The Payroll Case Study: Finding the Underlying Abstractions.

```
Put in the sidebar from my first column (July/Aug 94)
```

Introduction

This is the third in my series of columns which is dedicated to exploring the attributes of “good” object oriented design. In my last column we continued the object oriented analysis and design of a simple payroll problem. The specifications for that problem are presented in a side bar. In that column, we explored the notion that many, even most, of the classes in an object oriented design represent contrivances which serve the application rather than objects that describe real world entities from the problem domain.

In this column we will continue to work on the design of the payroll problem. Our goal will be to learn about one of the most fundamental of all object oriented design principles -- abstraction. According to Hoare: “In the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction.”[1] But is is not only the *understanding* of complex phenomina which benefits from abstraction; it is also our ability to organize that complexity in ways that decreases the interdependence of one part of a software application upon another. It is this organizational ability that we will study in this article.

This article, and several others which will follow in this series, come from the third chapter of my book: “Designing Object Oriented C++ Applications using the Booch Method”; which will be published by Prentice-Hall later this year.

I will be using Booch’s notation to document analysis and design decsions. For those of you who are unfamiliar with this notation, a lexicon of its major components is provided in a sidebar.

The Open-Closed Principle

Six years ago, Bertrand Meyer wrote *Object Oriented Software Construction*.[2]. On page 23 there is a section entitled “THE OPEN-CLOSED PRINCIPLE”. To paraphrase: this principle says that a software module should be *open* for extension, but *closed* for modification. A module that is open for extension is a module whose behavior can be altered to suit new requirements. A module that is closed for modification is a module whose source code is frozen and cannot be changed.

The benefit of such a module is clear. Its behavior can be manipulated at will, and yet its source code never needs to be altered. Such a module is extremely flexible and never needs maintenance. Yet how can such a thing be? How is it possible to alter the behavior of a module without making source code changes to it? The answer is: abstraction.

Consider the example of a set of objects that can be drawn in a GUI. Such objects might be described by the following class:

[1] *Structured Programming*, Dahl, Dijkstra and Hoare, Academic Press, 1972, Pg. 83.

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

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

`DrawnObject` is an abstract class which provides a pure virtual `Draw` interface. An unlimited number of different kinds of classes can be derived from `DrawnObject`. However, they all will describe objects which can respond to the `Draw` interface.

Now consider the following function for drawing a set of `DrawnObject` instances.

```
void DrawObjects(const Set<DrawnObject*>& s)
{
    for (Iterator<DrawnObject*> i(s); i; i++)
        (*i)->Draw();
}
```

Clearly this function will draw all the objects contained by the set, regardless of what kind of derivative of `DrawnObject` they may be. In fact, this module will continue to work, unchanged, even though more and more derivatives of `DrawnObject` are created. This module is open for extension since I can extend its behavior to handle any new kind of `DrawnObject` that I like. Yet it is closed for modification because I do not have to modify the source code of the module in order to achieve this extension of behavior. Thus, this module conforms to the Open-Closed principle.

So what? Why is this important? The more modules that we can close in this fashion, the more we approach the kind of system in which changes are made by adding new code rather than modifying existing *working* code. Moreover, if modules that control high level policies can be closed, they they can be reused in many different detailed contexts without source code modifications. Thus, conformance to the Open-Closed principle is one of the ways in which we create applications that are “easy” to maintain and contain reusable modules.

How did we achieve this conformance to the Open-Closed principle? By employing abstraction. DrawnObject is an abstract class. It describes the abstraction that there are certain kinds of objects that can be told to `Draw`. We don’t necessarily know what those objects are, or what they do when they are told to `Draw`, we only know that we can call the `Draw()` member functions of those objects.

Abstraction allows us to specify interfaces, like `Draw()`, that do not depend upon their implementations. For example, `DrawnObject::Draw()` does not depend upon `Circle::Draw()`. This means that algorithms, like the `DrawObjects()` function, that are specified in terms of those interfaces, do not depend upon any specific implementations of those interfaces. For example, `DrawObjects()` does not depend upon the implementation of `Draw()` in any derivative of `DrawnObject`. Thus, when new implementations of these interfaces are added, or old implementations are changed, the algorithm can remain closed.

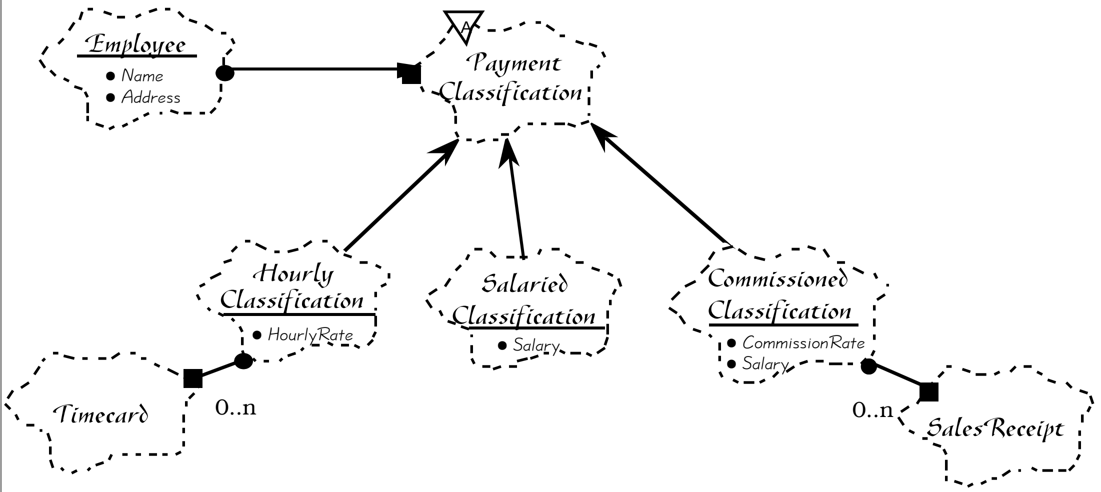
Thus it is abstraction that allows the open-closed principle to work. And therefore, the more abstractions we find in our applications, the more modules we can close. The more closed modules there are, the less code there is to change, and the more code there is to reuse.

Fiding the "Underlying Abstractions"

To be effective at employing the open/closed principle, we must hunt for abstractions. We must find the abstractions that "underlie" the application. Often these abstractions are not stated or even alluded to by the requirements of the application, or even the use cases. Requirements and use cases are too steeped in details to express the generalities of the underlying abstractions.

What are the underlying abstractions of the Payroll application? Lets look again at the requirements. We see statements like this: "Some employees work by the hour..." and "Some employees are paid a flat salary..." and again "Some [...] employees are paid a commission." This hints at a generalization which is: "Employees are all paid, but they are paid by different schemes". The abstraction here is that "Employees are all paid." Our model of the PaymentClassification in Figure 1 expresses this abstraction nicely. Thus, this abstraction was already found by the use case analysis.

Figure 1
Payment Classification



The "Schedule" abstraction

Looking for other abstractions we find: "They are paid every Friday." and "They are paid on the last working day of the month." and again "They are paid every other Friday." This leads us to another generality: "All employees are paid according to some schedule." The abstraction here is the notion of the *Schedule*. It should be possible to ask an Employee object whether a certain date is its payday. The use cases barely metion this. The requirements associate an employee's schedule with his payment classification. Specifically, hourly employees are paid weekly; salaried employees are paid monthly and employees receiving commissions are paid biweekly. However, is there any reason to believe that this association is important? Might not the policy change one day so that employees could select a particular schedule, or so that employees belonging to different departments or different divisions may have different schedules? Might not schedule policy change independent of payment policy? Certainly this must be so.

If, as the requirements imply, we delegated the issue of schedule to the PaymentClassification class, then our class could not be closed against issues of change in schedule. Each time we changed the schedule, we would have to open up the PaymentClassification. Thus, payment schedule and payment policy would be coupled together in the same module.

Such an association between schedule and payment policy could lead to bugs in which a change to a particular payment policy caused incorrect scheduling of certain employees. Bugs like this may make sense to programmers, but they strike fear in the hearts of managers and users. They fear, and rightly so, that if schedules can be broken by a change to payment policy, then any change made anywhere might cause problems in *any* other unrelated part of the system. This means that the effects of a change cannot be predicted. When effects cannot be predicted, confidence is lost and the program assumes the status of "dangerous and instable" in the minds of its managers and users.

Despite the essential nature of the schedule abstraction, neither our noun list, nor our use-case analysis gave us any clues about its existence. To spot it required careful consideration of the requirements and an insight into the wiles of the user community. Over-reliance upon tools and procedures, and under-reliance upon intelligence and experience are recipes for disaster.

Figures 2 and 3 show the static and dynamic models for the schedule abstraction. The *Employee* class contains the abstract *Schedule* class. There are three varieties of *Schedule* which corespond to the three known schedules by which employees can be paid.

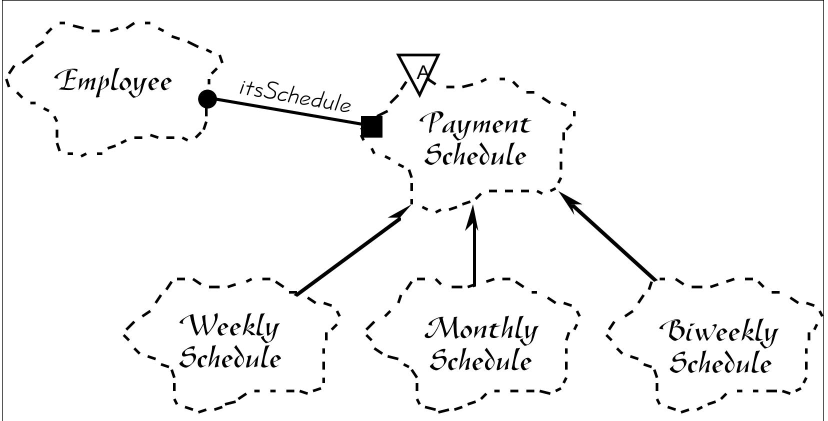


Figure 2
Static model of Schedule abstraction.

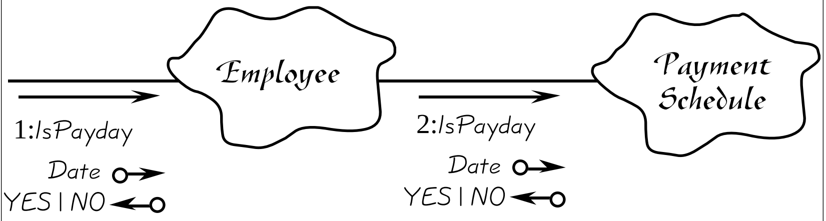


Figure 3
Dynamic model of Schedule abstraction.

Payment Methods

Another generalization that we can make from the requirements is: "All employees receive their pay via some method." The abstraction is the *PaymentMethod* class. Interestingly enough, this abstraction (See Figure 4) was mentioned in our noun list, and is already expressed in the previous articles.

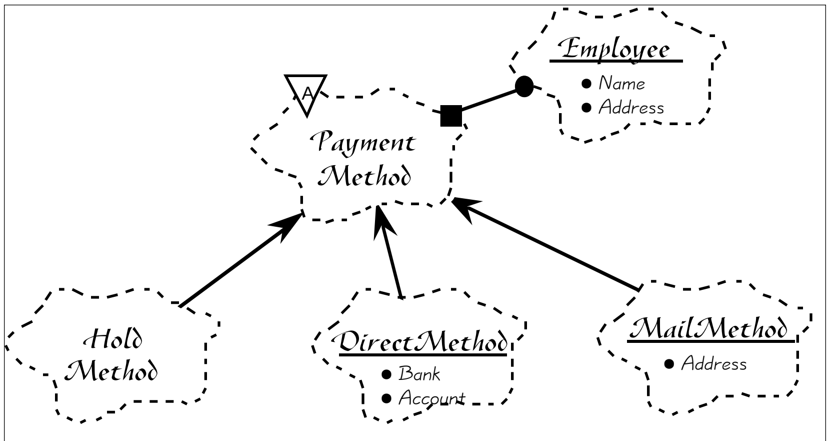


Figure 4
Static model of Payment Method abstraction.

Affiliations

The requirements imply that employees may have affiliations with a union. However, why should the union be the only organization that has a claim to some of the employee's pay? Might the employee not want to make automatic contributions to certain charities; or have his dues to professional associations be paid automatically? Thus the generalization becomes: "The employee may be affiliated with may organizations that should be automatically paid from the employee's paycheck."

In a previous article, I had described this abstraction as in Figure 5. However, this did not show the *Employee* containing more than one *Affiliation*. It also shows the presence of a *NoAffiliation* class. This model does not quite fit the abstraction we need. Figure 6 and 7 show the static and dynamic models which better represent the *Affiliation* abstraction.

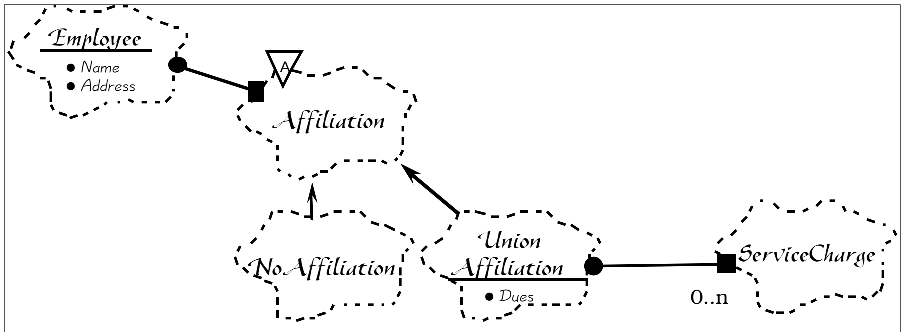


Figure 5
Previous static structure of Affiliation abstraction.

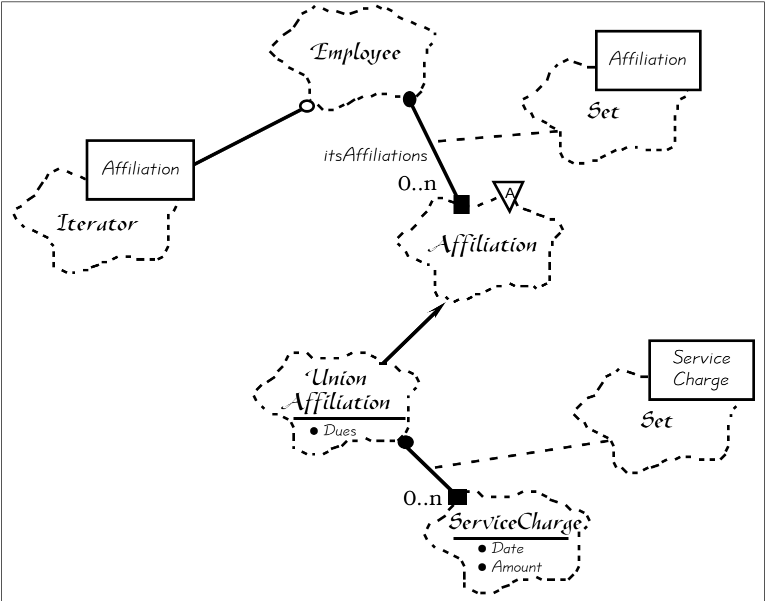


Figure 6
Static structure of Affiliation abstraction.

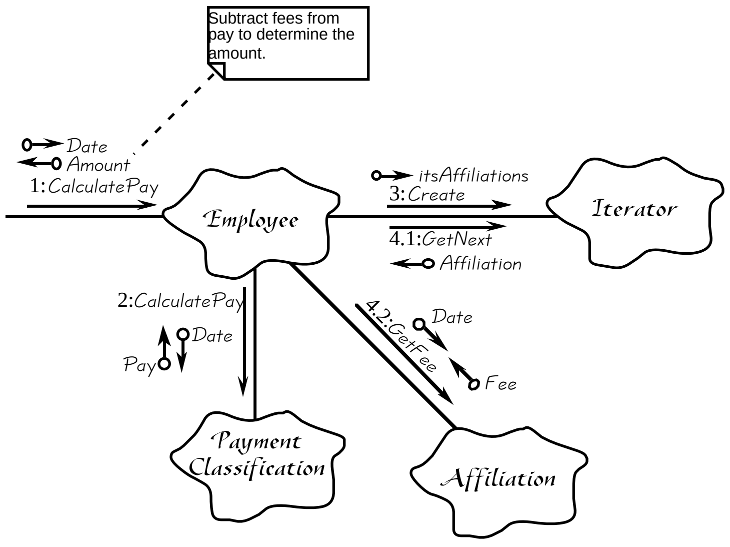


Figure 7
Dynamic structure of Affiliation abstraction.

Transactions

Looking again at the use cases we see that each transaction is like an object in that it contains data and has a behavior. The generality that can be applied to transactions is relatively simple. Transactions are built by some agency, and then they are executed. Figure 8 shows that this generality can be represented as an abstract base class named *Transaction* which has an instance method named *Execute()*.

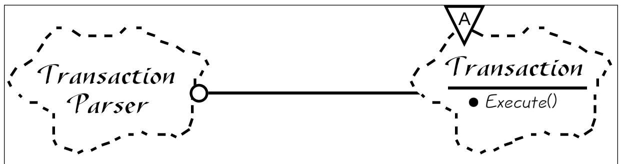


Figure 8
Abstract base Transaction class

This model allows several different processing modes. For example, *Transactions* can be executed as they are parsed. Or they can be parsed and then stored in an ordered set of *Transaction* objects for execution at a later date. Also, transactions become a convenient unit of work. If this system were ported to a GUI, then the GUI could build transaction objects and execute them. Also, since transactions have the capability to remember everything that they have done, they can be given the power to undo their actions. Thus if we ever needed an "undo" capability, we could provide a pure virtual *Undo* function in class *Transaction* and them implement this function in the derivatives.

Adding Employees

Figure 9 shows a potential structure for the transactions which add employees. Note that it is within these transactions that the employee's payment schedule is associated with his payment classification. This is appropriate since the transactions are contrivances instead of part of the real world model. Thus the real world model is unaware of the association; the association is merely part of one of the contrivances and can be changed at any time. For example, the application could easily be extended to allow schedules to be changed.

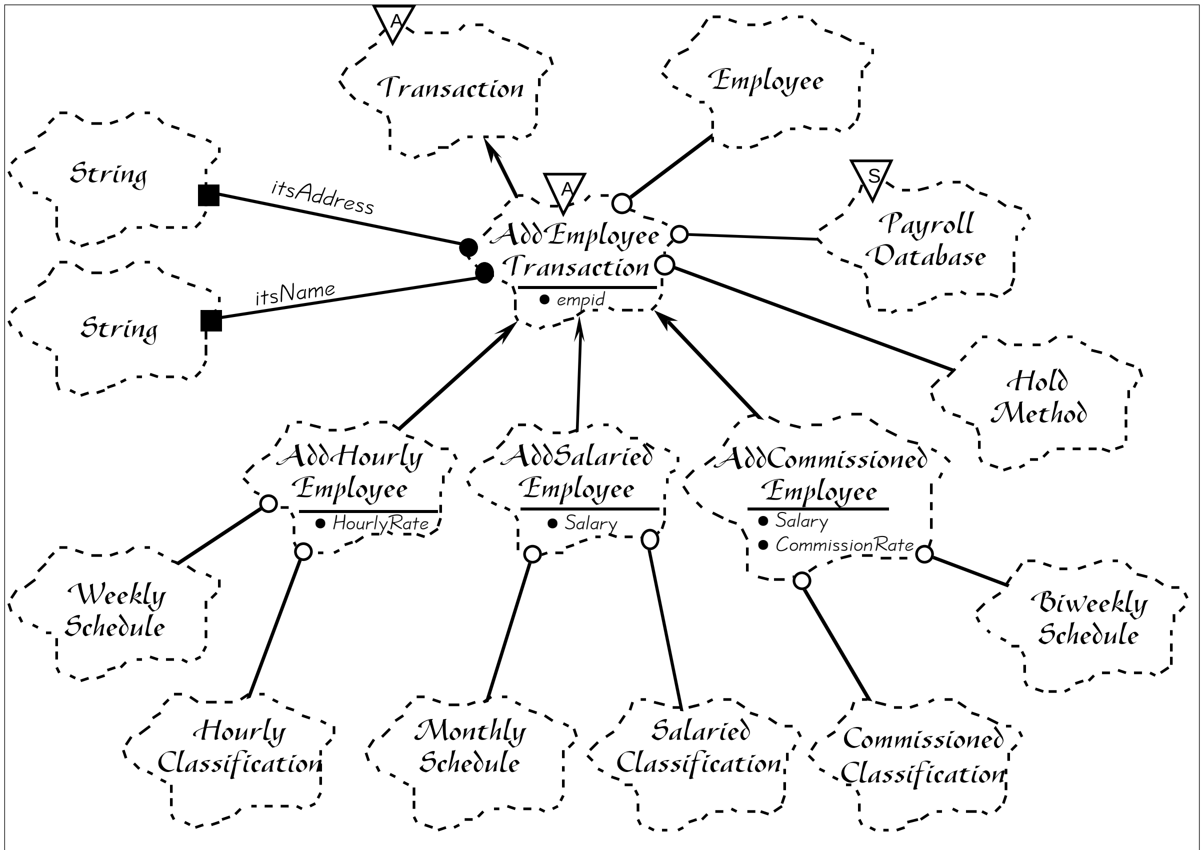


Figure 9
Static model of Add Employee Transactions.

Note too, that the default payment method is to hold the paycheck with the Paymaster. If an employee desires a different payment method, it must be changed with the appropriate `chgemp` transaction (see previous article in this series).

The *AddEmployee* class uses a class called *PayrollDatabase*. This class maintains all the existing *Employee* objects in a *Dictionary* which is keyed by `empid`. It also maintains a *Dictionary* which maps union member id's to `empid`s. The structure for this class is shown in Figure 10.

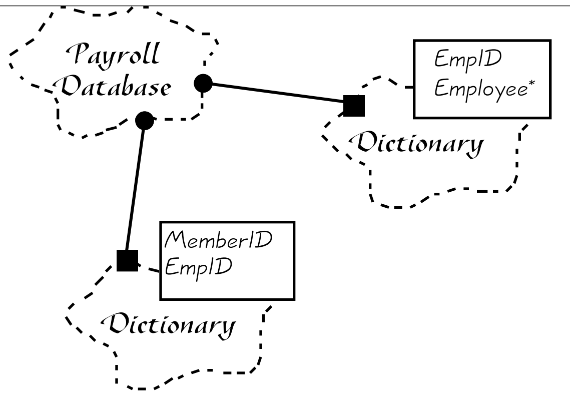


Figure 10
Static Structure of Payroll Database

Figure 11 shows the dynamic model for adding an employee. Note that the *AddEmployeeTransaction* object named X sends messages to *itself* in order to get the appropriate *PaymentClassification* and *PaymentSchedule* objects. These messages are implemented in the derivatives of the *AddEmployeeTransaction* class.

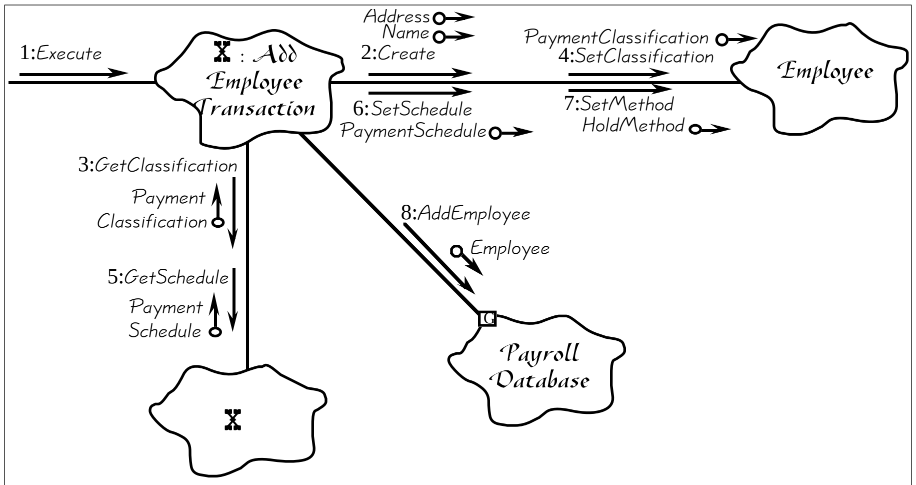


Figure 11
Dynamic model for adding an employee

Conclusion

Although there is still much left to be done to complete the analysis and design of the Payroll system, we are doing well. We have begun to understand the transaction model, and have found many of the abstractions that underly the problem domain. These abstractions have allowed us to produce designs for classes and modules that conform to the Open-Closed principle. For example, if ever we need a different kind of payment method, none of the current classes which describe the various methods of payment will need to change. Nor will any module which calls the Pay member function of the abstract *PaymentMethod* be required to change. Thus, these classes and modules are closed for modification, and yet they are open for extension and thereby lay the foundation for reuse and ease of maintenance.

In subsequent articles in this series we will study groups of classes called class categories; we will define what “cohesion” and “coupling” mean in an object oriented design; we will describe what is meant by the “inversion of source code dependencies”; we will develop metrics for measuring the quality of an object-oriented design; and we will discuss the benefits of design patterns such as “Object Factories”.