles.

Figure 1.



It is often said that, in C++, inheritance is the ISA relationship. In other words, if a new kind of object can be said to fulfill the ISA relationship with an old kind of object, then the class of the new object should be derived from the class of the old object.

Clearly, a square is a rectangle for all normal intents and purposes. Since the ISA relationship holds, it is logical to model the `Square` class as being derived from `Rectangle`. (See Figure 1.)

This use of the ISA relationship is considered by many to be one of the fundamental techniques of Object Oriented Analysis. A square is a rectangle, and so the `Square` class should be derived from the `Rectangle` class. However this kind of thinking can lead to some subtle, yet significant, problems. Generally these problem are not foreseen until we actually try to code the application.

Our first clue might be the fact that a `Square` does not need both `itsHeight` and `itsWidth` member variables. Yet it will inherit them anyway. Clearly this is wasteful. Moreover, if we are going to 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 extremely significant.

However, let's assume that we are not very concerned with memory efficiency. Are there other problems? Indeed! `Square` will inherit the `SetWidth` and `SetHeight` functions. These functions are utterly inappropriate for a `Square`, since the width and height of a square are identical.". This should be a significant clue that there is a problem with the design. 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 its height, the width will change with it. Thus, the invariants 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 heigt 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`.

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. Indeed, it violates the Open-Closed principle. We might counter this with argument 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`.

Still, let's assume that we accept the argument, and fix the classes. We wind up with the following code:

```
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:
    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

At this point in time we have two classes, `Square` and `Rectangle`, that 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 model is now self consistent, and correct. However, this conclusion would be amiss. A model that is self consistent is not necessarily consistent with all its users! Consider function `g` below.

```
void g(Rectangle& r)
{
  r.SetWidth(5);
  r.SetHeight(4);
  assert(r.GetWidth() * r.GetHeight()) == 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 declares an assertion error if passed a `Square`. So here is the real problem: Was the programmer who wrote that function justified in assuming that *changing the width of a `Rectangle` leaves its height unchanged*?

Clearly, the programmer of `g` made this very reasonable assumption. Passing a `Square` to functions whose programmers made this assumption will result in problems. Therefore, there exist functions that take pointers or references to `Rectangle` objects, but cannot operate properly upon `Square` objects. These functions expose a violation of the LSP. The addition of the `Square` derivative of `Rectangle` has broken these function; and so the Open-Closed principle has been violated.

Validity is not Intrinsic

This leads us to a very important conclusion. A model, viewed in isolation, can not 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.

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

What Went Wrong? (W^3)

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 ISA relationship hold?

No! A square might be a rectangle, but a `Square` object is definitely *not* a `Rectangle` object. Why? Because the *behavior* of a `Square` object is not consistent with 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 clear that in OOD the ISA relationship pertains to *behavior*. Not intrinsic private behavior, but extrinsic public behavior; behavior that clients depend upon. For example, the author of function `g` above depended on the fact that `Rectangles` behave such that their height and width vary independently of one another. That independence of the two variables is an extrinsic public behavior that other programmers are likely to depend upon.

In order for the LSP to hold, and with it the Open-Closed principle, all derivatives must conform to the behavior that clients expect of the base classes that they use.

Design by Contract

There is a strong relationship between the LSP and the concept of Design by Contract as expounded by Bertrand Meyer[2]. Using this scheme, methods of classes declare preconditions and postconditions. The preconditions must be true in order for the method to execute. Upon completion, the method guarantees that the postcondition will be true.

2. *Object Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 1988

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

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

Now the rule for the preconditions and postconditions for derivatives, as stated by Meyer[3], is:

> *...when redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one.*

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 then 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.

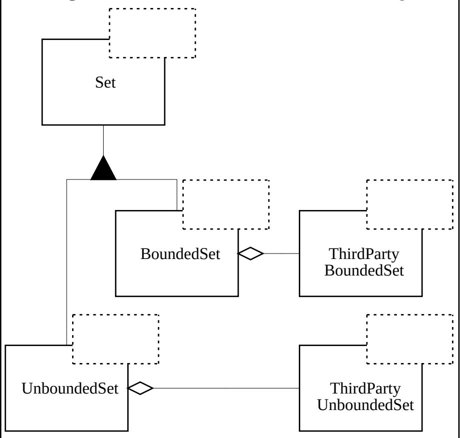
Clearly, the postcondition of `Square::SetWidth(double w)` is weaker than the postcondition of `Rectangle::SetWidth(double w)` above, since it does not conform to the base class clause "`(itsHeight == old.itsHeight)`". Thus, `Square::SetWidth(double w)` violates the contract of the base class.

Certain languages, like Eiffel, have direct support for preconditions and postconditions. You can actually declare them, and have the runtime system verify them for you. C++ does not have such a feature. Yet, even in C++ we can manually consider the preconditions and postconditions 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.

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.

Figure 2: Container Hierarchy



Motivation

I was unhappy with the interfaces of the container classes that were available through

3. *ibid*, p256

third parties. I did not want my application code to be horribly dependent upon these containers 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. (See Figure 2)

I had an abstract class called `Set` which presented pure virtual `Add`, `Delete`, and `IsMember` functions.

```
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;
};
```

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 listing.)

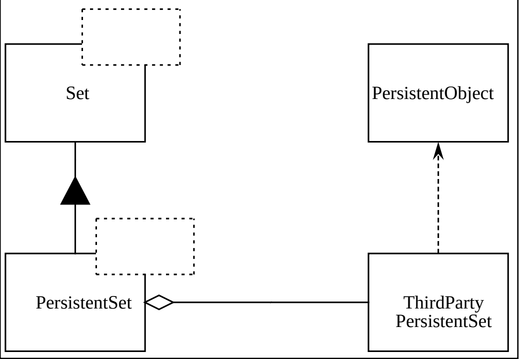
```
template <class T>
void PrintSet(const Set<T>& s)
{
   for (Iterator<T>i(s); i; i++)
     cout << (*i) << endl;
}
```

This ability to neither know nor care the type of `Set` you are operating on is a big advantage. It means that the programmer can decide which kind of `Set` is needed in each particular instance. None of the client functions will be affected by that decision. The programmer may choose a `BoundedSet` when memory is tight and speed is not critical, or the programmer may choose an `UnboundedSet` 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 which 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

Figure 3. Persistent Set



derived from the abstract base class `PersistentObject`. I created the hierarchy shown in Figure 3.

On the surface of it, this might look all right. However there is an implication that is rather ugly. When a client is adding members to the base class Set, how is that client supposed to ensure that it only adds derivatives of `PersistentObject` if the `Set` happens to be a `PersistentSet`?

Consider the code for `PersistentSet::Add:`

```
template <class T>
void PersistentSet::Add(const T& t)
{
  PersistentObject& p =
    dynamic_cast<PersistentObject&>(t); // throw bad_cast
  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` (one of the standard exception objects). None of the existing clients of the abstract base class `Set` expect 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`, will 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 the failed 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.

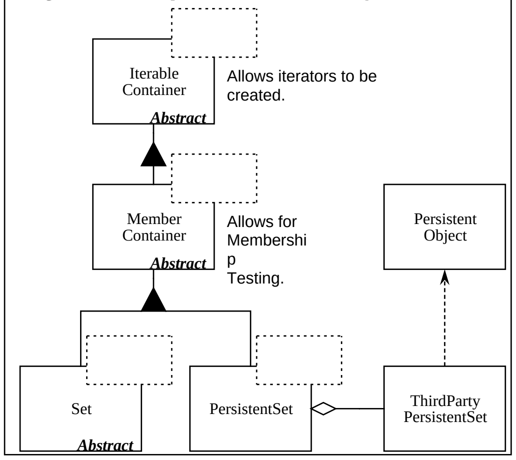
How do we solve this problem? Several years ago, I solved it by convention. Which is to say that I did not solve it in source code. Rather I instated a convention whereby `PersistentSet` and `PersistentObject` were not known to the application as a whole. They were only known to one particular module. This module was responsible for reading and writing all the containers. When a container needed to be written, its contents were copied into `PersistentObjects` 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 (non-persistent) 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 non-persistent objects to them. Moreover it broke the dependency of the rest of the application upon the whole notion of persistence.

Did this solution work? Not really. The convention was violated in several parts of the application by engineers who did not understand the necessity for it. That is the problem with conventions, they have to be continually re-sold to each engineer. If the engineer does not agree, then the convention will be violated. And one violation ruins the whole structure.

An LSP Compliant Solution

Figure 4: Separate Hierarchy



How would I solve this now? I would acknowledge that a `PersistentSet` does not have an ISA 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. Thus 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 4.) This would allow `PersistentSet` objects to be Iterated and tested for membership, etc. But would not afford the ability to add objects that were not derived from `PersistentObject` to a `PersistentSet`.

```
template <class T>
void PersistentSet::Add(const T& t)
{
 itsThirdPartyPersistentSet.Add(t);
   // This will generate a compiler error if t is
   // not derived from PersistentObject.
}
```

As the listing above shows, any attempt to add an object that is not derived from `PersistentObject` to a `PersistentSet` will result in a compilation error. (The interface of the third party persistent set expects a `PersistentObject&`).

Conclusion

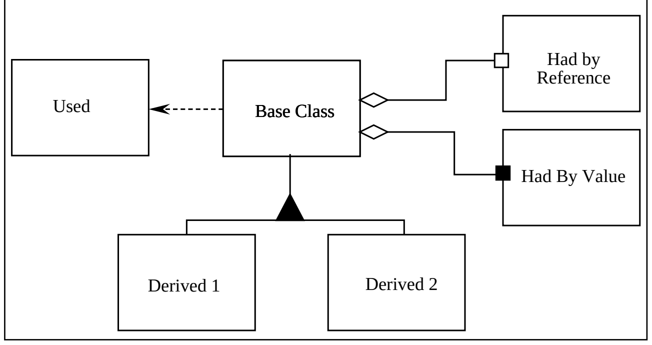
The Open-Closed principle is at the heart of many of the claims made for OOD. It is when this principle is in effect that applications are more maintainable, reusable and robust. The Liskov Substitution Principle (A.K.A Design by Contract) is an important feature of all programs that conform to the Open-Closed principle. It is only when derived types are completely substitutable for their base types that functions which use those base types can be reused with impunity, and the derived types can be changed with impunity.

This article is an extremely condensed version of a chapter from my new book: *Patterns and Advanced Principles of OOD*, to be published soon by Prentice Hall. In subsequent articles we will explore many of the other principles of object oriented design. We will also study various design patterns, and their strengths and weaknesses with regard to implementation in C++. We will study the role of Booch's class categories in C++, and their applicability as C++ namespaces. We will define what "cohesion" and "coupling" mean in an object oriented design, and we will develop metrics for measuring the quality of an object oriented design. And, after that, many other interesting topics.

The Interface Segregation Principle

This is the fourth of my *Engineering Notebook* columns for *The C++ Report*. The articles that appear in this column focus on the use of C++ and OOD, and address issues of software engineering. I strive for articles that are pragmatic and directly useful to the software engineer in the trenches. In these articles I make use of Booch's and Rumbaugh's new unified Modeling Langage (UML Version 0.8) for documenting object oriented designs. The sidebar provides a brief lexicon of this notation.

Sidebar: Unified Notation 0.8



Introduction

In my last column (May 96) I discussed the principle of Dependency Inversion (DIP). This principle states that modules that encapsulate high level policy should not depend upon modules that implement details. Rather, both kinds of modules should depend upon abstractions. To be more succinct and simplistic, abstract classes should not depend upon concrete classes; concrete classes should depend upon abstract classes. A good example of this principle is the TEMPLATE METHOD pattern from the GOF[1] book. In this pattern, a high level algorithm is encoded in an abstract base class and makes use of pure virtual functions to implement its details. Derived classes implement those detailed virtual functions. Thus, the class containing the details depend upon the class containing the abstraction.

In this article we will examine yet another structural principle: the Interface Segregation Principle (ISP). This principle deals with the disadvantages of "fat" interfaces. Classes that have "fat" interfaces are classes whose interfaces are not cohesive. In other

1. *Design Patterns*, Gamma, et. al. Addison Wesley, 1995

words, the interfaces of the class can be broken up into groups of member functions. 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 non-cohesive 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. Some languages refer to these abstract base classes as “interfaces”, “protocols” or “signatures”.

In this article we will discuss the disadvantages of “fat” or “polluted” interfacse. We will show how these interfaces get created, and how to design classes which hide them. Finally we will present a case study in which the a “fat” interface naturally occurs, and we will employ the ISP to correct it.

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 1).

Listing 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 upon 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 2.)

Listing 2

Timer

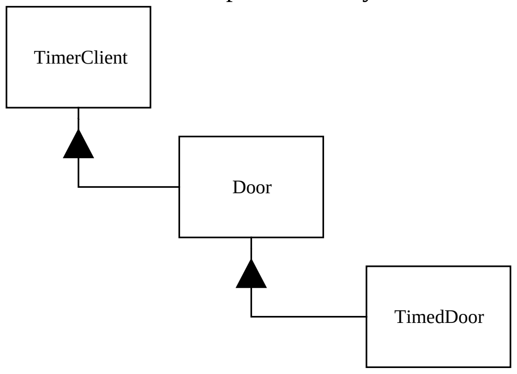
```
class Timer
{
  public:
    void Regsiter(int timeout, TimerClient* client);
};

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

When an object wishes to be informed about a timeout, it calls the Register function of the Timer. The arguments of this function are the time of the timeout, and a pointer to a TimerClient object whose TimeOut function will be called when the timeout 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 timeout? There are several alternatives. Figure 1 shows a common 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 1
TimerClient at top of hierarchy



Although this solution is common, it is not without problems. Chief among these is that the Door class now depends upon 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 nil implementations for the TimeOut method. Moreover, the applications that use those derivatives will have to #include the definition of the TimerClient class, even though it is not used.

Figure 1 shows a common syndrome of object oriented design in statically typed languates like C++. This is the syndrome of interface pollution. The interface of Door has been polluted with an interface that it does not require. It has been forced to incorporate this interface solely for the benefit of one of its subclasses. If this practice is pursued, then every time a derivative needs a new interface, that interface 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 interface is added to the base class, that interface must be implemented (or allowed to default) in derived classes. Indeed, an associated practice is to add these interfaces to the base class as *nil* virtual functions rather than pure virtual functions; specifically so that derived classes are not burdened with the need to implement them. As we learned in the second article of this column, such a practice violates the Liskov Substitution Principle (LSP), leading to maintenance and reusability problems.

Separate Clients mean Separate Interfaces.

Door and TimerClient represent interfaces that are used by complely 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, as we will see in the next section, clients exert forces upon their server interfaces.

The backwards force applied by clients upon 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. That is, sometimes it is the user that forces a change to the interface.

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

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

Listing 3

Timer with ID

```
class Timer
{
  public:
    void Regsiter(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 1 will also cause Door, and all clients of Door to be affected (i.e. at least recompiled) by this fix! Why should a bug in TimerClient have *any* affect on clients of Door derivatives that do not require timing? It is this kind of strange interdependency that chills customers and managers to the bone. When a change in one part of the program affects other completely unerlated parts of the program, the cost and repercussions of changes become unpredictable; and the risk of fallout from the change increases dramatically.

But it's just a recompile.

True. But recompiles can be very expensive for a number of reasons. First of all, they take time. When recompiles take too much time, developers begin to take shortcuts. They may hack a change in the "wrong" place, rather than engineer a change in the "right" place; because the "right" place will force a huge recompilation. Secondly, a recompilation means a new object module. In this day and age of dynamically linked libraries and incremental loaders, generating more object modules than necessary can be a significant disadvantage. The more DLLs that are affected by a change, the greater the problem of distributing and managing the change.

The Interface Segregation Principle (ISP)

CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE.

When clients are forced to depend upon interfaces that they don't use, then those clients are subject to changes to those interfaces. This results in an inadvertent coupling between all the clients. Said another way, when a client depends upon a class that contains interfaces 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 where possible.

Class Interfaces vs 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?

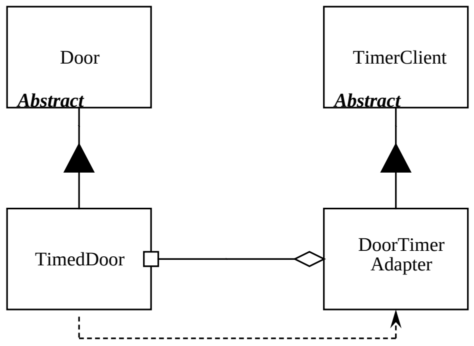
The answer to this lies in the fact that clients of an object do not need to access it through the interface of the object. Rather, they can access it through delegation, or through a base class of the object.

Separation through Delegation

We can employ the *object form* of theADAPTER[2] pattern to the TimedDoor problem. The solution is to create an adapter object that derives from TimerClient and delegates to the TimedDoor. Figure 2 shows this solution.

When the TimedDoor wants to register a timeout 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 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 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 4)

Listing 4
Object Form of Adapter Pattern

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

class DoorTimerAdapter : public TimerClient
{
  public:
    DoorTimerAdapter(TimedDoor& theDoor)
    : itsTimedDoor(theDoor)
    {}

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

2. Another GOF pattern. ibid.

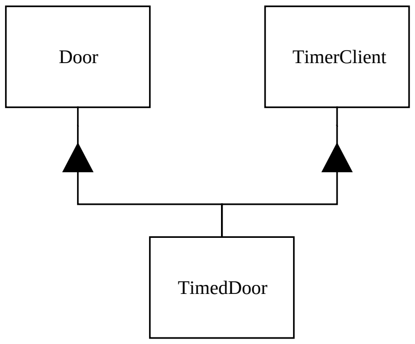
```
  private:
    TimedDoor& itsTimedDoor;
};
```

However, this solution is also somewhat inelegant. It involves the creation of a new object every time we wish to register a timeout. Moreover the delegation requires a very small, but still non-zero, 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 3 and Listing 5 show how Multiple Inheritance can be used, in the *class form* of the ADAPTER pattern, 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 depend upon the TimedDoor class. Thus, they use the same object through separate interfaces.

Figure 3
Multiply Inherited Timed Door



Listing 5
Class Form of Adapter Pattern

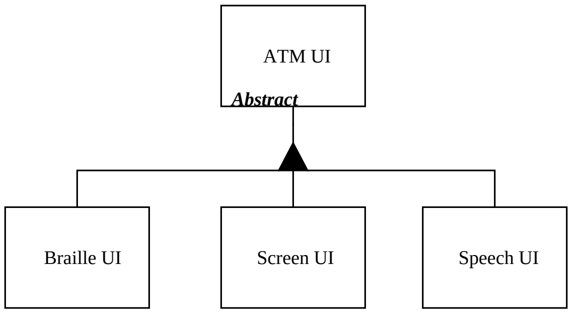
```
class TimedDoor : public Door, public TimerClient
{
  public:
    virtual void TimeOut(int timeOutId);
};
```

This solution is my normal preference. Multiple Inheritance does not frighten me. Indeed, I find it quite useful in cases such as this. The only time I would choose the solution in Figure 2 over Figure 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 language. 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 pure virtual functions for all the different messages that need to be presented by the interface.

Figure 4
ATM UI Hierarchy



Consider also that each different transaction that the ATM can perform is encasulated as a derivative of the class Transaction. Thus we might have classes such as DepositTransaction, WithdrawlTransaction, TransferTransaction, etc. Each of these objects issues message to the UI. For example, the DepositTransaction object calls the RequestDepositAmount member function of the UI class. Whereas the TransferTransaction object calls the RequestTransferAmount member function of UI. This corresponds to the diagram in Figure 5.

Notice that this is precicely the situation that the ISP tells us to avoid. Each of the transactions is using a portion of the UI that no other object uses. This creates the possibility that changes to one of the derivatives of Transaction will force coresponding change to the UI, thereby affecting all the other derivatives of Transaction, and every other class that depends upon the UI interface.

This unfortunate coupling can be avoided by segregating the UI interface into induvidual abstract base classes such as DepositUI, WithdrawUI and TransferUI. These abstract base classes can then be multiply inherited into the final UI abstract class. Figure 6 and Listing 6 show this model.

It is true that, whenever a new derivative of the Transaction class is created, a coresponding base class for the abstract UI class will be needed. Thus the UI class 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 contained.

Figure 5
ATM Transaction Hierarchy

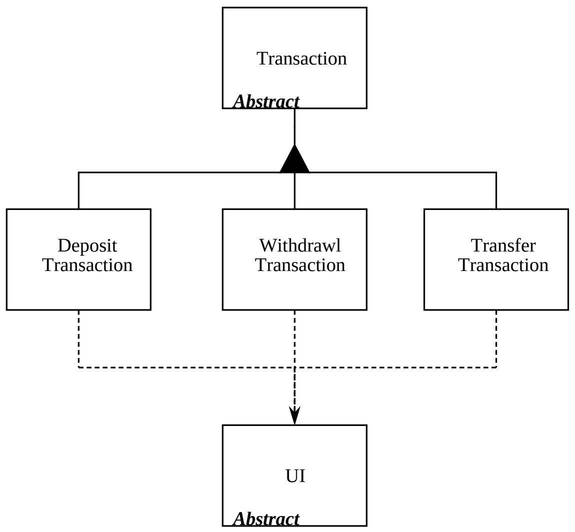
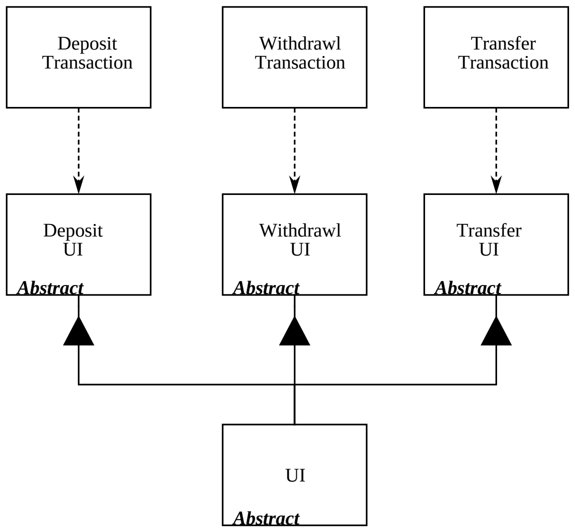


Figure 6
Segregated ATM UI Interface



Listing 6

Segregated ATM Interfaces

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

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

    virtual void Execute()
    {
      ...
      itsDepositUI.RequestDepositAmount();
      ...
    }
  private:
    DepositUI& itsDepositUI;
};

class WithdrawlUI
{
  public:
    virtual void RequestWithdrawlAmount() = 0;
};

class class WithdrawlTransation : public Transaction
{
  public:
    WithdrawlTransaction(WithdrawlUI& ui)
    : itsWithdrawlUI(ui)
    {}

    virtual void Execute()
    {
      ...
      itsWithdrawlUI.RequestWithdrawlAmount();
      ...
    }
  private:
    WithdrawlUI& itsWithdrawlUI;
};

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

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

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

class UI : public DepositUI,
         , public WithdrawlUI,
         , public TransferUI
{
  public:
    virtual void RequestDepositAmount();
    virtual void RequestWithdrawlAmount();
    virtual void RequestTransferAmount();
};
```

A careful examination of Listing 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 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 idom in Listing 7.

Listing 7

Interface Initialization Idiom

```
UI Gui; // global object;

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

This is handy, but 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 8. As we discovered when we discussed the Open Closed Principle in the January 96 issue, global variables are not always a symptom of a poor design. In this case they provide the distinct advantage of easy access. And 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 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;
WithdrawlUI& GwithdrawlUI = Lui;
TransferUI&  GtransferUI  = Lui;


// In the depositTransaction.h module


class class WithdrawlTransation : public Transaction
{
  public:

    virtual void Execute()
    {
      ...
```

```
          GwithdrawlUI.RequestWithdrawlAmount();
          ...
        }
    };
```

One might be tempted to put all the globals in Listing 8 into a single class in order to prevent pollution of the global namespace. Listing 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, withdrawUI.h, and transferUI.h. This means that any module wishing to use any of the UI interfaces transitively depends upon 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 9

Wrapping the Globals in a class

```
// in ui_globals.h

#include “depositUI.h”
#include “withdrawlUI.h”
#include “transferUI.h”

class UIGlobals
{
  public:
    static WithdrawlUI& withdrawl;
    static DepositUI&   deposit;
    static TransferUI&  transfer
};

// in ui_globals.cc

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

The Polyad vs. 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 would look like this: `g(ui, ui);` Somehow this seems perverse.

Perverse or not, the polyadic form is preferable to the monadic form. The monadic form forces ‘g’ to depend upon every interface included in UI. Thus, when WithdrawUI changed, ‘g’ and all clients of ‘g’ would have to recompile. This is more perverse than **`g(ui,ui);`**! Moreover, we cannot be sure that both arguments of ‘g’ will *always* refer to the same object! In the future, it may be that the interface objects are separated for some reason. From the point of view of function ‘g’, the fact that all interfaces are combined into a single object is information that ‘g’ does not need to know. Thus, I prefer the polyadic form for such functions.

Conclusion

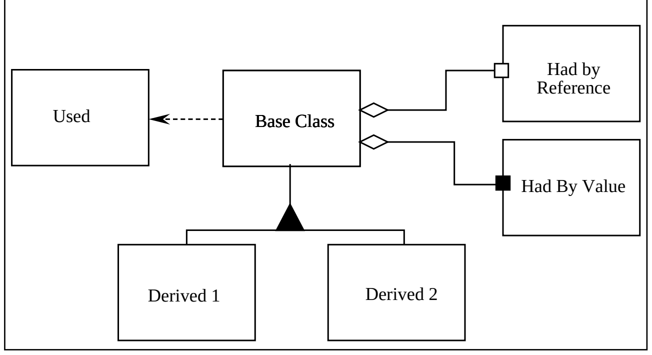
In this article we have discussed the disadvantages of “fat interfaces”; i.e. interfaces that are not specific to a single client. Fat interfaces lead to inadvertent couplings beweenclients that ought otherwise to be isolated. By making use of the ADAPTER pattern, either through delegation (object form) or multiple inheritance (class form), fat interfaces can be segregated into abstract base classes that break the unwanted coupling between clients.

This article is an extremely condensed version of a chapter from my new book: *Patterns and Advanced Principles of OOD*, to be published soon by Prentice Hall. In subsequent articles we will explore many of the other principles of object oriented design. We will also study various design patterns, and their strengths and weaknesses with regard to implementation in C++. We will study the role of Booch’s class categories in C++, and their applicability as C++ namespaces. We will define what “cohesion” and “coupling” mean in an object oriented design, and we will develop metrics for measuring the quality of an object oriented design. And after that, we will discuss many other interesting topics.

The Dependency Inversion Principle

This is the third of my *Engineering Notebook* columns for *The C++ Report*. The articles that will appear in this column will focus on the use of C++ and OOD, and will address issues of software engineering. I will strive for articles that are pragmatic and directly useful to the software engineer in the trenches. In these articles I will make use of Booch's and Rumbaugh's new *unified* notation (Version 0.8) for documenting object oriented designs. The sidebar provides a brief lexicon of this notation.

Sidebar: Unified Notation 0.8



Introduction

My last article (Mar, 96) talked about the Liskov Substitution Principle (LSP). This principle, when applied to C++, provides guidance for the use of public inheritance. It states that every function which operates upon a reference or pointer to a base class, should be able to operate upon derivatives of that base class without knowing it. This means that the virtual member functions of derived classes must expect no more than the corresponding member functions of the base class; and should promise no less. It also means that virtual member functions that are present in base classes must also be present in the derived classes; and they must do useful work. When this principle is violated, the functions that operate upon pointers or references to base classes will need to check the type of the actual object to make sure that they can operate upon it properly. This need to check the type violates the Open-Closed Principle (OCP) that we discussed last January.

In this column, we discuss the structural implications of the OCP and the LSP. The structure that results from rigorous use of these principles can be generalized into a principle all by itself. I call it "The Dependency Inversion Principle" (DIP).

What goes wrong with software?

Most of us have had the unpleasant experience of trying to deal with a piece of software that has a “bad design”. Some of us have even had the much more unpleasant experience of discovering that we were the authors of the software with the “bad design”. What is it that makes a design bad?

Most software engineers don’t set out to create “bad designs”. Yet most software eventually degrades to the point where someone will declare the design to be unsound. Why does this happen? Was the design poor to begin with, or did the design actually degrade like a piece of rotten meat? At the heart of this issue is our lack of a good working definition of “bad” design.

The Definition of a “Bad Design”

Have you ever presented a software design, that you were especially proud of, for review by a peer? Did that peer say, in a whining derisive sneer, something like: “Why’d you do it *that* way?”. Certainly this has happened to me, and I have seen it happen to many other engineers too. Clearly the disagreeing engineers are not using the same criteria for defining what “bad design” is. The most common criterion that I have seen used is the TNTWIWHDI or “That’s not the way I would have done it” criterion.

But there is one set of criteria that I think all engineers will agree with. A piece of software that fulfills its requirements and yet exhibits any or all of the following three traits has a bad design.

1. It is hard to change because every change affects too many other parts of the system. (Rigidity)
2. When you make a change, unexpected parts of the system break. (Fragility)
3. It is hard to reuse in another application because it cannot be disentangled from the current application. (Immobility)

Moreover, it would be difficult to demonstrate that a piece of software that exhibits none of those traits, i.e. it is flexible, robust, and reusable, and that also fulfills all its requirements, has a bad design. Thus, we can use these three traits as a way to unambiguously decide if a design is “good” or “bad”.

The Cause of “Bad Design”.

What is it that makes a design rigid, fragile and immobile? It is the interdependence of the modules within that design. A design is rigid if it cannot be easily changed. Such rigidity is due to the fact that a single change to heavily interdependent software begins a cascade of changes in dependent modules. When the extent of that cascade of change cannot be predicted by the designers or maintainers, the impact of the change cannot be estimated. This makes the cost of the change impossible to predict. Managers, faced with such unpredictability, become reluctant to authorize changes. Thus the design becomes officially rigid.

Fragility is the tendency of a program to break in many places when a single change is made. Often the new problems are in areas that have no conceptual relationship with the area that was changed. Such fragility greatly decreases the credibility of the design and maintenance organization. Users and managers are unable to predict the quality of their product. Simple changes to one part of the application lead to failures in other parts that appear to be completely unrelated. Fixing those problems leads to even more problems, and the maintenance process begins to resemble a dog chasing its tail.

A design is immobile when the desirable parts of the design are highly dependent upon other details that are not desired. Designers tasked with investigating the design to see if it can be reused in a different application may be impressed with how well the design would do in the new application. However if the design is highly interdependent, then those designers will also be daunted by the amount of work necessary to separate the desirable portion of the design from the other portions of the design that are undesirable. In most cases, such designs are not reused because the cost of the separation is deemed to be higher than the cost of redevelopment of the design.

Example: the “Copy” program.

A simple example may help to make this point. Consider a simple program that is charged with the task of copying characters typed on a keyboard to a printer. Assume, furthermore, that the implementation platform does not have an operating system that supports device independence. We might conceive of a structure for this program that looks like Figure 1:

Figure 1. Copy Program.

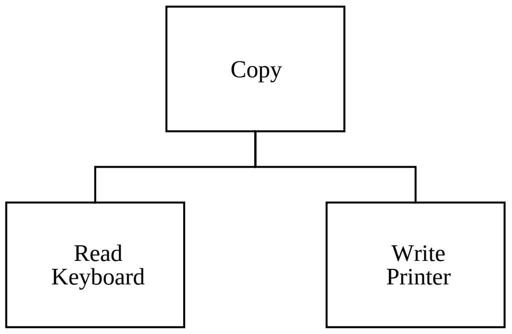


Figure 1 is a “structure chart”[1]. It shows that there are three modules, or subprograms, in the application. The “Copy” module calls the other two. One can easily imagine a loop within the “Copy” module. (See Listing 1.) The body of that loop calls the “Read Keyboard” module to fetch a character from the keyboard, it then sends that character to the “Write Printer” module which prints the character.

1. See: *The Practical Guide To Structured Systems Design*, by Meilir Page-Jones, Yourdon Press, 1988

The two low level modules are nicely reusable. They can be used in many other programs to gain access to the keyboard and the printer. This is the same kind of reusability that we gain from subroutine libraries.

Listing 1. The Copy Program

```
void Copy()
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    WritePrinter(c);
}
```

However the “Copy” module is not reusable in any context which does not involve a keyboard or a printer. This is a shame since the intelligence of the system is maintained in this module. It is the “Copy” module that encapsulates a very interesting policy that we would like to reuse.

For example, consider a new program that copies keyboard characters to a disk file. Certainly we would like to reuse the “Copy” module since it encapsulates the high level policy that we need. i.e. it knows how to copy characters from a source to a sink. Unfortunately, the “Copy” module is dependent upon the “Write Printer” module, and so cannot be reused in the new context.

Listing 2. The “Enhanced” Copy Program

```
enum OutputDevice {printer, disk};
void Copy(outputDevice dev)
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    if (dev == printer)
      WritePrinter(c);
    else
      WriteDisk(c);
}
```

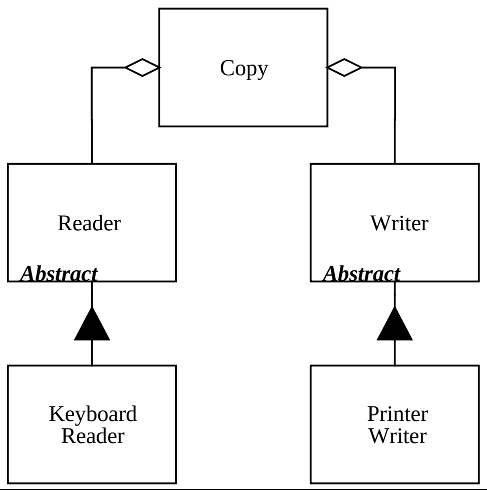
We could certainly modify the “Copy” module to give it the new desired functionality. (See Listing 2). We could add an ‘if’ statement to its policy and have it select between the “Write Printer” module and the “Write Disk” module depending upon some kind of flag. However this adds new interdependencies to the system. As time goes on, and more and more devices must participate in the copy program, the “Copy” module will be littered with if/else statements and will be dependent upon many lower level modules. It will eventually become rigid and fragile.

Dependency Inversion

One way to characterize the problem above is to notice that the module containing the high level policy, i.e. the Copy() module, is dependent upon the low level detailed modules that it controls. (i.e. WritePrinter() and ReadKeyboard()). If we could find a way to make the Copy() module independent of the details that it controls, then we could reuse it freely. We could produce other programs which used this module to copy characters from any input device to any output device. OOD gives us a mechanism for performing this *dependency inversion*.

Consider the simple class diagram in Figure 2. Here we have a “Copy” class which contains an abstract “Reader” class and an abstract “Writer” class. One can easily imagine a loop within the “Copy” class that gets characters from its “Reader” and sends them to its “Writer” (See Listing 3). Yet this “Copy” class does not depend upon the “Keyboard Reader” nor the “Printer Writer” at all. Thus the dependencies have been *inverted*; the “Copy” class depends upon abstractions, and the detailed readers and writers depend upon the same abstractions.

Figure 2: The OO Copy Program



Now we can reuse the “Copy” class, independently of the “Keyboard Reader” and the “Printer Writer”. We can invent new kinds of “Reader” and “Writer” derivatives that we can supply to the “Copy” class. Moreover, no matter how many kinds of “Readers” and “Writers” are created, “Copy” will depend upon none of them. There will be no interdependencies to make the program fragile or rigid. And Copy() itself can be used in many different detailed contexts. It is mobile.

Listing 3: The OO Copy Program

```
class Reader
{
  public:
    virtual int Read() = 0;
};

class Writer
{
  public:
    virtual void Write(char) = 0;
};

void Copy(Reader& r, Writer& w)
{
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```

Device Independence

By now, some of you are probably saying to yourselves that you could get the same benefits by writing Copy() in C, using the device independence inherent to `stdio.h`; i.e. `getchar` and `putchar` (See Listing 4). If you consider Listings 3 and 4 carefully, you will realize that the two are logically equivalent. The abstract classes in Figure 3 have been replaced by a different kind of abstraction in Listing 4. It is true that Listing 4 does not use classes and pure virtual functions, yet it still uses abstraction and polymorphism to achieve its ends. Moreover, it still uses dependency inversion! The Copy program in Listing 4 does not depend upon any of the details it controls. Rather it depends upon the abstract facilities declared in `stdio.h`. Moreover, the IO drivers that are eventually invoked also depend upon the abstractions declared in stdio.h. Thus the device independence within the `stdio.h` library is another example of dependency inversion.

Listing 4: Copy using stdio.h

```
#include <stdio.h>
void Copy()
{
  int c;
  while((c = getchar()) != EOF)
    putchar(c);
}
```

Now that we have seen a few examples, we can state the general form of the DIP.

The Dependency Inversion Principle

A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.

B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.

One might question why I use the word “inversion”. Frankly, 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 upon low level modules, and in which abstractions depend upon details. 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. Figure 1 is a good example of such a hierarchy. Thus, 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 upon low level modules. It is the high level modules that contain the important policy decisions and business models of an application. It is these models that contain the identity of the application. Yet, when these modules depend upon the lower level modules, then changes to the lower level modules can have direct effects upon them; and can force them to change.

This predicament is absurd! It is the high level modules that ought to be forcing the low level modules to change. It is the high level modules that should take precedence over the lower level modules. High level modules simply should not depend upon low level modules in any way.

Moreover, it is high level 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 upon 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 is the principle that is at the very heart of framework design.

Layering

According to Booch[2], "...all well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services though a well-defined and controlled interface." A naive interpretation of this statement might lead a designer to produce a structure similar to Figure 3. In this diagram the high level policy class uses a lower level Mechanism; which in turn uses a detailed level utility class. 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 upon something that depends upon the Utility Layer, thus the Policy Layer transitively depends upon the Utility Layer. This is very unfortunate.

Figure 3: Simple Layers

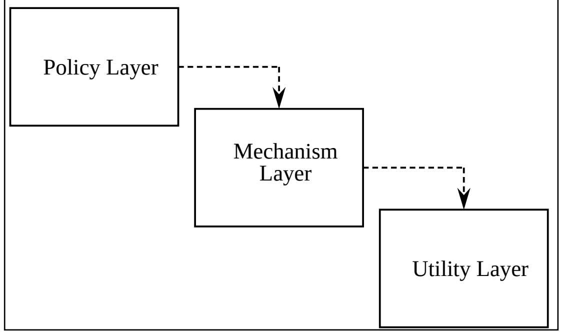


Figure 4: Abstract Layers

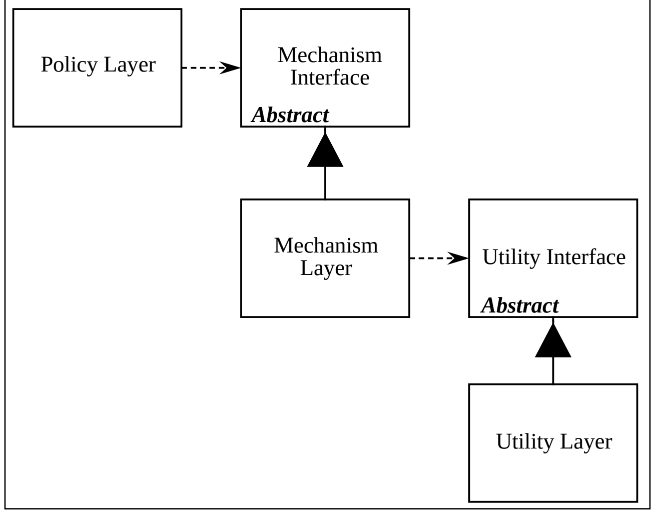


Figure 4 shows a more appropriate model. Each of the lower level layers are represented by an abstract class. The actual layers are then derived from these abstract classes. Each of the higher level classes uses the next lowest layer through the abstract interface. Thus, none of the layers depends upon any of the other layers. Instead, the layers depend upon abstract classes. Not only is the transitive dependency of Policy Layer upon Utility Layer broken, but even the direct dependency of Policy Layer upon Mechanism Layer is broken.

2. *Object Solutions*, Grady Booch, Addison Wesley, 1996, p54

Using this model, Policy Layer is unaffected by any changes to Mechanism Layer or Utility Layer. Moreover, Policy Layer can be reused in any context that defines lower level modules that conform to the Mechanism Layer interface. Thus, by inverting the dependencies, we have created a structure which is simultaneously more flexible, durable, and mobile.

Separating Interface from Implementation in C++

One might complain that the structure in Figure 3 does not exhibit the dependency, and transitive dependency problems that I claimed. After all, Policy Layer depends only upon the *interface* of Mechanism Layer. Why would a change to the implementation of Mechanism Layer have any affect at all upon Policy Layer?

In some object oriented language, this would be true. In such languages, interface is separated from implementation automatically. In C++ however, there is no separation between interface and implementation. Rather, in C++, the separation is between the definition of the class and the definition of its member functions.

In C++ we generally separate a class into two modules: a `.h` module and a `.cc` module. The `.h` module contains the definition of the class, and the `.cc` module contains the definition of that class's member functions. The definition of a class, in the `.h` module, contains declarations of all the member functions and member variables of the class. This information goes beyond simple interface. All the utility functions and private variables needed by the class are also declared in the `.h` module. These utilities and private variables are part of the implementation of the class, yet they appear in the module that all users of the class must depend upon. Thus, in C++, implementation is not automatically separated from interface.

This lack of separation between interface and implementation in C++ can be dealt with by using purely abstract classes. A purely abstract class is a class that contains nothing but pure virtual functions. Such a class is pure interface; and its `.h` module contains no implementation. Figure 4 shows such a structure. The abstract classes in Figure 4 are meant to be purely abstract so that each of the layers depends only upon the *interface* of the subsequent layer.

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.

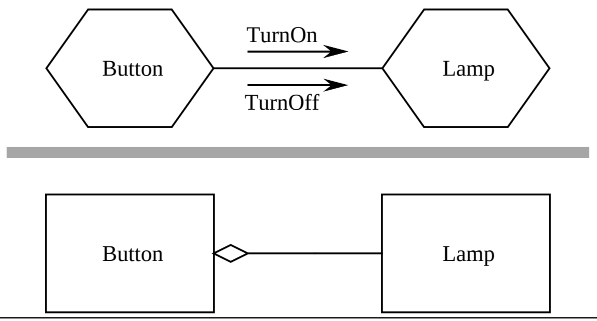
The Button object senses the external environment. It can determine whether or not a user has "pressed" it. It doesn't matter what the sensing mechanism is. It could be a button icon on a GUI, a physical button being pressed by a human finger, or even a motion detector in a home security system. The Button object detects that a user has either activated or deactivated it. The lamp object affects the external environment. Upon receiving a TurnOn message, it illuminates a light of some kind. Upon 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 5 shows a naive model. The Button object simply sends the TurnOn and TurnOff message to the Lamp. To facilitate this, the Button class uses a “contains” relationship to hold an instance of the Lamp class.

Listing 5 shows the C++ code that results from this model. Note that the Button class depends directly upon the Lamp class. In fact, the `button.cc` module `#includes` the `lamp.h` module. This dependency implies that the button class must change, or at very least be recompiled, whenever the Lamp class changes. Moreover, it will not be possible to reuse the Button class to control a Motor object.

Figure 5, and Listing 5 violate the dependency inversion principle. The high level policy of the application has not been separated from the low level modules; the abstractions have not been separated from the details. Without such a separation, the high level policy automatically depends upon the low level modules, and the abstractions automatically depend upon the details.

Figure 5: Naive Button/Lamp Model



Listing 5: Naive Button/Lamp Code

```
--------------lamp.h----------------
class Lamp
{
  public:
    void TurnOn();
    void TurnOff();
};
-------------button.h---------------
class Lamp;
class Button
{
  public:
    Button(Lamp& l) : itsLamp(&l) {}
    void Detect();
  private:
    Lamp* itsLamp;
};
-------------button.cc--------------
#include “button.h”
#include “lamp.h”

void Button::Detect()
{
  bool buttonOn = GetPhysicalState();
  if (buttonOn)
    itsLamp->TurnOn();
  else
    itsLamp->TurnOff();
}
```

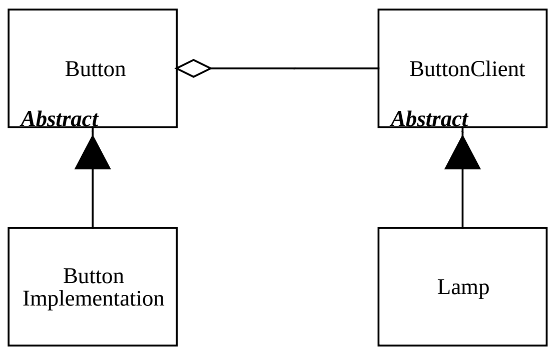
Finding the Underlying Abstraction

What is the high level policy? It is the abstractions that underlie the application, the truths that do not vary when the details are changed. 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.

To conform to the principle of dependency inversion, we must isolate this abstraction from the details of the problem. Then we must direct the dependencies of the design such that the details depend upon the abstractions. Figure 6 shows such a design.

In Figure 6, we have isolated the abstraction of the Button class, from its detailed implementation. Listing 6 shows the corresponding code. Note that the high level policy is entirely captured within the

Figure 6: Inverted Button Model



Listing 6: Inverted Button Model

```
----------byttonClient.h---------
class ButtonClient
{
  public:
    virtual void TurnOn() = 0;
    virtual void TurnOff() = 0;
};
-----------button.h---------------
class ButtonClient;
class Button
{
  public:
    Button(ButtonClient&);
    void Detect();
    virtual bool GetState() = 0;
  private:
    ButtonClient* itsClient;
};
---------button.cc----------------
#include button.h
#include buttonClient.h

Button::Button(ButtonClient& bc)
: itsClient(&bc) {}
```

```
void Button::Detect()
{
  bool buttonOn = GetState();
  if (buttonOn)
    itsClient->TurnOn();
  else
    itsClient->TurnOff();
}
-----------lamp.h----------------
class Lamp : public ButtonClient
{
  public:
    virtual void TurnOn();
    virtual void TurnOff();
};
---------buttonImp.h-------------
class ButtonImplementation
: public Button
{
  public:
    ButtonImplementaton(
      ButtonClient&);
    virtual bool GetState();
};
```

abstract button class[3]. The Button class knows nothing of the physical mechanism for detecting the user's gestures; and it knows nothing at all about the lamp. Those details are isolated within the concrete derivatives: ButtonImplementation and Lamp.

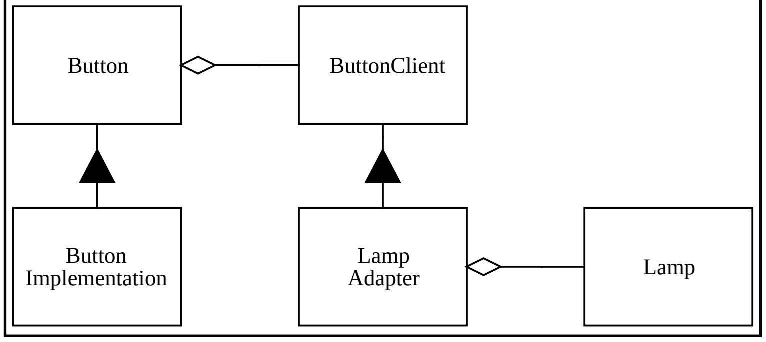
The high level policy in Listing 6 is reusable with any kind of button, and with any kind of device that needs to be controlled. Moreover, it is not affected by changes to the low level mechanisms. Thus it is robust in the presence of change, flexible, and reusable.

Extending the Abstraction Further

Once could make a legitimate complaint about the design in Figure/Listing 6. The device controlled by the button must be derived from ButtonClient. What if the Lamp class comes from a third party library, and we cannot modify the source code.

Figure 7 demonstrates how the Adapter pattern can be used to connect a third party Lamp object to the model. The LampAdapter class simply translates the TurnOn and Turn-Off message inherited from ButtonClient, into whatever messages the Lamp class needs to see.

Figure 7: Lamp Adapter



Conclusion

The principle of dependency inversion is at the root of 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

3. Aficionados of Patterns will recognize the use of the Template Method pattern in the Button Hierarchy. The member function: Button::Detect() is the template that makes use of the pure virtual function: Button::GetState(). See: *Design Patterns*, Gamma, et. al., Addison Wesley, 1995

change. And, since the abstractions and details are all isolated from each other, the code is much easier to maintain.

This article is an extremely condensed version of a chapter from my new book: *Patterns and Advanced Principles of OOD*, to be published soon by Prentice Hall. In subsequent articles we will explore many of the other principles of object oriented design. We will also study various design patterns, and their strengths and weaknesses with regard to implementation in C++. We will study the role of Booch's class categories in C++, and their applicability as C++ namespaces. We will define what “cohesion” and “coupling” mean in an object oriented design, and we will develop metrics for measuring the quality of an object oriented design. And after that, we will discuss many other interesting topics.