Abstract—Producer-Consumer problem is a classical and representative synchronization problem in the field of computer science, which can be implemented by object-oriented programming language. However, the implementation of object-oriented programming often leads to being tangled between functional codes and synchronization codes, which are easy to lead code-scattering and code-tangling. Aspect-oriented programming (AOP) has been proposed as a technique for improving separation of concerns for the final software. This paper aims to resolve concrete aspect and implement the synchronization of producer-consumer based on AOP under two ways: single buffer and multiple buffers, and the execution result shows that AOP can resolve the producer-consumer problem effectively. Furthermore, the execution time of AOP program is shown to contrast with that of object-oriented programming, which shows that AOP can almost get the same execution time with object-oriented programming, while AOP can gain the modularization better than object-oriented programming.

Keywords—Producer-Consumer problem; synchronization; Aspect-oriented programming; separation of concerns

I. INTRODUCTION

In the field of computer science, the producer-consumer problem is a classical example of the multi-process synchronization problem. Synchronization is an important and familiar problem in the design and development of the software, and many applications use synchronization as their method and settlement, such as Synchronous Product [1] and FlexSync[2] etc. When multiple processes or threads access the critical resource, synchronization is needed to be control.

Nowadays, many object-oriented programming languages have supported the synchronization and can implement the producer-consumer problem. For example, Java programming language implements the synchronization through the keyword synchronized, as the prefix of the method, that allows only one thread enters the synchronized code at the same time and avoid abusing the critical resource. Java can also control the communication among the thread by the methods: wait(), notify() or notifyAll(). The method wait() keeps the thread waiting until the method notify() notify it to end the waiting. Besides Java, many object-oriented programming (OOP) languages have resolved the synchronization well. However, the implementation of OOP leads to be tangled between the function codes and nonfunctional codes, which are easy to lead code-scattering and code-tangling. It is not beneficial to the development and maintenance of the software.

Aspect-Oriented Programming (AOP) was first proposed in [3] as a programming technique for modularizing concerns that cross-cut the basic functionality of programs. The aim of AOP is to resolve the code-scattering and code-tangling and modularize the crosscutting concerns. The crosscutting concerns include security, logging, exception handling and synchronization etc. Many distinguish work has done to deal with the discrete aspect. Zhu[4] implements the producer-consumer problem based on JAC that implements the concurrency based on the extended JAVA comments, but the JAC needs the specific compiler.

Though much work has done in the specific aspect, there is less work on the producer-consumer problem using AOP. As the producer-consumer problem is a representative problem in synchronization, the solution will help to the software design and programming.

This paper takes the classical producer-consumer problem as an example to provide the solution to the synchronization using AOP. In section 2, the implementation of producer-consumer problem is presented. In order to present explicitly, the implementation using single buffer and multiple buffers are shown separately. In section 3, the comparison of execution time is done between OOP and AOP. Section 4 concludes the paper.

II. IMPLEMENTATION USING AOP

This section presents the implementation of producer-consumer problem using AOP under two manners: single buffer and multiple buffers.

A. Description of the Producer-Consumer Problem

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
The solution for the producer is to go to sleep if the buffer is full. The next time the consumer removes an item from the buffer, it wakes up the producer who starts to fill the buffer again. In the same way the consumer goes to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

### B. Implementation using Single Buffer

In this mode, there is only one buffer that can be used by producer and consumer. Once the producer produces a product, he puts it into the buffer and waits for the consumer to consume the product. The state of the buffer is full. After the consumer has consumed the product, he will be blocked and wait for the producer to produce, then the buffer is null.

The implementation code of class CubbyHole, that includes the buffer, is shown in figure 1. In class CubbyHole, the variant `goods` act the role of the buffer and the variant `available` is used to judge whether the buffer is full or not.

```java
public aspect SynAspect {
    pointcut syncPut(): call(void CubbyHole.put(int,int));
    before(): syncPut(){
        synchronized(this){
            while(CubbyHole.available==true){
                try {wait();}
                catch (InterruptedException e) {
                }
            }
            CubbyHole.available=true;
        }
        after() returning: syncPut(){
            synchronized(this){notify;}
        }
    }
    pointcut syncGet(): call(int CubbyHole.get(int));
    before(): syncGet(){
        synchronized(this){
            while (CubbyHole.available == false) {
                try {wait();}
                catch (InterruptedException e) {
                }
            }
            CubbyHole.available=false;
        }
        after() returning: syncGet(){
            synchronized(this){notify;}
        }
    }
}
```

**Figure 2. The source code of synchronization Aspect**

The source code of Class CubbyHole, as shown in figure 1, doesn’t include the code that related to the synchronization control, such as communication between producer and consumer. The synchronization control is abstracted to be an aspect that is shown in figure 2. In aspect SynAspect, two pointcuts and its advice are built. The two pointcuts have the different advice. Every advice includes the before advice and after advice.

The programming tools that we employed is the Eclipse 3.3 and the AspectJ5 that are used to execute the program. The AspectJ plugin ajdt_1.5.2 for eclipse_3.3 is used to collaborate with Eclipse to get the result. The execution result is shown in figure 3. Besides the code as shown in figure 1 and figure 2, some other threads including of producer thread, consumer thread and main thread is built in the program. We employ two producer instance (be signed as 1 and 2) and two consumer instance (be signed as 1 and 2) in the program. Seen in the figure 3, when the producer produces a product, the consumer consumes the product. Conclusion from the figure 3, the producer and consumer can be synchronized.

**Figure 3. The execution result using single buffer**

### C. Implementation using Multiple Buffers

In this mode, there are multiple buffers between producer and consumer. The producer can continue to put the product into the buffers until the buffer is full, while the consumer continues to get the product from the buffer until the buffer is null. Different from the single buffer, this way needs more communications between producer and consumer.

During the implementation of multiple buffers, the main work is to identify the crosscutting concerns, which need to separate from the functional parts and to be modularized as an aspect. As existing shared resource (the buffers) and the producer or the consumer need to access the buffer, synchronization is separated and abstracted to be an aspect.
After removing the code related synchronization, the result of abstraction is shown in Figure 4. Seen from it, there is a class CubbyHole we defined. The method get() is charged of getting the product from the buffer, and method put() is responsible to put the product into the buffer. The buffer size is four.

class CubbyHole {
    // the buffer length is four
    private int[] goods=new int[4];
    // the point of first producer
    private int front;
    // the point of last producer
    private int rear;
    // the number of products in the buffers
    public static int count;
    // the number of blocked consumers
    private int getter;
    // the number of blocked producers
    private int putter;
    public CubbyHole(){
        front=0;
        rear=0;
        }
    public synchronized int get(int id) {
        front=(front+1)%4;
        System.out.println("Consumer #" + id + " got: " +
        goods[front]);
        count--;
        return goods[front];
    }
    public synchronized void put(int value, int id) {
        rear=(rear+1)% 4;
        goods[rear]=value;
        System.out.println("Producer #" + id + " put: " +
goods[rear]);
        count++;
    }
    }
}

Figure 4. The result of abstracting synchronization code

After abstracting the synchronization code, the next step is to construct a synchronization aspect that includes joint point, pointcut and advice. The synchronization aspect that named as SynAspect is shown in Figure 5. Joint point, being the place where pointcut captures, is usually selected in some specific methods. Here is the method put() and get(). In the definition of the pointcut, the head of put() and get() method is needed and the method begin with the name of class CubbyHole as shown in line 6 and 23 of Figure 5.

Advice is also an important part of an aspect. Advice includes before advice, after advice and around advice. Before advice and after advice are implemented for every pointcut. In figure 5, we present the before advice and after advice of pointcut, named as syncGet() and syncPut(). For example, the task of before advice of pointcut syncGet() is to keep the producer waiting when the buffer is full. The task of after advice of pointcut syncGet() is to notify the consumer to consume the product when there are some consumers waiting for the product.

The execution result is shown in Figure 6 with the tool Eclipse 3.3 and AspectJ. AspectJ allows Java programmers to divide program code into separate pieces, each of that describes a different aspect. These pieces are merged at join points by the AspectJ compiler into a single regular program. Conclusion from Figure 6, AOP can resolve the producer-consumer problem.

Figure 5 implementation of synchronization aspect

Figure 6. The execution result using multiple buffers
III. EXPERIMENTATION

The execution time is compared between AOP and OOP. In the experimentation, the program includes four threads: two producer threads and two consumer threads. The hardware and software environment is as following. In the aspect of hardware, the frequency of CPU is Intel Core™ 2 Duo T5600 1.83GHz and the capacity of memory is 1G. In the aspect of software, operating system is Windows XP, and the software uses Eclipse 3.3 and AspectJ’s Eclipse plug-in: ajdt_1.5.2_for_eclipse_3.3. We separately test the execution time according to the OOP and AOP implementation in different execution times. As the experimentation is performed in multi-thread environment and the call of thread is uncertain, we performed eight times for in different execution times-5, 50, 500 and 5000. The result of execution time is shown in TABLE I.

<table>
<thead>
<tr>
<th>NO.</th>
<th>5</th>
<th>50</th>
<th>500</th>
<th>5000</th>
<th>5</th>
<th>50</th>
<th>500</th>
<th>5000</th>
<th>5</th>
<th>50</th>
<th>500</th>
<th>5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>15</td>
<td>125</td>
<td>688</td>
<td>16</td>
<td>47</td>
<td>125</td>
<td>656</td>
<td>0</td>
<td>16</td>
<td>94</td>
<td>579</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>15</td>
<td>110</td>
<td>641</td>
<td>31</td>
<td>31</td>
<td>140</td>
<td>578</td>
<td>16</td>
<td>16</td>
<td>125</td>
<td>500</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>16</td>
<td>125</td>
<td>687</td>
<td>15</td>
<td>31</td>
<td>141</td>
<td>594</td>
<td>15</td>
<td>15</td>
<td>141</td>
<td>531</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>15</td>
<td>125</td>
<td>657</td>
<td>31</td>
<td>32</td>
<td>125</td>
<td>569</td>
<td>16</td>
<td>31</td>
<td>94</td>
<td>516</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>32</td>
<td>110</td>
<td>672</td>
<td>16</td>
<td>47</td>
<td>125</td>
<td>610</td>
<td>0</td>
<td>16</td>
<td>109</td>
<td>500</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>31</td>
<td>109</td>
<td>687</td>
<td>16</td>
<td>31</td>
<td>140</td>
<td>563</td>
<td>0</td>
<td>32</td>
<td>109</td>
<td>500</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>15</td>
<td>125</td>
<td>640</td>
<td>47</td>
<td>31</td>
<td>125</td>
<td>594</td>
<td>16</td>
<td>31</td>
<td>94</td>
<td>516</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>16</td>
<td>109</td>
<td>672</td>
<td>15</td>
<td>32</td>
<td>156</td>
<td>578</td>
<td>0</td>
<td>31</td>
<td>110</td>
<td>500</td>
</tr>
</tbody>
</table>

As shown in TABLE I, the execution time of AOP is very close to that of OOP. When the execution times is very few(eg. 5 times), the execution time of OOP may be distinct from that of AOP. For example, the execution of OOP is zero sometimes while the execution of AOP is not zero. Sometimes the execution time of AOP is less than that of OOP. We can conclude that the execution effective of AOP is not declined, though AOP needs to capture the joint point and execution the advice. AOP can obtain almost the same execution time with object-oriented programming, but using AOP can gain the modularization better than OOP.

IV. CONCLUSION

The main contribution of this paper is that the producer and consumer problem is implemented using AOP and the execution time of AOP is comparison with that of OOP. The result shows that AOP is the supplement of OOP. AOP can obtain the separation of concerns and make the function parts more reuse and functional cohesion without losing efficiency. Our work will benefit to the development and maintenance of the software that related to the synchronization.

ACKNOWLEDGMENT

It is a project supported by Natural Science Foundation of Hebei Province under Grant No.F2009000852, P.R. China, and Science-Technology Foundation of Hebei Province under Grant No.07215601D-3, P.R. China.

REFERENCES