From Agile Principles, Patterns, and Practices in C#

The Open/Closed Principle (OCP)

Diagram

Description automatically generated

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

As Ivar Jacobson has 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.” How can we create designs that are stable in the face of change and that will last longer than the first version? The Open/Closed Principle.

**The Open/Closed Principle (OCP)**

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

When a single change to a program results in a cascade of changes to dependent modules, the design smells of rigidity. OCP advises us to refactor the system so that further changes of that kind will not cause more modifications. If OCP is applied well, further changes of that kind are achieved by adding new code, not by changing old code that already works.

Description of OCP

Modules that conform to OCP have two **primary attributes**.

1. **They are open for extension**. This means that the behavior of the module can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does.
2. **They are closed for modification**. Extending the behavior of a module does not result in changes to the source, or binary, code of the module. The binary executable version of the module—whether in a linkable library, a DLL, or a .EXE file—remains untouched.

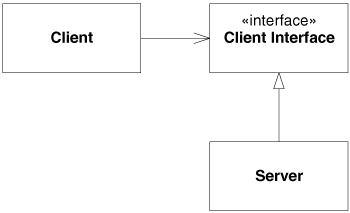
The normal way to extend the behavior of a module is to make changes to the source code of that module. A module that cannot be changed is normally thought to have a **fixed behavior**.

We do this through **abstraction**. In C# or any other object-oriented programming language (OOPL), 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 are 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 on an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction.



This shows a simple design that does not conform to OCP. Both the Client and Server classes are concrete. The Client class uses the Server class. If we want for a Client object to use a different server object, the Client class must be changed to name the new server class.

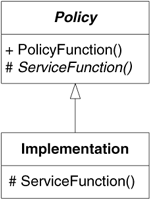


This shows the corresponding design that conforms to the OCP by using the STRATEGY pattern. In this case, the ClientInterface class is abstract with abstract member functions. The Client class uses this abstraction. However, objects of the Client class will be using objects of the derivative Server class. If we want Client objects to use a different server class, a new derivative of the ClientInterface class can be created. The Client class can remain unchanged.

The Client has some work that it needs to get done and can describe that work in terms of the abstract interface presented by ClientInterface. Subtypes of Client-Interface can implement that interface in any manner they choose. Thus, the behavior specified in Client can be extended and modified by creating new subtypes of ClientInterface.

The reason for why the interface is named ClientInterface and not AbstractServer is because abstract classes are more closely associated to their clients than to the classes that implement them.

An alternate structure is the Template Method pattern.



The Policy class has a set of concrete public functions that implement a policy, similar to the functions of the Client. As before, those policy functions describe some work that needs to be done in terms of some abstract interfaces. However, in this case, the abstract interfaces are part of the Policy class itself. In C#, they would be abstract methods. Those functions are implemented in the subtypes of Policy. Thus, the behaviors specified within Policy can be **extended or modified** by creating new derivatives of the Policy class.

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

**The Shape Application**

The Shape example has been shown in many books on object-oriented design. This infamous example is normally used to show how **polymorphism** works. However, this time, we will use it to elucidate OCP.

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

Violating OCP

In C, using procedural techniques that do not conform to OCP, we might solve this problem as shown below. 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.

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

--circle.h---------------------------------------  
struct Circle  
{  
  ShapeType itsType;  
  double itsRadius;  
  Point itsCenter;  
};  
  
void DrawCircle(struct Circle\*);  
  
--square.h---------------------------------------  
struct Square  
{  
  ShapeType itsType;  
  double itsSide;  
  Point itsTopLeft;  
};  
  
void DrawSquare(struct Square\*);

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

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

Because it cannot be closed against new kinds of shapes, the function DrawAllShapes does not conform to OCP. If I wanted to extend this function to be able to draw a list of shapes that included triangles, I would have to modify the function. In fact, I would have to modify the function for any new type of shape that I needed to draw.

Of course, this program is only a simple example. In real life, the switch statement in the DrawAllShapes function would be repeated over and over again in various functions all through the application, each one doing something a little different. There might be one each for dragging shapes, stretching shapes, moving shapes, deleting shapes, and so on. 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 to “simplify” the local decision making. In some pathological situations, functions may do precisely the same things to Squares that they do to Circles. Such functions would not even have the switch/ case statements or if/else chains. Thus, the problem of finding and understanding all the places where the new shape needs to be added can be nontrivial. Also, consider the kinds of changes that would have to be made. We’d have to add a new member to the ShapeType enum. Since all the different shapes depend on the declaration of this enum, we’d have to recompile them all.3 And we’d also have to recompile all the modules that depend on Shape.

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

In summary, the solution is rigid because the addition of Triangle causes Shape, Square, Circle, and DrawAllShapes to be recompiled and redeployed. The solution is fragile because there will be many other switch/case or if/else statements that are both difficult to find and difficult to decipher. The solution is immobile because anyone attempting to reuse DrawAllShapes in another program is required to bring along Square and Circle, even if that new program does not need them. In short, the solution exhibits many of the smells of bad design.

**Conforming to OCP**

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

public interface Shape

{

  void Draw();

}

public class Square : Shape

{

  public void Draw()

  {

    //draw a square

  }

}

public class Circle : Shape

{

  public void Draw()

  {

    //draw a circle

  }

}

  public void DrawAllShapes(IList shapes)

  {

    foreach(Shape shape in shapes)

      shape.Draw();

  }

Note that if we want to extend the behavior of the DrawAllShapes function 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 OCP. Its behavior can be extended without modifying it. Indeed, adding a Triangle class has absolutely no effect on any of the modules shown here. Clearly, some part of the system must change in order to deal with the Triangle class, but all the code shown here is immune to the change.

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

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

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

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

But consider what would happen to the DrawAllShapes function if we decided that all Circles should be drawn before any Squares. The DrawAllShapes function is not closed against a change, like this. To implement that change, we’ll have to go into DrawAllShapes and scan the list first for Circles and then again for Squares.

**Anticipation and “Natural” Structure**

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

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

We don’t want to load the design with lots of unnecessary abstraction. Rather, we want to wait until we need the abstraction and then put them in.

**Fool me once**

When a change occurs, we implement the abstractions that protect us from future changes of that kind. In short, we take the first bullet and then make sure that we are protected from any more bullets coming from that particular gun.

**Stimulating change**

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

• We write tests first. Testing is one kind of usage of the system. By writing tests first, we force the system to be testable. Therefore, changes in testability will not surprise us later. We will have built the abstractions that make the system testable. We are likely to find that many of these abstractions will protect us from other kinds of changes later.

• We use very short development cycles: days instead of weeks.

• We develop features before infrastructure and frequently show those features to stakeholders.

• We develop the most important features first.

• We release the software early and often. We get it in front of our customers and users as quickly and as often as possible.

**Using Abstraction to Gain Explicit Closure**

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

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

An ordering policy implies that, given any two objects, it is possible to discover which ought to be drawn first. C# provides such an abstraction. IComparable is an interface with one method, CompareTo. This method takes an object as a parameter and returns -1 if the receiving object is less than the parameter, 0 if they’re equal, and 1 if the receiving object is greater than the parameter.

public interface Shape : IComparable

{

  void Draw();

}

This shows what the Shape class might look like when it extends the IComparable interface.

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. C# code:

public void DrawAllShapes(ArrayList shapes)

{

  shapes.Sort();

  foreach(Shape shape in shapes)

    shape.Draw();

}

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

public class Circle : Shape

{

  public int CompareTo(object o)

  {

    if(o is Square)  return -1;

    else return 0;

  }

}

It should be very clear that this function, and all its siblings in the other derivatives of Shape, do not conform to OCP. There is no way to close them against new derivatives of Shape. Every time a new derivative of Shape is created, all the CompareTo() functions will need to be changed. Of course, this doesn’t matter if no new derivatives of Shape are ever created. On the other hand, if they are created frequently, this design would cause a significant amount of thrashing. Again, we’d take the first bullet.

**Using a Data-Driven Approach to Achieve Closure**

If we must close the derivatives of Shape from knowledge of one another, we can use a table-driven approach.

/// <summary>

/// This comparer will search the priorities

/// hashtable for a shape's type. The priorities

/// table defines the odering of shapes. Shapes

/// that are not found precede shapes that are found.

/// </summary>

public class ShapeComparer : IComparer

{

  private static Hashtable priorities = new Hashtable();

  static ShapeComparer()

  {

    priorities.Add(typeof(Circle), 1);

    priorities.Add(typeof(Square), 2);

  }

  private int PriorityFor(Type type)

  {

    if(priorities.Contains(type))

      return (int)priorities[type];

    else

      return 0;

  }

  public int Compare(object o1, object o2)

  {

    int priority1 = PriorityFor(o1.GetType());

    int priority2 = PriorityFor(o2.GetType());

    return priority1.CompareTo(priority2);

  }

}

  public void DrawAllShapes(ArrayList shapes)

  {

    shapes.Sort(new ShapeComparer());

    foreach(Shape shape in shapes)

      shape.Draw();

  }

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

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

**Conclusion**

In many ways, the Open/Closed Principle is at the heart of object-oriented design. Conformance to this principle is what yields the greatest benefits claimed for object-oriented technology: flexibility, reusability, and maintainability. Yet conformance to this principle is not achieved simply by using an object-oriented programming language. Nor is it a good idea to apply rampant abstraction to every part of the application. Rather, it requires a dedication on the part of the developers to apply abstraction only to those parts of the program that exhibit frequent change. Resisting premature abstraction is as important as abstraction itself.

**Quiz:**

1. **What is the OCP?**

**Open for extension, closed for modification**

Open for modification, closed for abstraction

Open for abstraction, closed for modification

Open for modification, closed for extension

1. **If a program conforms to OCP, then a good way to change it is by changing existing code rather than adding new code.**

True

**False**

1. I completed all of the pre-class readings before completing this quiz.

**True**

False