Arch-pattern based design and aspect-oriented implementation of Readers- Writers concurrent problem

Dumitru Ciorbă, Victor Beşliu, Anthony Chronopoulos, Andrei Poștaru

Abstract
The classical problems of concurrent programming start from the design problems of operating systems in the 80-s. But today there are still proposed new solutions for these problems with the help of various design and programming approaches. The present article describes a solution which was designed according to some new object-oriented principles, based on design patterns and proposes two program solutions: firstly – an object-oriented implementation in Java language, the secondly – an aspect-oriented one in AspectJ language.

Key words: concurrency, design arch-pattern, aspect, Java, AspectJ.

1 A new solution?

Concurrent programming is the science determined by the art of organizing cooperation. Most of the times, this art implies using some programming techniques developed on certain technological support. In other words, if one has at his disposal object-oriented languages, the problems of concurrency should be approached only in an object-oriented context.

The implementation of concurrency by object based techniques is accompanied by a number of difficulties generically described as inheritance anomalies. Understanding them by means of numerous classification attempts [1, 2] helps to determine the causes of this phenomenon.

©2007 by D. Ciorbă, V. Beşliu, A. Chronopoulos, A. Poștaru

338
Arch-pattern based design and aspect-oriented implementation of …

This means that organizing the cooperation (implemented with class methods) cannot be inherited by subclasses without explicit modifications.

At the same time, classical approaches of generalization and modularity subjected to the dominant decomposition tyranny [3, 4] cannot be used for determining specific properties of concurrent systems, such as synchronization, resource allocation, security, persistence, etc.

Thus, our task is to propose a new solution for the classical Readers-Writers problem based on flexible techniques of organizing concurrent activities deprived of inheritance anomalies and with no scattering of the non-functional requirements’ code in the entire system.

2 Formulating the problem

The Readers-Writers problem is a modification of the problem of mutual exclusion. It implies the existence of some readers that may have simultaneous access to a book (critical section) and of some writers, who have an exclusive access to it.

The policies of synchronization and access scheduling determine the existence of several problem versions. However, we will agree with the formulation described in [5] (also proposed in [6]), that associates an executed concurrent process to each reader and each writer. Their activities will consist of opening and closing the book.

One should distinguish between the operations executed by readers and the ones executed by writers. When we plan access we make the following assumptions:

a) If none of the processes has entered the critical section (did not open the book) and there are readers and writers who wish to do so, a writer will be chosen;

b) If a writer exited the critical section (closed the book), and there are readers and writers who wish to enter the critical section, readers will be chosen.
3 Functional and non-functional requirements of the problem

Identifying requirements is a necessary process that results in a specification, determined by the vision of the users on the system. The analysis of the requirements obtained implies creating a new model that developers will be able to interpret in a non-trivial way. On the initial stages this fact will ensure a decomposition that will propose a code that would not be either entangled or scattered [7] by dividing all the functional and non-functional requirements into modules.

Functional requirements determine the behavior of the system, in terms of the services it supplies. The non-functional requirements impose some restrictions onto these services that may affect the performance and the semantics of the system. By employing Use Cases (the Use Cases describe a system’s functionality), UML allows to specify functional requirements (Tables 1 and 2 display their partial definition), and also to offer a possibility to identify some scattered restrictions, that must be placed in separate language structures.

Table 1. Use case Open (Reader actor)

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Actor</th>
<th>Pre-condition</th>
<th>Post-condition</th>
<th>Basic scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Reader</td>
<td>No writer writes in the book (synchronization)</td>
<td>A writer will open the book for reading that may be performed simultaneously by several readers.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Incrementing the number of active readers (synchronization)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No writer is waiting to write (scheduling)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In Figure 1 one can see the use case diagram for the Readers-Writers
Table 2. Use case Close (Reader actor)

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Close</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor</td>
<td>Reader</td>
</tr>
</tbody>
</table>
| Pre-condition | Decrementing the number of the active readers (synchronization)  
Will offer the writers the first chance of opening, if there are no active readers (scheduling) |
| Basic scenario | The Reader closes the book |

Table 3. Use case Open (Writer actor)

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor</td>
<td>Writer</td>
</tr>
</tbody>
</table>
| Pre-condition | No writer is writing in the book (synchronization)  
There is no active readers (scheduling)  
The first chance for opening will be offered to the writers who expressed their wish to open the book before any reader did (scheduling)  
Incrementing the number of active writers (synchronization) |
| Post-condition | The writer will open the book which can be read by only one writer |
| Basic scenario | The writer will open the book which can be read by only one writer |
Table 4. Use case Close (Writer actor)

<table>
<thead>
<tr>
<th>Use case</th>
<th>Close</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor</td>
<td>Writer</td>
</tr>
<tr>
<td>Pre-condition</td>
<td>A first chance to open will be offered to the readers who expressed a wish to open the book before the writers did (scheduling) Decrementing the number of active readers (synchronization)</td>
</tr>
<tr>
<td>Post-condition</td>
<td>A writer will close the book that may be read by several readers or written by one author.</td>
</tr>
</tbody>
</table>

Figure 1. Use case diagram for the Readers-Writers problem
problem. It can be noticed that certain requirements (like Synchronization and Scheduling) expressed by pre-conditions or post-conditions are scattered (implemented in several use cases).

4 The StrategyProxy – design arch-pattern

Creating systems designs that correspond to reusability principle when there are concurrent activities is a difficult task because one must obtain a specific solution for the problem which can be reused.

Design patterns are a way for finding such solutions. Many works, as [8] and [9], confirm this fact presenting a number of design patterns, which can be accepted for describing abstract interactions between the system objects. The design patterns described in [9] have aroused considerable interest. They are based on primitives of Java language and intended for concurrent design.

The advantage gained by use of design patterns is obvious because using them enhance reusability of tested and successful solutions, and saves development time. We can add here that systems can easily be described using patterns because they are directly determining the use cases. Still, the use of patterns remains unsystematic and must be improved.

The fact that, when looking for a solution of a problem, the patterns describe the elements (classes and objects) and the interactions between them [8], allows us to consider them also architectural patterns, even if they determine micro-architectures [11].

The presented design method, in contrast to many existing, is an inductive one in which the design of the architecture comes at the “unification” of the components, which can be classes or design patterns from a specific catalog. The synthesis is realized through connectors which can be classes (objects) shared by two or more patterns, or can just not exist if there can be established a direct relationship between the classes (objects) of the unified patterns. The result of the synthesis is a super-structure of design patterns which define a micro-architecture. This is why the structure has been named arch-pattern in order to avoid the eventual confusion with the architectural pattern
term, which describes the system on a more abstract level [12].

The analysis of Readers-Writers problem has resulted in a class model named **StrategyProxy** which should be used when:

a) We need an intermediary which will represent and manage access to an object,
b) The object that is called may be in a different address space than the one of the clients,
c) Clients can have different access rights,
d) Clients can have different behavior, defined independently of them.

The motivations for this class model combine those of the **Strategy** and **Proxy** design patterns from [8]. That’s why, for better understanding, it was decided that its name should combine the names of the above patterns and **StrategyProxy** should be identified as a design arch-pattern (Figure 2).

![figures]

Figure 2. Design arch-pattern **StrategyProxy** applied to the Readers-Writers problem
At this point the design arch-pattern can already provide some information regarding the inheritance anomalies, and an object-oriented implementation can underline that.

5 Object-oriented implementation of the problem in Java

The active entities (Reader and Writer threads) of the concurrent system will call Open and Close methods (Figure 3) applied on the object defined by the BookIF interface, but will realize the tasks with their own strategies (Figure 5).

```java
public class Reader extends Thread{
    public int Id;
    public BookIF m_Book;
    public Reader(int Id, BookIF ReadStrategy){
        this.Id = Id;
        m_Book = ReadStrategy;
    }
    public void run(){
        m_Book.Open();
        m_Book.Close();
    }
}
```

Figure 3. Java implementation of Reader class

The StrategyProxy element from Figure 2 will be implemented as an abstract class (Figure 4) which will include the static variables activeReaders, waitingReaders, activeWriters, waitingWriters and preferWriters for the common access policy and will define a Book interface for the ReadStrategy (Figure 5) and WriteStrategy strategies.
public abstract class StrategyProxy implements BookIF {
    public static Book m_Book;
    protected static int activeReaders = 0;
    protected static int waitingReaders = 0;
    protected static int activeWriters = 0;
    protected static int waitingWriters = 0;
    protected static boolean preferWriters = true;
    static{
        m_Book = new Book();
    }
    public synchronized void Open(){ }
    public synchronized void Close(){ }
}

Figure 4. Java implementation of StrategyProxy class

The ReadStrategy class will implement the access policies of the Readers described in the pre-conditions and the post-conditions of the use cases using CheckReadAvailability() and ClearReaderPresence().

The pass-through condition [while (activeWriters != 0 || (waitingWriters != 0 && preferWriters)) from ReadStrategy.CheckReadAvailability()] is determined by the formulation of the problem which denies the access to the Book if there are active Writers (activeWriters != 0) or there are waiting Writers (waitingWriters != 0) who have announced their wish to access the book earlier or are preferred (preferWriters=true).

The waking of the suspended threads from ReadStrategy.ClearReaderPresence() is conditioned by the fact that all the threads of the Readers must end their activity. This will allow all new Readers to access the book independently of the number of suspended Writers (they cannot modify the value of the preferWriters variable in WriteStrategy.CheckReadAvailability()). The condition [if(waitingReaders==0) preferWriters = true] is necessary only to prevent the halting of the last thread of the Writers and not for implementation of the access policies to Book.
public class ReadStrategy extends StrategyProxy {
    int Id;
    public ReadStrategy(int Id){
        this.Id = Id;
    }
    public synchronized void Open(){
        CheckReadAvailability();
        m_Book.Open(); // Common for read/write operation
        m_Book.Read(); // Read operation
    }
    public synchronized void Close(){
        m_Book.Close();
        ClearReaderPresence();
    }
    private void CheckReadAvailability(){
        waitingReaders++;
        while (activeWriters != 0 ||
            (waitingWriters != 0 && preferWriters)){
            synchronized(m_Book)
            {
                try{ m_Book.wait(); } catch(Exception ex) { }
            }
        }
        waitingReaders--; activeReaders++;
        synchronized(m_Book)
        {
            notifyAll(); // announce All waiting readers
        }
    }
    private void ClearReaderPresence(){
        activeReaders--;
        if(activeReaders == 0) {
            if(waitingReaders==0) preferWriters = true;
            synchronized(m_Book)
        }
    }
}
The major inconvenience of this solution consists in the necessity to redefine the entire method if a derived class from ReadStrategy wishes to change the synchronization behavior of Open and Close methods.

6 Aspect-Oriented implementation of the problem in AspectJ

After the analysis of design patterns presented in [8] and [9] it was discovered that there are no solutions for the description of the interactions between the functional and non-functional code. The aspect-oriented approach offers such a possibility and allows us to localize the scattering of the requirements in separate constructive units named aspects.

In the problem Readers-Writers, the non-functional requirements such as, the synchronization, scheduling, multiple and exclusive access (Figure 1) are implemented mandatory as an aspect. This will allow an anomaly-less inheritance of the classes that realize the functional requirements. Moreover, the weaving mechanisms [3, 10] of aspects with classes allow us to simplify the system development, eliminating the classes of the strategies for the Readers and the Writers (Figure 2).

The description of the specific behavior of Readers and Writers and the identification of these will be implemented by means of special aspect constructions pointcut and advice of the AspectJ language (Figure 6).
public aspect StrategyAspect{
    protected int activeReaders = 0;
    protected int waitingReaders = 0;
    protected int activeWriters = 0;
    protected int waitingWriters = 0;
    protected boolean preferWriters = true;
    pointcut ReaderOpen(Reader reader, Proxy proxy): call(* *.Open())
        && this(reader) && target(proxy);
    pointcut ReaderClose(Reader reader, Proxy proxy): call(* *.Close())
        && this(reader) && target(proxy);
    pointcut WriterOpen(Writer writer, Proxy proxy): call(* *.Open())
        && this(writer) && target(proxy);
    pointcut WriterClose(Writer writer, Proxy proxy): call(* *.Close())
        && this(writer) && target(proxy);
    before(Reader reader, Proxy proxy): ReaderOpen(reader, proxy)
        waitingReaders++;
    while (activeWriters != 0 || (waitingWriters != 0 && preferWriters))
    {
        synchronized(proxy.m_Book)
        {
            try{ proxy.m_Book.wait(); }catch(Exception ex){}
        }
        waitingReaders--; activeReaders++;
    }
    after(Reader reader, Proxy proxy): ReaderOpen(reader, proxy)
    {
        synchronized(proxy.m_Book)
        {
            proxy.m_Book.Read();
        }
    }
    after(Reader reader, Proxy proxy): ReaderClose(reader, proxy)
    {
        activeReaders--;
        if(activeReaders == 0) {
            }
if(waitingReaders)preferWriters = true;
synchronized(proxy.m_Book)
{
    proxy.m_Book.notifyAll();
}
}

before(Writer writer, Proxy proxy):WriterOpen(writer, proxy){
    waitingWriters++;
    while(activeWriters <= 1 || activeReaders != 0 || !preferWriters){
        if(activeReaders != 0) preferWriters = true;
        synchronized(proxy.m_Book)
        {
            try {proxy.m_Book.wait();} catch (Exception e){}
        }
    }
    waitingWriters--; activeWriters++;
}

after(Writer writer, Proxy proxy):WriterOpen(writer, proxy){
    synchronized(proxy.m_Book){
        proxy.m_Book.Write();
    }
}

after(Writer writer, Proxy proxy):WriterOpen(writer, proxy){
    activeWriters--;
    if(waitingReaders != 0) preferWriters = false;
    synchronized(proxy.m_Book)
    {
        proxy.m_Book.notifyAll();
    }
}

Figure 6. StrategyAspect aspect

350
For the Reader objects the access policies to the Book will be implemented in the advice before(Reader reader, Proxy proxy):ReaderOpen(reader, proxy) run in the moment specified by the pointcut ReaderOpen. This is defined by the pointcut ReaderOpen(Reader reader, Proxy proxy): call(* *,Open()) ? this(reader) ? target(proxy), identifies the call of Open() of the Proxy type object (identified by target) in one of the Reader objects (identified by this) (Figure 7).

![Diagram](image)

Figure 7. The interaction of the Reader-Proxy-Book objects

As a result of weaving the aspect StrategyAspect with the base classes we will obtain Java compatible classes. This is due to the AJDT plug-in (AspectJ Development Tools) of the Eclipse IDE.

**Conclusions**

We studied the problem of aspect-oriented implementation of Readers-Writers concurrent problem. Identification of functional and non-functional requirements right from the beginning of the development can offer a chance for modularity free of the tyranny of dominant decomposition and reusability anomalies.
We proposed the arch-pattern design for the Readers-Writers problem which offers a certain flexibility in description of client behavior, but is not enough for reaching the primary goal: separating the non-functional requirements in independent units.

The aspect-oriented approach and the new arch-pattern design have the advantage that they offer more generic mechanisms for describing interactions between objects, in comparison to patterns described in a strictly object-oriented context. But the implementation of this for aspect-oriented programming is not a proper solution because the principle of reusability cannot be guaranteed for aspects.

References


352
Arch-pattern based design and aspect-oriented implementation of ...


Dumitru Ciorbă, Victor Beșiu, Received May 21, 2007
Anthony Chronopoulos, Andrei Poștaru, Revised October 16, 2007

Dumitru Ciorbă, Victor Beșiu, Andrei Poștaru
Technical University of Moldova
Automation & Information Technology Department
str. Studentilor, 7/3, corp 3, 504 Chisinau, MD-2068
Phone: (+373 22) 49 70 18
E-mail: vbesliu@yahoo.com, besliu@mail.utm.md
E-mail: dciorb@yahoo.com

Dr. Anthony T. Chronopoulos
University of Texas at San Antonio
Department of Computer Science
6900 North Loop 1604 West
San Antonio, TX 78249
Phone: (210)458 – 7214
Fax: (210)458 – 4437
E-mail: atc@cs.utsa.edu