4-2006

# Realizing the Open-Closed Principle

Chong-wei Xu
*Kennesaw State University*, cxu@kennesaw.edu

Jose Hughes
*Kennesaw State University*



Recommended Citation
Chong-wei Xu, Jose Hughes, "Realizing the Open-Closed Principle," Computer and Information Science, ACIS International Conference on, pp. 274-279, Fourth Annual ACIS International Conference on Computer and Information Science (ICIS'05), 2005

# Realizing the Open-Closed Principle

Chong-wei Xu and Jose Hughes
Computer Science and Information Systems
Kennesaw State University
cxu@kennesaw.edu

## Abstract

*The first principle in developing large software systems is the Open-Closed Principle (OCP). This principle seems asking for two conflict goals. How to realize the principle in the real software practice? What are the enabling technologies that can be used to implement the principle? This paper uses a case study to demonstrate the importance of the principle, the design methodology for realizing the principle, and its enabling technologies.*

## Key words

Component-based Software Engineering, Software Reuse, Software Architecture, JavaBeans, Multithreading.

## 1. Introduction

The first principle in developing large software systems is the Open-Closed Principle (OCP). The principle says, "Software entities should be open for extension, but closed for modification" [1]. That is, a software system can go through extensions to satisfy new requirements such that increasing software's adaptability and flexibility but without modifying its existing source code.

Definitely, the principle is very important for software systems' maintainability and reusability. The maintainability is one of the most important considerations in software design and implementation because the maintainability deals with the life time of software and its cost is very high. Statistics said that the cost for maintenance is a double of the cost for its original development [2]. The problems that against easy maintenance include rigidity--adding a new module may affect many other modules; fragility--modifying a module leads problems in some places that look nothing to do with the modified module; immobility--a module relies on many other modules so that the module cannot be easily separated and reused; and viscosity--using ad-hoc way to modify a software is easier than using a systematic way, that is the software has no structures for extension [3]. The positive suggestions indicate that the correct software design should support extensibility--easy for adding new functionality (against rigidity); flexibility--a modification won't affect other modules (against fragility); and plugability--a class or a module can be plugged-and-played (against immobility and viscosity) [4]. Meanwhile, the reusability gains many benefits, including productivity, quality, and maintainability that are desired features of software systems.

Clearly, the realization of the OCP principle is tightly associated with the entire modeling, designing, and implementing process. However, it seems that the principle asks for unifying two conflict goals. How to realize the OCP principle in the real software practice? What are the enabling technologies that can be used to implement the OCP principle? This paper is intending to discuss these questions and using a case study to demonstrate the answers.

The case study that we are involving is the animated sorting algorithms that are the beginning part of a project, which proposes to develop component libraries for visualizing computer science algorithms, data communication algorithms, as well as some other fields including animated experiments in physics and animated molecular structures in biology. The reason for selecting the animated sorting algorithms as our starting point is that every one knows what a sorting algorithm is so that we don't need to spend much effort to explain the application but concentrate on the problem solving. There are many different sorting algorithms. We are looking for a well-structured design that is open for extension from one sorting algorithm to another but closed for

Proceedings of the Fourth Annual ACIS International Conference on Computer and Information Science (ICIS'05)
0-7695-2296-3/05 $20.00 © 2005 IEEE

modification. Even in many occasions we have seen the animated sorting algorithms, such as the SortDemo included in the JDK package and some examples provided by books, for example [5]. Due to the reasons either no source codes supplied or the implementation is not fit with our purpose, we started our own developing practice. From this case study, we can see what supports can be found from OOD (Object-Oriented Design) and OOP (Object-Oriented Programming) and what Java enabling technologies can be used for the implementation of the OCP principle.

Section 2 highlights the design methodology for realizing the OCP principle. Section 3 describes the enabling technologies: the JavaBeans and the multithreading. Section 4 discusses the implementation strategy for the case study. Section 5 illustrates how the case study realizes the OCP principle with the open for extension but closed for modification. Section 6 draws the conclusion and future work. Section 7 presents our acknowledgement for the support of this project.

## 2. The design methodology

The software practice for realizing the OCP principle should start from the very beginning of the modeling and designing processes. In general, software systems mainly consist of objects and their behaviors. For modeling and designing objects, our guide-line is the "dependency inversion principle". The dependency inversion principle [3] indicates, "depend on abstract, don't depend on concrete." The inheritance hierarchy and the interface definition support the possibility of extension. Designing an abstract level by using abstract classes and/or interfaces to predict all possible extensions in the future so that any extension won't modify this abstract level—it realizes the "closed" part of the OCP principle. At the same time, the abstract level can be extended by any concrete subclasses—it realizes the "open" part of the OCP principle. Thus, depending on abstract is the key for realizing the OCP principle.

Applications are different in thousand and one ways. How to define the abstract level as the root of inheritance hierarchy? Our guideline is the "encapsulation of variation" principle. This encapsulation of variation principle [6] says that to find the possible variations of a system, then encapsulate them and don't allow the variations spread over and don't mix one variation with another variation. Based on the specification of the application in development, looking for the variation portions and isolate them so that the relationships among variations would be decoupled and any modification in one variation won't affect the other variations. This kind of software realizes the extensibility, flexibility, and plugability. In other words, the domain of the inheritance hierarchy and the design of the abstract level depend on the analysis of the variations in the application.

After finishing the analysis and design, what are the enabling technologies that possible to implement the hierarchies and the entire application for satisfying the OCP principle is a further question should be answered.

## 3. The enabling technologies

Before implementation, we do need to determine what enabling technologies are suitable for the case study. Since the case study mainly involves visualization, definitely, the animation is the core part. An animation is made up of two components: objects and their animation algorithms. In the simplest notation, the objects are abstracted as their position coordinates (x, y) and the animation algorithms are abstracted as the computation for updating the coordinates (x, y). Both the objects and the animation algorithms could be any kind of, such as, rectangles, circles with line-moving, rotation-moving, and so on. That is, both the objects and the animation algorithms are the variations. The traditional way for programming the animation, as many books illustrated, is to define a class for the object and its motion behavior, such as a rectangle moves along the x-axis, a circle moves around a triangle path, and so on. If we ask for a rectangle moves around a triangle path, we need to define another class. In other words, this kind of design mixes the two variations in one place and is not easy for software maintenance and software reuse.

### 3.1. The JavaBeans technology

In order to satisfy the OCP principle, we selected the JavaBeans technology [7] to encapsulate the variations. Two beans are implemented, one is for the object and another one is for the

animation algorithm. Following the guide-lines that discussed above, we end up with two hierarchies as shown in Figure 1.

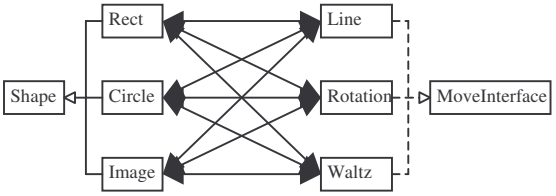


Figure 1. An abstract class hierarchy and an interface hierarchy.

One hierarchy uses an abstract class Shape as its root, another hierarchy uses an interface MoveInterface as its root. Both hierarchies may go over several levels eventually reach the concrete shapes, images, and different moving paths. The benefit of splitting the objects and the animation algorithms into two beans lies in the fact that an object bean, such as a rectangle bean, can be easily substituted by any object bean that implements a different shape, such as a circle, an image, and so on and an animation algorithm bean can be easily replaced by any bean that implements different moving algorithms, such as a waltz path, a rotating path, and so on. Consequently, different shapes and different animation algorithms can be pre-implemented as a library of beans. These beans can be selected and paired together to make different shapes with different animation behaviors. Any combination of two beans selected from the two hierarchies will make a different animation scenario.

As discussed above, the object bean paints itself based on its current coordinates (x, y) and the animation algorithm bean determines the updated coordinate (x, y). A new question arises: how can the animation algorithm bean knows the current coordinates of the object bean so that it can determine the updated coordinates based on the animation algorithm and how the object bean knows the updated coordinates generated by the animation bean? That is, what is the communication mechanism of the two beans? The communication mechanism is the bounded property of JavaBeans. A simple example, named Smiley, can demonstrate this communication mechanism. The Smiley consists of two beans. One is a Smiley face, another is a TextField as shown in Figure 2. The user can enter the age of the Smiley face using the TextField. Suppose the age threshold is defined as 30, that is, if the user enters the age is larger than the age threshold, the Smiley face will be "crying", otherwise the Smiley face will be "smiling". In other words, the behavior of the Smiley face bean is controlled by the age entered in the TextField bean. The control behind the scene is that the age is a bounded property defined in the TextField bean, whenever the value of the bounded property changes, an object with the type of PropertyChangeSupport will carry the change to a PropertyChangeListener that will pass the change to the Smiley face bean. Whenever the user changes the value of the age in the TextField bean, it is passed to the Smiley face bean so that the Smiley bean will either cry or smile [8]. In this case, the TextField bean is the sender and the Smiley face bean is the receiver. They form a uni-directional communication client-server relationship.

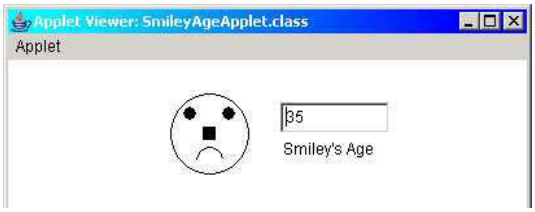


Figure 2. The Smiley face with its age.

Extending this idea to the animation case, the object's current coordinates (x, y) should be sent from the object bean to the animation algorithm bean and the updated coordinates (x, y) should be sent back from the animation algorithm bean to the object bean. They form a bi-directional communication client-server relationship as Figure 3 depicted.

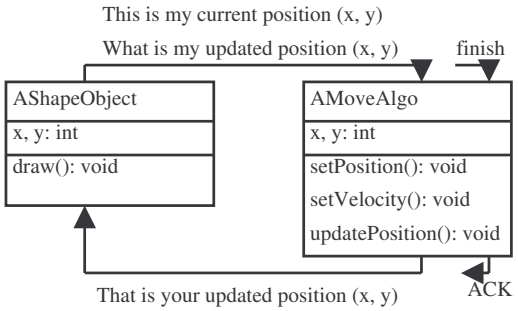


Figure 3. Bi-directional communication among beans

### 3.2. Multithreading

An animation is made by using the class Thread. A thread has a run() method that invokes a loop. The loop performs the following functions [9]:

```
while(true)
{
   display the current frame f;
   f = next frame;
   pause for a while;
}
```

The next frame contains the same objects with a slightly changed parameter, such as the updated coordinates (x, y), the angle, and so on. The loop thus displays a moving object or a skewed object that makes an animated effect.

In general, real applications consist of multiple objects and multiple animation algorithms with different behaviors, which need multiple threads. Multiple threads can work asynchronously. However, a thread has a set of status including runnable, blocked, sleep, wait, and dead. An application can globally control them and a thread can synchronize itself with other threads and vise versa by calling methods, such as sleep(), wait(), notify()/notifyAll(), yield() etc. The wait() method will block the threads invoked. The notify()/notifyAll() method will unblock the threads. The effect of these two methods can be seen clearly in a produce-consumer relationship.

A producer-consumer mechanism implements a generic mutual exclusion for accessing a critical section or a shared object. The producer can write a new value into the shared object only if the shared object is empty. The consumer can get a new value out of the shared object only if the shared object contains a value. If implementing the producer and the consumer as two threads, that is, they are running asynchronously, a Boolean variable, say writeable, is needed to control the access to the shared object as depicted in Figure 4 [10].

Figure 4 says that only when the Boolean variable writeable is true, the producer can call the setSharedObject() method for putting a new value into the shared object, and at the same time, the method sets the writeable as false to keep the producer away from the shared object and to allow the consumer to call the getSharedObject() method for getting the value held by the shared object. Inside the setSharedObject() and getSharedObject() methods, the wait() and notify() methods are used to implement this mutual exclusion no matter the running speed of two threads. Two threads, the producer and consumer beans, run in parallel that form a peer-to-peer relationship and their communication is forced by the shared object.

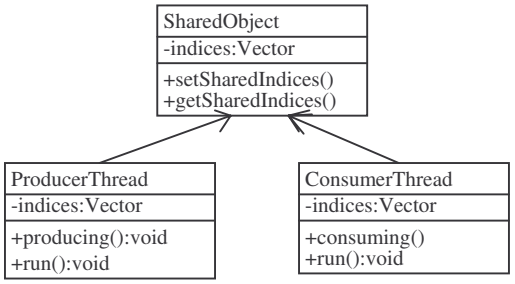


Figure 4. The producer-consumer threads and their control.

## 4. The animated sorting algorithms

The case study for demonstrating the OCP principle is the animated sorting algorithms. We started with the select sorting algorithm. The fundamental idea of the selection sort is that given an array of random numbers, the algorithm loops for swapping the number in the "start" index with the number in the "minimum" index to make the set of numbers end up with the ascending or descending order. Analyzing the project based on the design methodology presented in Section 2, the animated selection sorting algorithm can be divided into two parts: an animation display part and the sorting algorithm part. The sorting algorithm provides a pair of indices as a producer and the animation display part consumes the pair of indices. The animation display part uses a set of bars (rectangles) for representing the set of random numbers and two of them corresponding to the pair of Rec indices provided by the sorting algorithm are animated for presenting the swap process. Obviously, the objects are the set of RectBeans and the animation algorithm is the LineBean along the x-axes that can be found from the library shown in Figure 1. The LineBean updates the coordinates of two tBeans step-by-step until they swapped their positions.

In this way, the display part is rapidly implemented by using the component library.

The following step is to put the sorting algorithm and the display part together. The first version of the implementation uses one thread only. The thread loops over the selection sorting algorithm and the animation display part sequentially. Because the number of animation steps depends on the distance between two swapping bars, the sorting algorithm has to generate the same indices so many times as the number of steps needed for animating two bars. It is a kind of busy waiting that wastes the resources. It is even worse that the selection sorting algorithm is embedded in the loop coding. Whenever the sorting algorithm would be changed, say animating the bubble sorting algorithm, the existing source code must be modified. It is not a well-structured design for extensions because it violates the OCP principle due to the fact that we mix the variation for changing the sorting algorithms with the animation display together.

The better design is to apart the sorting algorithm from the animation display so that they can be implemented as separated variations and reusable components as discussed in Section 2. Following this design strategy, a thread, named SortAlgo, is implemented for providing the pair of indices of "start" and "minimum". Another thread, named SortShow, is implemented to illustrate the dynamic behaviors of the sorting algorithm. In addition, the third thread, named DisplayThread, is added. These three threads perform the following actions, respectively:

1. Thread SortAlgo reads the data array and generates a pair of indices
2. Thread SortShow gets the indices and lets the LineBean to update the coordinates of two bars
3. Thread DisplayThread swaps two bars step-by-step and eventually write the new coordinates into the data array when the swap finishes

These actions are related with each other. The first action can be started only after the third action has been finished and the second action can be started only after the first action has been done because the three threads share two physical resources. The resource "indices" are shared by the threads SortAlgo and SortShow and the resource "data array" is shared by the threads SortAlgo and DisplayThread.

Clearly, a question is raised as how to synchronize these two pair of threads that have a peer-to-peer relationship. A producer-consumer mechanism that discussed in Section 3 is used for controlling the indices writing and reading as shown in Figure 4. The producer thread, named SortAlgo, uses the SelectionSortBean to generate the shared indices and calls the method sharedObj.setSharedIndices() to write the indices into the shared object. The consumer thread, named SortShow, calls the sharedObj.getSharedIndices() method to get the indices from the shared object [10]. Meanwhile, the share in data array forces the two threads, namely SortAlgo and DisplayThread, forming a suspend-resume relationship. After the thread SortAlgo found the pair of indices by reading the data array, it should be suspended until the thread DisplayThread finishes one swap of two bars, which modifies the data array and then resumes the accessing to the data array [11]. All classes, threads, and their relationships are illustrated in Figure 5.

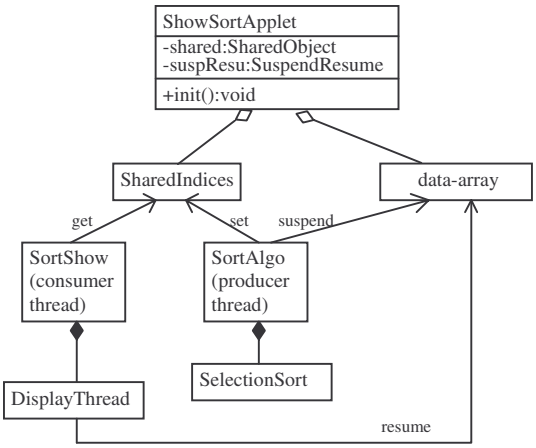


Figure 5. The class diagram of the entire application.

## 5. Extending the application

From Figure 5 we can see that the SelectionSort algorithm is a totally independent component that belongs to the independent thread SortAlgo. In other words, we can recognize the SortAlgo is the manager of the sorting algorithms. Therefore, we can easily implement multiple sorting algorithms that are under the SortAlgo's control and allow the user to select one of them.

Other control functions, such as changing the number of data in the data array, scrambling the data, switching ascending or descending order, and so on can be added easily. Similarly, the SortShow is the manager of the DisplayThread. It can switch the display screen from the animation display to a curve plot so that when the user would like to comparatively study the performance of multiple sorting algorithms, the user can select multiple sorting algorithms and start the performance measurement. The display portion of the screen will show a coordinate system that plots multiple curves representing the performance of multiple sorting algorithms, respectively. All these extensions are summarized in Figure 6. Even we can change the producer-consumer and the suspend-resume peer-to-peer relationships to the client-server relationships without affect the sorting algorithms and animation display. Evidently, this well-structured design realizes the OCP principle.

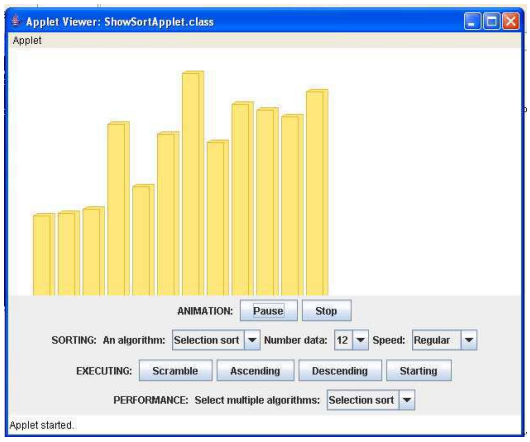


Figure 6. Extensions for the animated sorting algorithm application.

## 6. Conclusion and future work

The software architecture of the case study defines a framework for animating all kinds of sorting algorithms and it is open for extension but closed for modification. That is, this case study illustrates the possibility to realize the OCP principle, demonstrates the design methodology, and explores the enabling technologies for the real implementation. These design strategy and implementation technology can also be applied for computer game design and implementation to make games and their components easier to be maintained and reused.

We will further design and implement more applications, such as animated tree and graph data structures, animated experiments in physics, and animated molecular structures in biology, to deeply study the OCP principle. In addition, we will further study patterns and apply them for realizing the OCP principle [12].

## 7. Acknowledgement

This project is partially supported by the College Mentor-Protégé grand funded by the College of Science and Mathematics, Kennesaw State University.

## References

[1] Bertrand Meyer, "Object Oriented software Construction", Prentice-Hall, 1988.
[2] Walker Royce, "Next-Generation Software Economics", The Rational Edge, 11, 2000.
[3] Robert Martin, "Design Principles and Design Patterns", www.objectmentor.com, 2000.
[4] Peter Coad, Mark Mayfield, and Jon Kern, "Java Design – Building Better Apps and Applets", Prentice-Hall, 1999.
[5] Robert Lafore, "Data Structures and Algorithms in Java", The White Group, Inc., 1998.
[6] Alan Shalloway and James Trott, "Design Patterns Explained – A New Perspective on Object-Oriented Design", Addison-Wesley, 2001.
[7] JavaBeans, http://java.sun.com/products/javabeans/
[8] A. Gittleman, "Internet Applications with the Java 2 Platform", Scott/Jones Inc. 2001.
[9] J. Fan, E. Ries, and C. Tenitchi, "Black art of Java Game Programming", Waite Group Press, 1996.
[10] Deitel and Deitel, "Java How to Program", 6/e, Prentice-Hall, 2004.
[11] C.S. Horstmann and G. Cornell, "Core Java Volume II – Advanced Features", PH-PTR, 2002.
[12] Hong Yan, "Java and Patterns", (in Chinese), Publishing house of Electronics Industry, 2004.