Mini-Proceedings of the Master Seminar

Advanced Software Engineering Topics: Aspect Oriented Programming

Summer Semester 2006

Editors
Dr. Patrik Fuhrer and Prof. Jacques Pasquier

Organized by the

UNIVERSITÉ DE FRIBOURG SUISSE
UNIVERSITÄT FREIBURG SCHWEIZ

Software Engineering Group
Department of Informatics
University of Fribourg (Switzerland)

Foreword

Every summer semester, the Software Engineering Group of the University of Fribourg proposes a seminar to his master level students on a chosen "advanced software engineering" topic. The 2006 edition of the seminar is devoted to Aspect Oriented Programming (AOP).

In object-oriented programming (OOP), one usually captures the main concerns of a problem (e.g. for a banking application, money withdrawal at an ATM, funds transfer,...) and then encapsulates them into classes. The problem is that some "accessory", non-business operations are needed (i.e. logging, authentication, caching, transactions,...). These operations are shared by several classes of a system, and each of them has to code their firing.

The main idea behind AOP is to consider these operations as *crosscutting aspects* and to group them into separate modules, which are automatically called by the system. This new approach helps building clean software architectures and completes the toolbox of software engineers, already containing design patterns, frameworks, refactoring, unit testing,... AOP is particularly well suited for implementing middleware frameworks which are typically responsible for these crosscutting concerns. Indeed, the additional functionnalities can be transparently injected (or weaved) at compile, load or run time into the core business code which (at the source level) remains clean and unchanged.

The goal of this seminar is to familiarize its participants to this new AOP paradigm, to its advantages through examples and to its possible implementations. To achieve this goal, the book *AspectJ in Action - Practical Aspect-Oriented Programming* by Ramnivas Laddad is a very good starting point. Thus the main part of the seminar focuses on AspectJ, but the last presentation introduces two other main AOP frameworks, Spring AOP and JBoss AOP, and compares them with AspectJ.

The participants did a great job learning this new exciting technology. They also spent a fair amount of time programming and debugging their own AOP example applications, which they formally presented and explained to their colleagues. We are thankful for their efforts and hope they will consider using AOP in their future software development projects.

Patrik Fuhrer and Jacques Pasquier

Table of Contents

Implementing Thread Safety

Lawrence Michel
University of Fribourg
Department of Informatics
Bd de Pérolles 90
CH-1700 Fribourg
lawrence.michel@unifr.ch

Abstract This paper is a synthesis of a seminar course given by the Software Engineering Group at the University of Fribourg, Switzerland. It focuses on *Aspect Oriented Programming* and shows with the use of concrete examples how the *AspectJ* language may address *Thread-Safety* issues in Java Programs.

Keywords: AOP, AspectJ, Aspect, Java, Policy enforcement, Thread, Thread-safety, UI-responsiveness, Read/Write lock.

1. Introduction

Along the last few decades, we could assist to an impressive development of programming languages and techniques that addresses a large spectrum of problems. Business processes, which activities relies on computer-based solution, are a good example. At the beginning, computer were used within businesses to address very specific problems. The user was commonly a computer specialist. Nowadays, due to the explosion of internet, having access to a plethora of information is much easier for everyone. And because computer (and software) designing has drastically improved within this last years, (almost) every problem can be computationally solved: managing our bank account while being at home, pay using credit card electronically, use software to visualize statistical data, and so on.
Programming language has evolved in a way that we, as programmers, don't have to take care anymore about how we should implement a software solution using machine-instructions. *Object Oriented Programming* (OOP) enabled us to think the implementation as a combination of specific objects, which leads to *object abstraction*.
Aspect Oriented Programming (AOP) focuses on concern abstraction, which will be our main discussion topic.
This paper is part of the Master seminar on *Aspect Oriented Programming* organized by the Software Engineering Group of the University of Fribourg. We will first begin with some introductory AOP basics, then focus on the *AspectJ* implementation by understanding how it may address the *Thread-safety implementation* problematic.

2. AOP fundamentals and AspectJ

In this chapter, we will briefly introduce some fundamental basics of *Aspect Oriented Programming* followed by it's *AspectJ* implementation. The goal of this introduction is to keep some fundamental aspects in mind while reading the remaining chapters of this paper.

2.1. Aspect Oriented Programming

Aspect Oriented Programming is a programming methodology which enables a clear separation of concerns in software applications.
A *concern* in mean of software development can be understood as "*a specific requirement [...] that must be addressed in order to satisfy the overall system goal*"[3].
That is, *concerns* can be subdivided in two subgroups:

Core concerns Specify the main functionality of a software module, such as account withdrawing or a button click behaviour. We can define this as the *business logic* of our implementation.

Crosscutting concerns Specify the peripheral requirements that cross multiple modules, such as transaction, threading or authentication.

The idea is to be able to model, or visualize our goal system as a combination of core and crosscutting concerns in a multi-dimensional space implementation rather than a mixture of all of them along a one-dimensional space implementation. Because concrete software are one-dimension implementation, all concerns are then mapped into it. (see Figure 1).

The *AOP methodology* proposes to follow 3 distinctive steps when developing a software system:

1. Distinguish all program requirements and identify each of them as either core or crosscutting concerns. (*Aspectual decomposition*)
2. Implement each concern independently. (*Concern implementation*)
3. Specify recomposition rule, or *weaving* rule, in modularization units. (*Aspectual recomposition*)

That is, several implementation of *AOP* that fulfill these specifications can be found on the market[1], such as:

[1] An exhaustive list can be found from [6].

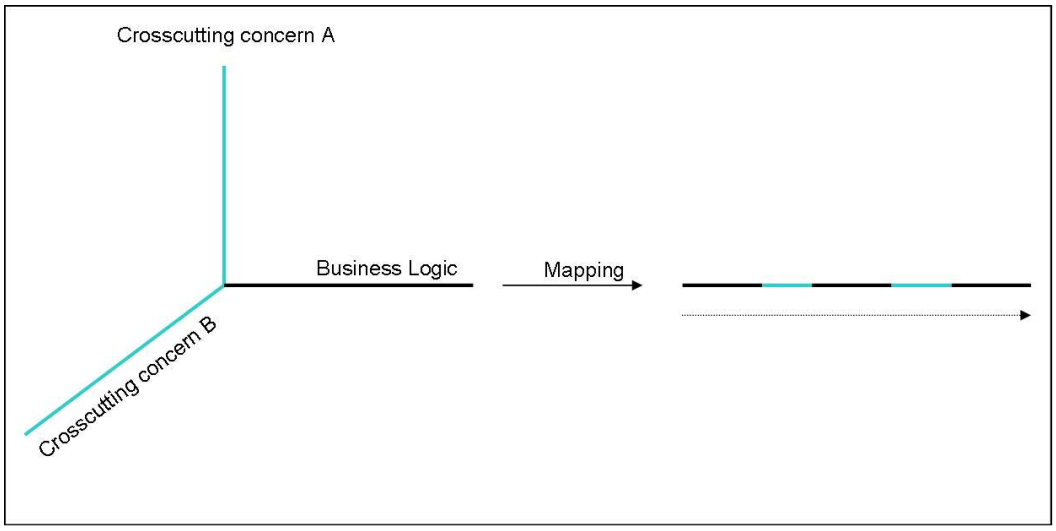

Figure 1.: Mapping from a multi-dimensional concern space into a one-dimensional implementation space.

- For C/++: AspectC++, XWeaver Project, FeatureC++;
- For PHP: PHPaspect, Aspect-Oriented PHP;
- For Java: JBoss AOP, AspectWerkz, AspectJ.

2.2. AspectJ

AspectJ is an implementation of *AOP*. It is an extension to the Java programming language. Core concerns, which implement the business logic, remain pure Java code. The mapping rules, called the *weaving* rules, are implemented within *crosscutting* concerns using a specific syntax and semantic. AspectJ distinguish several crosscutting elements, whereas:

Join point An identifiable point in the execution program.

Pointcut A program construct that selects join points.

```
execution(void Account.credit(float))
```

Advice The code to be executed (or *injected*) at a join point.

```
before() : execution(void Account.credit(float)) {
    System.out.printLn("About to perform credit operation");
}
```

Aspect The final class which encapsulates the global aspect code that expresses the weaving rules for crosscutting.

```
public aspect ExampleAspect {
    before() : execution(void Account.credit(float)) {
        System.out.printLn("About to perform credit operation");
    }
//(...)
}
```

As an example, Listing 1 is a basic pure Java implementation. This class simply outputs `Sir or Madam` in the Java console. This is our core concern. Let's imagine that our company's policy forces us to add a more polite manner to it, let's say `Dear Sir or Madam`, we will use an aspect implementation that fulfills this new requirement. Listing 2 is an aspect implementation that fits our new policy.
Line 1 of Listing 2 is the aspect declaration. We then find at line 4 the single pointcut declaration. It sets that all calls to the deliver method of class named `MessageCommunicator` require a specific behaviour. Line 7 sets the advice rules to be applied to the `deliverMessage` pointcut captured. Before executing the pointcut, print the string `Dear` .

```
public class MessageCommunicator {
    String msg = new String("Sir or Madam");
    public static void deliver() {
        System.out.print(msg);
    }
}
```

Listing 1: The `MessageCommunicator` class.

```
public aspect MannersAspect {

    pointcut deliverMessage() :
        call(* MessageCommunicator.deliver(..));


    before() : deliverMessage() {

        System.out.print("Dear ");
    }
}
```

Listing 2: An aspect implementing a good manner rule to be injected.

2.3. Summary

This section introduced the *AOP* paradigm and the *AspectJ*'s implementation. A more detailed introduction of *AspectJ* can be found in [3]. Software construction may be done by a clever mapping of core and crosscutting concerns, which leads to a higher global system understanding, less maintenance harassment and finally, to better productivity.
Now that we have gained a global overview of important concepts of *AOP* and AspectJ's concrete implementation, we can now jump to the next section to get a closer look on how *AOP* may be used when addressing several problematic aspects of business logic in a Java software.

3. Thread-Safety and AOP

Before we get through our AspectJ concrete examples, we will briefly go back to some useful notions of *threads*, *thread-safety* and *policy enforcement*. These are closely related to the case we will discuss.
We will then analyze how *AOP* may address *thread-safety* in software development. We will see how AspectJ's implementation of *AOP* may be applied to fulfil the *thread-safety* requirement of a software. We will focus on the Swing-related pattern, the UI-responsiveness issue and finally the read-write lock pattern.

3.1. General background

Before we get through our AspectJ concrete examples, The notions of threads, of implementing thread-safety and of policy enforcement are necessary to be refreshed in our knowledge base.

Threads

In mean of computer science, a *thread* is a part of a program which can be eligibly split from it and executed independently. Having multiple threads running in parallel in one process (a program currently being executed) is analogous to having multiple processes running in parallel in one computer. Multiple *threads* can share the same resource, such as an address space or a disk. The advantage of multi-threaded programs is clearly that it allows to operate faster by having the benefit from every free CPU time.
In a *multithreaded* software application, every *thread* can access every memory address within the process' address space. That is, there is no protection between concurrent threads[2]. Control mechanisms between concurrent *threads* must remain in the software implementation itself. Most of well known programming language propose a set of libraries which specifically address this issue.

[2]See [5] pp.81-100.

Implementing thread-safety

"*Thread-safety is a computer programming concept applicable in the context of multi-threaded programs.*"[7]. In other words, implementing *thread-safety* means maintaining correct system behavior when multiple threads share a common system state.
To determine if our piece of program is thread-safe is not easy. Several indicators need careful examination, such as if our thread may access global variables, disk files or allocating/freeing resources.

Policy enforcement

Basically, there are multiple ways to implement a software solution that fulfils our core requirements. Thereby, if we take a team of software programmers as an example, anyone might solve a problem in the way he has got best experienced. Unfortunately, in the scope of big and high modularized software design, this can lead to an exotic implementation of software modules. Because no crosscutting rules have been settled, the understanding and maintaining of such a system can get harder.
Introducing some standards rules that guides the way software modules have to be implemented are nowadays quite trendy. Design Patterns show good reusable implementation to solve a certain scope of software cases.

Policy enforcement is a mechanism for ensuring that system components follow certain programming practice, comply with specified rules, and meet the assumptions.[3]

In mean of *AOP*, applying standardization rules or policies tends to address much more specific particularities, such as controlling if the piece of code is correct or fits some side-requirements.[3]

3.2. Implementing thread-safety with AspectJ: Swing

Swing's single-thread rule

Java's Swing GUI library has been designed in a way that all access to its component states from another thread may be done using it's own event-dispatching thread. But to avoid being too restrictive when implementing a Swing based GUI, which may lead to too much coding harassment, the use of this functionality still remains as not mandatory. But in certain cases, especially when the UI thread may perform I/O or network operation, such as updating a table from a database, concurrent access to it's states can lead to instability, or even worse, a crash.
Swing's single-thread rule states: "Once a Swing component has been realized, all code that might affect or depend on the state of that component should be executed in the event-dispatching thread" [4].
That means, once the GUI components have been built (displayed or painted), all access to them should be done through the event-dispatcher using the `EventQueue.invokeLater()` and `EventQueue.invokeAndWait()` methods.

Thread-safety in Swing: A test problem

Let's examine our first test program (Fig.2). Listing 3 implements a small Swing user-interface which is composed of a table and a message dialog dedicated to require user input (button press). Listing 4 is an implementation of a logging aspect, which will help us to better understand our problematic. This aspect is dedicated to track all `javax.*` method calls and print out their signature and the current thread they are belonging to.
Now, let's see what happens when we execute our first test combined with its tracking aspect. We get output in Listing 5.
After reading this log file, we observe that all calls to the UI `Javax.*` methods are done within the same thread: The method is executed in thread `main`, with priority set to `5`, and has been called from the thread `main`[5].
This implementation doesn't comply with Swing's thread-safety rule: methods calls at lines 13, 17 and 20 are methods accessing Swing UI components. But these are not done using the Event-Dispatching thread, but directly from the caller thread. Next section will now show us a conventional way to fix our problem.

[3] Refer to chapter 6 of [3] for various AspectJ examples of *policy enforcement*.
[4] A more complete documentation about Swing's thread rule can be found in chapter 9 of [2].
[5] See Sun's Java documentation about thread at [4].

Figure 2.: The Swing application.

```
import java.awt.*; import javax.swing.*; import javax.swing.table.*;

public class Test {
    public static void main(String[] args) {
    JFrame appFrame = new JFrame();
    appFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    DefaultTableModel tableModel = new DefaultTableModel(4,2);
    JTable table = new JTable(tableModel);
    appFrame.getContentPane().add(table);
    appFrame.pack();
    appFrame.setVisible(true);
    String value = "[0,0]";
    tableModel.setValueAt(value, 0, 0);

    JOptionPane.showMessageDialog(appFrame,
                      "Press OK to continue");
    int rowCount = tableModel.getRowCount();

    System.out.println("Row count = " + rowCount);
    Color gridColor = table.getGridColor();

    System.out.println("Grid color = " + gridColor);
    }
}
```

Listing 3: The basic Swing test class.

```
import java.awt.EventQueue;

public aspect LogUIActivitiesAspect {
    pointcut uiActivities()
    : call(* javax..*+.*(..));

    before() : uiActivities() {
    System.out.println("Executing:\n\t"
                + thisJoinPointStaticPart.getSignature()
                + "\n\t"
                + Thread.currentThread() + "\n");
    }
}
```

Listing 4: A universal aspect that logs every method calls.

```
Executing:
    void javax.swing.JFrame.setDefaultCloseOperation(int)
    Thread[main,5,main]

Executing:
    Container javax.swing.JFrame.getContentPane()
    Thread[main,5,main]

Executing:
    void javax.swing.JFrame.pack()
    Thread[main,5,main]

Executing:
    void javax.swing.JFrame.setVisible(boolean)
    Thread[main,5,main]

Executing:
    void javax.swing.table.DefaultTableModel.setValueAt(Object, int, int)
    Thread[main,5,main]

Executing:
    void javax.swing.JOptionPane.showMessageDialog(Component, Object)
    Thread[main,5,main]

Executing:
    int javax.swing.table.DefaultTableModel.getRowCount()
    Thread[main,5,main]

Row count = 4 Executing:
    Color javax.swing.JTable.getGridColor()
    Thread[main,5,main]

Grid color = javax.swing.plaf.ColorUIResource[r=122,g=138,b=153]
```

Listing 5: The output of the standard Swing implementation.

Thread-safety in Swing: A conventional solution

A conventional solution to fix this problem is to encapsulate each method call to UI components within an anonymous `Runnable` classes. Then, we call either the `EventQueue.invokeLater()` or the `EventQueue.invokeAndWait()` if such methods require to be run synchronously or asynchronously. Listing 6 partially shows the resulting correction of our base implementation given from Listing 5.
Let's get a bit into details of Listing 6:

- Before line 4, all method calls could be executed within the same thread. Since the AWT thread is not accessing the components, the `main` Thread is the only one updating the UI component state.
- The call at line 12 requires to be used synchronously. The message dialog first needs to perform its confirmation step before giving the hand back to the next methods. The `EventQueue.invokeAndWait()` method is applied.
- At line 25, the call should be executed synchronously, because it's return value will be needed to update a global variable.
- And finally, the call at line 36 must behave equally, because it's object return value will be required to update a variable.

Now, after having executed it with our logging aspect, we get the output shown in Listing 7.
At this point, we see precisely that all access to UI methods are done within the `AWT-EventQueue` thread. But this way to solve this problem might not be very attractive when implementing it in bigger UIs. We observe as well that all these UI methods are run synchronously, which might not be necessary in certain circumstances.

Thread-safety in Swing: The AspectJ solution

Now let's treat our case with an AspectJ-based solution. Our aim is to get our basic UI implementation of Listing 3 get weaved in a manner that all it's methods accessing UI component states are done within the `Event-dispatcher` Thread.
The *AOP* methodology detailed in section 2.1 showed us the steps we might respect to implement our program in an aspect way. Let's apply it to here:

```
import java.awt.*; import javax.swing.*; import javax.swing.table.*;

public class Test {
    (...) Same as inListing 3 until line13


    EventQueue.invokeLater(new Runnable() {
        public void run() {
            tableModel.setValueAt(value, 0, 0);
        }
        });
    try {
        EventQueue.invokeAndWait(new Runnable() {
            public void run() {
            JOptionPane.showMessageDialog(appFrame,"Press OK to continue");



            }
        });
    } catch (Exception ex) {
        (...)
    }
    final int[] rowCountValueArray = new int[1];
    try {
        EventQueue.invokeAndWait(new Runnable() {
            public void run() {
            rowCountValueArray[0] = tableModel.getRowCount();

            }
        });
    } catch (Exception ex) {
        (...)
    }
    (...)
    try {
        EventQueue.invokeAndWait(new Runnable() {
            public void run() {
            gridColorValueArray[0] = table.getGridColor();

            }
        });
    } catch (Exception ex) {
        (...)
    }
    (...)
    }
}
```

Listing 6: The test class implementing the Swing conventional solution.

```
Executing:
    void javax.swing.JFrame.setDefaultCloseOperation(int)
    Thread[main,5,main]

Executing:
    Container javax.swing.JFrame.getContentPane()
    Thread[main,5,main]

Executing:
    void javax.swing.JFrame.pack()
    Thread[main,5,main]

Executing:
    void javax.swing.JFrame.setVisible(boolean)
    Thread[main,5,main]

Executing:
    void javax.swing.table.DefaultTableModel.setValueAt(Object, int, int)
    Thread[AWT-EventQueue-0,6,main]

Executing:
    void javax.swing.JOptionPane.showMessageDialog(Component, Object)
    Thread[AWT-EventQueue-0,6,main]

Executing:
    int javax.swing.table.DefaultTableModel.getRowCount()
    Thread[AWT-EventQueue-0,6,main]

Row count = 4 Executing:
    Color javax.swing.JTable.getGridColor()
    Thread[AWT-EventQueue-0,6,main]

Grid color = javax.swing.plaf.ColorUIResource[r=122,g=138,b=153]
```

Listing 7: The output printed from Swing's conventional solution.

Step 1: Aspectual decomposition Our program can be decomposed in 3 abstracted concerns:

1. The *core concern* states that we have to create a simple GUI in Swing with a table in its main frame, a `messageDialog` requiring user input, then some table being updated.
2. The first *crosscutting concern* should track all UI component method's access, and print out the thread name and priority it is belonging to (Listing 4).
3. The second *crosscutting concern* will capture all UI method calls that interfere with the Swings thread-safety rule and make the necessary fixes to them.

Step 2: Concern implementation

1. The core concern is found in (Listing 3).
2. The first crosscutting concern remains the same aspect we used until now (Listing 4).
3. The second crosscutting concern, which is the key solution of our problem, can be found in (Listing 9). This aspect requires import of a `pattern.worker` package, which offers a class that enables method encapsulation into anonymous `runnable` classes. We observe that this concern has been declared as abstract. The aspect of Listing 8 extends this abstract aspect and implements the `uiMethodCalls()` pointcut. That is, now have a general solution.

Step 3: Aspectual recomposition Compile the whole stuff with AspectJ compiler, which will process the concrete weaving rules stated in the crosscutting concerns.

The output we get after running our aspect-based test program clearly shows the same characteristics as the one of our previous conventional implementation (see Listing 10). We notice that all calls to UI methods are processed synchronously. If we wanted to have the aspect handle synchronous method calls separately from one running asynchronously, we will simply need to refine our pointcut declarations, then add their dedicated advices.

Discussion

There are still much room for further improvements within the examples we have studied. We could still add specific advices concerning UI methods that do not require to be run synchronously. Furthermore, the case of exception handling has not been presented here. This is an important case that every Swing

```
public aspect DefaultSwingThreadSafetyAspect
    extends SwingThreadSafetyAspect {
    pointcut viewMethodCalls()
    : call(* javax..JComponent+.*(..));

    pointcut modelMethodCalls()
    : call(* javax..*Model+.*(..))
    || call(* javax.swing.text.Document+.*(..));

    pointcut uiMethodCalls()
    : viewMethodCalls() || modelMethodCalls();
}
```

Listing 8: the aspect extending the abstract thread-safety aspect.

```
import java.awt.*; import java.util.*; import javax.swing.*; import pattern.worker.*
;

public abstract aspect SwingThreadSafetyAspect {
    abstract pointcut uiMethodCalls();

    pointcut threadSafeCalls()
    : call(void JComponent.revalidate())
    || call(void JComponent.repaint(..))
    || call(void add*Listener(EventListener))
    || call(void remove*Listener(EventListener));

    pointcut excludedJoinpoints()
    : threadSafeCalls()
    || within(SwingThreadSafetyAspect)
    || if(EventQueue.isDispatchThread());

    pointcut routedMethods()
    : uiMethodCalls() && !excludedJoinpoints();

    Object around() : routedMethods() {
    RunnableWithReturn worker = new  RunnableWithReturn()  {
        public void run() {
            returnValue = proceed();
        }};
    try {
        EventQueue.invokeAndWait(worker);
    } catch (Exception ex) {
        // ... log exception
        return null;
    }
    return worker.getReturnValue();
    }
}
```

Listing 9: The abstract thread-safety aspect.

```
Executing:
    void javax.swing.JFrame.setDefaultCloseOperation(int)
    Thread[main,5,main]

Executing:
    Container javax.swing.JFrame.getContentPane()
    Thread[main,5,main]

Executing:
    void javax.swing.JFrame.pack()
    Thread[main,5,main]

Executing:
    void javax.swing.JFrame.setVisible(boolean)
    Thread[main,5,main]

Executing operation synchronously Executing:
    void javax.swing.table.DefaultTableModel.setValueAt(Object, int, int)
    Thread[AWT-EventQueue-0,6,main]

Executing operation synchronously Executing:
    void javax.swing.JOptionPane.showMessageDialog(Component, Object)
    Thread[AWT-EventQueue-0,6,main]

Executing operation synchronously Executing:
    int javax.swing.table.DefaultTableModel.getRowCount()
    Thread[AWT-EventQueue-0,6,main]

Row count = 4 Executing operation synchronously Executing:
    Color javax.swing.JTable.getGridColor()
    Thread[AWT-EventQueue-0,6,main]

Grid color = javax.swing.plaf.ColorUIResource[r=122,g=138,b=153]
```

Listing 10: The output printed from the AspectJ-based implementation.

GUI application using AspectJ should consider, because exception thrown from a method encapsulated in a `invokeLater()` method might not be promptly caught by the called thread. That is, the solution proposed by [3] is currently to surround the `proceed` method in the aspect into a try-catch.
The use of aspects in the case of Swing based applications can bring much efficiency and improvement in a manner of time. However, applying this methodology can become quite cumbersome when the user-interface gets more complex, which is quite common on most Java-based UI applications. On the other hand, we believe that Swing developers may propose beside their Swing Package a set of AspectJ implementations where Swing's basic rules, exception handling rules - and probably much more - are stated. This would avoid that every GUI designer develops it's own aspects to address common issues.

3.3. Improving responsiveness of UI applications

In our last section, we used a Swing-based application to demonstrate how aspects can be used for policy enforcement. We will now have a look on a very common issue of GUI implementation.

Improving responsiveness in Swing: A test problem

Let's examine our second test program (Fig.3). Listing 11 implements a small Swing user-interface which is composed of a button. When the user clicks on it, it will start a waiting process which simulates, for example, a long I/O or a database query.
This issue is relatively common when implementing GUI-based applications. When we click the button, the waiting process locks the whole GUI until it has finished. This is sometimes not much aesthetic from the user point of view, because the global application UI gets freezed, and depending on how long the waiting process occurs, the user can definitely conclude that the program is getting unstable. One possible solution to that problem is to display a progress bar allowing the user to know the actual progression of the process. On the other hand, there are certain cases that a process should not necessary lock the whole application, because it might not been designed to alter other GUI's states.

Improving responsiveness in Swing: The AspectJ solution

Let's have a look to the AspectJ way to address this issue.
The abstract aspect in Listing 12 captures all operations that may not be run in the current thread.

Figure 3.: The Swing sendmail application.

```
import java.awt.event.ActionEvent; import
java.awt.event.ActionListener;

import javax.swing.JButton; import javax.swing.JFrame;

public class TestResponsiveness {
    public static void main(String[] args) {
    JFrame appFrame = new JFrame();
    appFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JButton sendEmailButton = new JButton("Send Emails");
    sendEmailButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            sendEmails();
        }
        });
    appFrame.getContentPane().add(sendEmailButton);
    appFrame.pack();
    appFrame.setVisible(true);
    }

    private static void sendEmails()  {
    try {
        // simulate long execution...
        Thread.sleep(20000);
    } catch (InterruptedException ex) {
    }
    }
}
```

Listing 11: The main test class.

```
import java.awt.EventQueue;

public abstract aspect
AsynchronousExecutionAspect {
    public abstract pointcut asyncOperations();

    void around() : asyncOperations()

    && if(EventQueue.isDispatchThread()) {
        Runnable worker = new Runnable() {
            public void run() {
            proceed();
            }
        };
         Thread asyncExecutionThread = new Thread(worker);
         asyncExecutionThread.start();
    }
}
```

Listing 12: The abstract aspect that addresses the Read/Write lock pattern.

```
public aspect TestAsynchronousExecutionAspect
    extends AsynchronousExecutionAspect {
     public pointcut asyncOperations()
     :  call(void sendEmails());
}
```

Listing 13: The aspect that extends the Read/Write Lock aspect.

The advice at line 7 encapsulates these captured operations in a new anonymous runnable object. This new object will then be given a new dedicated thread from where it can be processed. Because it is now working in a separate thread away of the caller one, it won't lock the overall GUI anymore.
Listing 13 is it's extension aspect that implements the concrete pointcut elements (in this case, all operations with the profile similar as `void sendEmails()`) and set them to belong to the `asyncOperations` pointcut.

Discussion

The use of aspects in the case of UI responsiveness is quite easy. But there is the need to clearly distinguish all aspects of our UI based application behaviors. For example, the UI designer should be able to list all methods within his UI implementation that my not require to be run in their own thread. Furthermore, he should pay attention not to simply give each called method it's own thread, as it could lead to have too many threads running at the same time, which may finally provoke program instability.

3.4. Implementing a reusable Read-Write lock pattern

Let's finally have a closer look on the Read-Write Lock Pattern. Basically, the idea behind this pattern is that any number of readers could be simultaneously reading the state of an object as long as there are no thread modifying the state at the same time.
The solution we discuss here is based upon the concurrency utility library by Doug Lea[6]. We will implement this pattern using the conventional way, then we will see how AspectJ may address this issue.

The Read-Write lock pattern: A conventional solution

The conventional solution requires adding some additional code into the account class found in Listing 14. The example class presented here is a simple implementation of a banking account. A fundamental aspect of banking accounts is that beside all transactional requirements it has, the notion of data integrity is an important issue: there must be such a mechanism that manages data access properly.
The `Account` class listed here proposes 4 distinctive methods: `credit()`, `debit()`, `getBalance()` and `setBalance()`. For each of those methods, we make a `writeLock()` or `readLock()` access to either `acquire()` or `release()` the lock variable.

[6]Documentation can be found at [1].

```
import EDU.oswego.cs.dl.util.concurrent.*;

public abstract class Account {
    private float _balance;
    private int _accountNumber;

    private ReadWriteLock _lock
    = new ReentrantWriterPreferenceReadWriteLock();

    public Account(int accountNumber) {
    _accountNumber = accountNumber;
    }

    public void credit(float amount) {
    try {
         _lock.writeLock().acquire();
        setBalance(getBalance() + amount);
    } catch (InterruptedException ex) {
        throw new RuntimeException(ex);
    } finally {
         _lock.writeLock().release();
    }
    }

    public void debit(float amount)
    throws InsufficientBalanceException {
    try {
         _lock.writeLock().acquire();
        float balance = getBalance();
        if (balance < amount) {
        throw new InsufficientBalanceException(
                    "Total balance not sufficient");
        } else {
        setBalance(balance - amount);
        }
    } catch (InterruptedException ex) {
        throw new RuntimeException(ex);
    } finally {
         _lock.writeLock().release();
    }
    }

    public float getBalance() {
    try {
         _lock.readLock().acquire();
        return _balance;
    } catch (InterruptedException ex) {
        throw new RuntimeException(ex);
    } finally {
         _lock.readLock().release();
    }
    }

    public void setBalance(float balance) {
    try {
         _lock.writeLock().acquire();
        _balance = balance;
    } catch (InterruptedException ex) {
        throw new RuntimeException(ex);
    } finally {
         _lock.writeLock().release();
    }
    }
}
```

Listing 14: The `Account` class.

```
public class Test {
    public static void main(String[] args)
    throws InsufficientBalanceException {
    SavingsAccount account = new SavingsAccount(12456);
    account.credit(100);
    account.debit(50);
    }
}
```

Listing 15: The test class.

```
public abstract class Account {
    private float _balance;
    private int _accountNumber;

    public Account(int accountNumber) {
    _accountNumber = accountNumber;
    }

    public void credit(float amount) {
    setBalance(getBalance() + amount);
    }

    public void debit(float amount)
    throws InsufficientBalanceException {
    float balance = getBalance();
    if (balance < amount) {
        throw new InsufficientBalanceException(
               "Total balance not sufficient");
    } else {
        setBalance(balance - amount);
    }
    }

    public float getBalance() {
    return _balance;
    }

    public void setBalance(float balance) {
    _balance = balance;
    }
}
```

Listing 16: The business logic implementing the `Account` class.

The Read-Write lock pattern: The AspectJ solution

We clearly saw that the conventional solution contains a recurrent lock pattern surrounding each of the class' methods. We can then easily get through an aspectJ-based solution.
First of all, we keep from Listing 14 only the core implementation of the `Account` class. This leads to the abstract aspect in Listing 16. We implement the AspectJ-based rules. The idea is to inject the specific lock pattern in each of the class methods, depending if it is processing a variable as read or write. Because each method has first to acquire a lock, we inject the necessary lock-acquire code `before()` processing the method, and then we inject the lock-release code `after()` the method may have finished. Listing 17 is the abstract implementation of the new aspect
Notice that Listing 18 extends it by defining the concrete pointcuts where the advice code might be injected. Furthermore, we used the `perthis()` association specification. It will associate an aspect instance with each worker object that match the `read()` or `write()` methods defined in the concrete aspect[7].

Discussion

The Read/Write lock issue is another good example to show how effective the use of AspectJ-based solution can become. We clearly separated the core concern from it's crosscutting concern. Furthermore, the solution can be easily reused within other kind of class requiring a lock pattern.

4. Conclusion

We have seen in this paper how *AOP* addresses *Thread-Safety* requirements in Java programs. The aim for having these crosscutting concerns being implemented separately from the business, or *core* concern, is clearly to bring the specific implementation of our aspects to maximum understandability, which leads to better maintenance, and reusability. We have seen in the Swing's policy rule enforcement that implementing aspects should be better done from the Swing's package developer rather from the application developer. On the other hand, the UI responsiveness example and the Read/Write lock pattern lets the application developer benefit of the great advantages that *AOP* offers.

[7] Refer to chapter 4 section 4.3.2 of [3] for further information on aspect association

```
import EDU.oswego.cs.dl.util.concurrent.*;

public abstract aspect ReadWriteLockSynchronizationAspect
    perthis(readOperations() || writeOperations()) {
    public abstract pointcut readOperations();

    public abstract pointcut writeOperations();

    private ReadWriteLock _lock
    = new ReentrantWriterPreferenceReadWriteLock();

    before() : readOperations() {
     _lock.readLock().acquire();
    }

    after() : readOperations() {
     _lock.readLock().release();
    }

    before() : writeOperations() {
     _lock.writeLock().acquire();
    }

    after() : writeOperations() {
     _lock.writeLock().release();
    }

    static aspect SoftenInterruptedException {
    declare soft : InterruptedException :
        call(void Sync.acquire());
    }
}
```

Listing 17: The abstract aspect that injects the Read/Write lock pattern.

```
aspect BankingSynchronizationAspect
    extends ReadWriteLockSynchronizationAspect {
    public pointcut readOperations()
    : execution(* Account.get*(..))
    || execution(* Account.toString(..));

    public pointcut writeOperations()
    : execution(* Account.*(..))
    && !readOperations();
}
```

Listing 18: The aspect that extends the Read/Write lock aspect.

AOP is nowadays a trend in the programming era. Most famous languages offer the aspect extension to them. This is clearly an advantage from a researcher point of view. But an important issue may really be taken in consideration: **cross plateform abilty**. That is an important issue that could better convince the business to really engage themselves in it. In order to do it, the *AOP* should be hosted at a standardization consortium, such as the World wide Web Consortium (see http://www.w3.org). Furthermore, an XML-based *AOP* implementation may be the right path to follow to have it clearly be able to cross platforms[8].

[8] See http://aspectwerkz.codehaus.org/

References

[1] Doug Lea. Doug Lea's Homepage. [online]. http://gee.cs.oswego.edu/) (accessed June 12, 2006).

[2] Hans Muller and Kathy Walrath. Threads and Swing. [online]. http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html) (accessed June 7, 2006).

[3] R. Laddad. *AspectJ in Action, Practical Aspect-Oriented Programming*. Manning, 2003.

[4] Sun Microsystem. Thread library in Java. [online]. http://java.sun.com/j2se/1.3/docs/api/java/lang/Thread.html) (accessed June 8, 2006).

[5] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.

[6] Wikipedia. Aspect-oriented programming. [online]. http://en.wikipedia.org/wiki/Aspect-oriented_programming (accessed June 5, 2006).

[7] Wikipedia. Thread-safe. [online]. http://en.wikipedia.org/wiki/Thread_safety) (accessed June 5, 2006).

Authentication and Authorization

Pascal Bruegger
University of Fribourg
Department of Informatics
Bd de Pérolles 90
CH-1700 Fribourg
pascal.bruegger@unifr.ch

Abstract Aspect Oriented Programing (AOP) is a relatively young methodology. It uses new concepts in programing. The Object Oriented Programing (OOP) has introduced object abstraction years ago. Now with AOP, we talk about concern abstraction.
This paper is one part of a Master course seminar on AOP organized by the Software Engineering Research Group of the University of Fribourg. It describes one possible application of this methodology. We focus here on the *authorization* and *authentication* using AspectJ-based solutions.
To illustrate the discussion, we will progressively develop a banking system. We start with a basic account management without any access control. We'll end up with a program using authentication and authorization solution.

Keywords: AOP, AspectJ, Java, JAAS, Authentication, Authorization.

1. Introduction

Aspect Oriented Programing (AOP) is a relatively young methodology. It uses new concepts in programing. The Object Oriented Programing (OOP) has introduced object abstraction years ago. Now with AOP, we talk about concern abstraction.
This paper is one part of a Master course seminar on AOP organized by the Software Engineering Research Group of the University of Fribourg. It describes one possible application of this methodology.
We focus here on the *authorization* and *authentication* using AspectJ-based solutions. The main readings for this seminar is the book of R. Laddad, *AspectJ in action* [3] and the article from the same author *Simplify your logging with AspectJ* [2].
To illustrate the discussion, we will progressively develop a banking system. We start with a basic account management without any access control. We'll end up with a program using authentication and authorization solution.
The version of AOP chosen is the *AspectJ 1.1*. The example proposed and discussed in the paper are taken from the book *AspectJ In Action* [3]. The source codes are compiled and executed with Eclipse 3.1. The AspectJ compiler v. 1.1.0 comes from Eclipse.org web site [1]. The Java runtime is the 1.5 [5].

2. Problem overview

Nowadays, modern software systems must take the security into consideration. This notion includes many components like authentication, authorization, web site attacks protection and cryptography.
We describe *authentication* as the process verifying that you are who you pretend to be. On the other hand the *authorization* is the process of checking the permissions that an authenticated user has to access certain resources.
The question now is how can we include good security control in a software without making the development and evolution too complex. If each class of the system contains some security checks, it becomes quickly difficult to properly handle the development. Moreover, any extension of the system or any modification including security will become a nightmare. We will see that AspectJ can offer elegant solutions to that problem. The solutions proposed in the examples are not purely AspectJ-based, they use existing security APIs.

3. Existing solutions

Modern security APIs have been developed since security became a crucial issue in software engineering. The aim of those API's is to abstract the underlying mechanism of control. The separation of the access control configuration from the core code make any changes less invasive.
JAAS, for instance, proposes the necessary methods avoiding any invasive control code into the core or business codes. Some functionality will be presented and used in the examples. They consist of the followings:

- Authentication
  - `LoginContext` object.
  - Callback Handlers presenting the login challenge to the user.
  - Login configuration files that allow to modify the configuration without changing the source code.
- Authorization
  - Subject class which encapsulates the information about a single entity, such as identification and credentials. It offers a method `doAsPrivileged()`.
  - Action object which encapsulates all methods that need to be checked. The action object is executed on behalf of the authenticated subject using the `doAsPrivileged()` method.
  - The check access performed by the calling the `AccessController.checkPermission()`.
  - Policy file which contains the subject permissions. This file is indirectly used by the `AccessController.checkPermission()` to grant access to the subject.

The main problem with JAAS is that all modules of your system which need control access will contain calls to its methods.
EJB framework proposes an other solution in order to avoid this spread of calls all over the application. It handles the authorization in much more modularized way. The security attributes are defined in the deployment descriptor files.
Unfortunately, EJB framework is not a solution for all type of applications and therefore, we come back to the starting point: we need more modularization. AspectJ has solutions for such situations.

Note
It is supposed in this paper that the reader has the necessary knowledge of JAAS API. Not all implementation details will be explained in the different descriptions. Information about JAAS is available on the web page:
http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/spec/.

---

4. Simple Banking example

To illustrate what AspectJ can do for our problem, we will implement a simple Banking system as testbed. The example shows at first a conventional solution and then AspectJ-based solution. It helps to see the advantage of the second solution.

4.1. Account implementation

First of all, we have the definition of an account and operations which can be done on this account. The main application will make these operations. It contains the following elements:

- An interface `Account` which defines the basic public method on the account.
- A class `AccountImpl` which implements the Account.
- A class `InsufficientBalanceException` which identifies insufficient balance.
- A class `InterAccountTransferSystem` containing one method for transferring founds from an account to another.
- A class `Test`.

```
package banking;

public interface Account {
    public int getAccountNumber();
    public void credit(float amount);
    public void debit(float amount)
       throws InsufficientBalanceException;
    public float getBalance();
}
```

Listing 1: Account.java

```
package banking;

public class AccountSimpleImpl implements Account {
    private int _accountNumber;
    private float _balance;

    public AccountSimpleImpl(int accountNumber) {
      _accountNumber = accountNumber;
    }

    public int getAccountNumber() {
      return _accountNumber;
    }

    public void credit(float amount) {
```

```
16      _balance = _balance + amount;
    }
18
    public void debit(float amount)
20      throws InsufficientBalanceException {
      if (_balance < amount) {
22       throw new InsufficientBalanceException(
       "Total balance not sufficient");
24    } else {
      _balance = _balance - amount;
26    }
    }
28
    public float getBalance() {
30     return _balance;
    }
32 }
```

Listing 2: AccountSimpleImpl.java

```
package banking;
2
public class InsufficientBalanceException extends Exception {
4    public InsufficientBalanceException(String message) {
       super(message);
6    }
}
```

Listing 3: InsufficientBalanceException.java

```
package banking;
2
public class InterAccountTransferSystem {
4    public static void transfer(Account from, Account to,
                                 float amount)
6       throws InsufficientBalanceException {
       to.credit(amount);
8      from.debit(amount);
    }
10 }
```

Listing 4: InterAccountTransferSystem.java

```
package banking;
2
public class Test {
4    public static void main(String[] args) throws Exception {
      Account account1 = new AccountSimpleImpl(1);
6     Account account2 = new AccountSimpleImpl(2);
      account1.credit(300);
8     account1.debit(200);

10    InterAccountTransferSystem.transfer(account1,
                                          account2, 100);
12    InterAccountTransferSystem.transfer(account1,
                                          account2, 100);
14   }
}
```

Listing 5: Test.java

We can see in this example that there is no authentication mechanism. When the Test program is executed, we have an `InsufficientBalanceException` thrown. The next step is to implement a basic logging aspect to help us understanding the activities taking place. To do this, we extend the `IndentedLogging` aspect presented in the chapter 5 of the book *AspectJ in action* [3].

```
package banking;
2
import org.aspectj.lang.*;
4 import logging.*;

6 public aspect AuthLogging extends IndentedLogging {
    declare precedence: AuthLogging, *;
8
    public pointcut accountActivities()
```

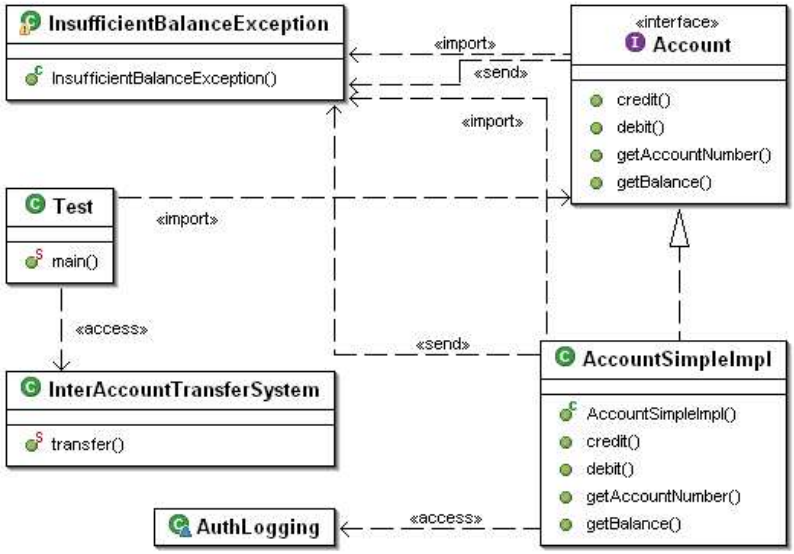

Figure 1.: Banking system class diagram

```
10      : execution(public * Account.*(..))
        || execution(public * InterAccountTransferSystem.*(..));
12
    public pointcut loggedOperations()
14    : accountActivities();

16  before() : loggedOperations() {
       Signature sig = thisJoinPointStaticPart.getSignature();
18     System.out.println("<" + sig.getName() + ">");
    }
20 }
```

Listing 6: AuthLogging.java

And now we obtain the following output:

```
  > ajc banking\*.java logging\*.java
2 > java banking.Test
  <credit>
4 <debit>
  <transfer>
6     <credit>
      <debit>
8 <transfer>
      <credit>
10    <debit>
  Exception in thread "main" banking.InsufficientBalanceException:
12                           Total balance not sufficient
      at banking.AccountSimpleImpl.debit(AccountSimpleImpl.java:24)
14    at banking.InterAccountTransferSystem.transfer(
                             InterAccountTransferSystem.java:10)
16    at banking.Test.main(Test.java:13)
```

This example will serve now as the base to compare authentication and authorization implementations.

4.2. Authentication

Taking the banking example, we add now the authentication functionality. It is presented in two parts. The first one uses JAAS straightforward in the code and the second one implements AspectJ-based solution including JAAS authentication functionality.

Conventional solution

Let's start with JAAS. The upfront login approach is employed in the test program. The user is identified at the beginning of the main method. If the login fails, an exception is thrown and the program stops. In the book [3], R. Laddad avoid the just-in-time authentication too complex for our discussion.
We use the `LoginContext` and the callback handlers. The login configuration file stores the name of the authentication module to be executed. In our case, a simple module provided in the JAAS tutorial [4] is used. The classes are located in `sample` package. The file `sample_jaas.config` stores the path of the authentication module.

```
Sample {
   sample.module.SampleLoginModule required debug=true;
};
```

We will not go deeper into the JAAS technology but focus on the AspectJ-based solution.
Let us see what's changed in the `TestJAAS.java` compared to the `Test.java`:

```
package banking;

import javax.security.auth.login.LoginContext;
import com.sun.security.auth.callback.TextCallbackHandler;

public class TestJAAS {
    public static void main(String[] args) throws Exception {

      LoginContext lc = new LoginContext(
                                "Sample",
                                new TextCallbackHandler());
      lc.login();

      Account account1 = new AccountSimpleImpl(1);
      Account account2 = new AccountSimpleImpl(2);
      account1.credit(300);
      account1.debit(200);

      InterAccountTransferSystem.transfer(account1, account2, 100);
      InterAccountTransferSystem.transfer(account1, account2, 100);
    }
}
```

Listing 7: TestJAAS.java

We have created the `LoginContext` object and the call of the method `lc.login()`. The login will call `SampleLoginModule` class and checks the user name and password. The other methods are called only if the login is successful. To improve our logging aspect, we have add the capture of the authentication (login challenge) in the join point:

```
package banking;

import org.aspectj.lang.*;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import logging.*;

public aspect AuthLogging extends IndentedLogging {
    declare precedence: AuthLogging, *;

    public pointcut accountActivities()
  : execution(public * Account.*(..))
  || execution(public * InterAccountTransferSystem.*(..));

    public pointcut authenticationActivities()
  : call(* LoginContext.login(..));

    public pointcut loggedOperations()
  : accountActivities()
  || authenticationActivities();

    before() : loggedOperations() {
  Signature sig = thisJoinPointStaticPart.getSignature();
  System.out.println("<" + sig.getName() + ">");
```

```
    }
}
```

Listing 8: AuthLogging.java including the login

And now if we compile the aspect and run the `TestJAAS.java` we get the following output:

```
<login>
user name: testUser
password: testPassword
    [SampleLoginModule] user entered user name: testUser
    [SampleLoginModule] user entered password: testPassword
    [SampleLoginModule] authentication succeeded
    [SampleLoginModule] added SamplePrincipal to Subject
<credit>
<debit>
<transfer>
    <credit>
    <debit>
<transfer>
    <credit>
    <debit>
Exception in thread "main" banking.InsufficientBalanceException: Total balance not sufficient
  at banking.AccountSimpleImpl.debit(AccountSimpleImpl.java:24)
  at banking.InterAccountTransferSystem.transfer(InterAccountTransferSystem.java:10)
  at banking.TestJAAS.main(TestJAAS.java:20)
```

AspectJ-based solution

Now comes the solution with AspectJ. The idea is to insert the login into the aspect and leave the main program without instructions which involves an authentication process. It will also avoid code scattering in case of just-in-time authentication.
In the solution presented in [3], we have two aspects:

1. An abstract aspect used to authenticate any system.

2. A concrete aspect implementing the abstract one for our specific banking system.

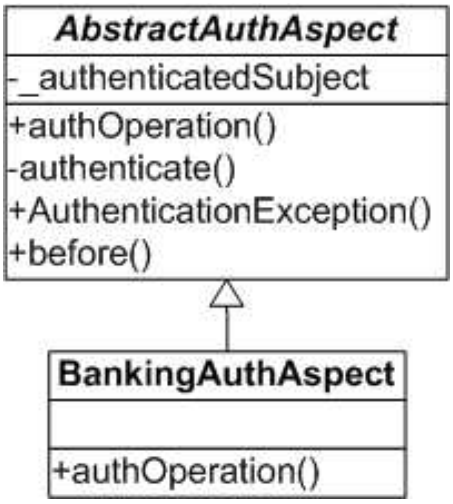

Figure 2.: Banking aspect class diagram

Some differences with the previous solution are noticed in the abstract aspect:

- The creation of a subject. The subject will keep the trace of authentication. Once the user authenticated, the information can be obtain from the `LoginContext`.

- The exception handling within the aspect.

The proper authentication is done exactly as in the conventional solution. We have `LoginContext lc = new LoginContext()` and `lc.login()` contained in a method `authenticate()`.

```
package auth;

import javax.security.auth.Subject;
import javax.security.auth.login.*;
import com.sun.security.auth.callback.TextCallbackHandler;

public abstract aspect AbstractAuthAspect {
    private Subject _authenticatedSubject;
    public abstract pointcut authOperations();

    before() : authOperations() {
  if(_authenticatedSubject != null) {
      return;
    }
    try {
       authenticate();
    } catch (LoginException ex) {
      throw new AuthenticationException(ex);
  }
    }

    private void authenticate() throws LoginException {
        LoginContext lc = new LoginContext("Sample",
          new TextCallbackHandler());
        lc.login();
        _authenticatedSubject = lc.getSubject();
    }

    public static class AuthenticationException
           extends RuntimeException {

           private static final long serialVersionUID = 1L;

           public AuthenticationException(Exception cause) {
                 super(cause);
           }
    }
}
```

Listing 9: AbstractAuthAspect.java

The authentication functionality, once the abstract aspect's done, is very simple to add to our banking system. We need an aspect which extends the `AbstractAuthAspect`:

```
package banking;

import auth.AbstractAuthAspect;

public aspect BankingAuthAspect extends AbstractAuthAspect {
    public pointcut authOperations()
         : execution(public * banking.Account.*(..))
         || execution(
              public * banking.InterAccountTransferSystem.*(..));
}
```

Listing 10: BankingAuthAspect.java

Any call to methods in `Account` and `InterAccountTransferSystem` classes will be captured by the pointcut `authOperations()`. This pointcut extends the `AbstractAuthAspect.authOperations()` with specific details.

We are ready now to test our new configuration. When the aspects are compiled we can run the main application `Test.java`. We will get strictly the same output as we got with the conventional solution.

Discussion

The implementation of aspects seems to take more time than the conventional solution. This is true in this specific example. Now if we think about an application which implements ten times this number of methods, it is obvious that we gain in time and flexibility. Moreover, the implementation of a complex just-in-time authentication becomes very simple.

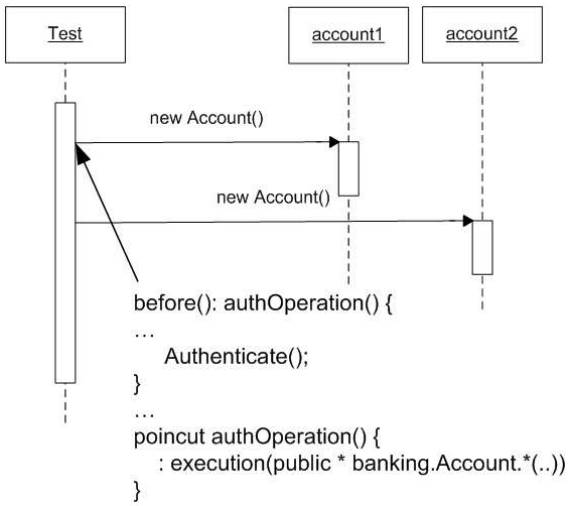

Figure 3.: Sequence diagram of the authentication mechanism

4.3. Authorization

In this section, we will study the authorization issues. As we have implemented the authentication process, now we are ready to check if the user who wants to access resources has the necessary credentials. Again, we'll discuss the conventional solution and then study the aspect-based solution.

Conventional solution

In JAAS, the authorization process is divided into three operations:

1. The authentication of the user (discussed in the previous section)
2. The retrieval of the user's credentials.
3. The verification of the credentials sufficiency for a specific resource access.

In our conventional solution, the retrieval and the verification of credentials will be implemented as such:

- The `AccountSimpleImpl` class is modified. For each public methods, the `AccessController.checkPermission` will be added:

```
public void credit(float amount) {
   AccessController.checkPermission(
                    new BankingPermission("credit"));
   ...
}
```

- We create a `BankingPermission` class which extends the `BasicPermission` class (`java.security.Permission`). The parameter will later map with the string in the security policy files.
- The main program will create an object `Subject` just after the login. It contains the information provided by `lc.getSubject()`. The class `Subject` offers a method `doAsPrivilege(Subject, PrivilegeAction)`. The `PrivilegeAction` is an action object. This object has a `run()` method executing the intended operation like `getAccountNumber()` or `credit()`.
- The policy security which grants the accesses:

```
...
permission banking.BankingPermission "debit";
permission banking.BankingPermission "credit";
...
```

The following listings show how many code lines we have to modify to add the authorization operations. For each operation needing access control, a `Privilege*` has to be created. Each privilege is routed with a try/catch and an `PrivilegeExceptionAction` is thrown if the permission is not granted.

```
package banking;

import java.security.*;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import com.sun.security.auth.callback.TextCallbackHandler;

public class Test {
    public static void main(String[] args) throws Exception {
         LoginContext lc
               = new LoginContext("Sample",
               new TextCallbackHandler());
         lc.login();
         final Account account1 = new AccountSimpleImpl(1);
         final Account account2 = new AccountSimpleImpl(2);
         Subject authenticatedSubject = lc.getSubject();
         Subject
                .doAsPrivileged(authenticatedSubject,
                        new PrivilegedAction() {
                           public Object run() {
                              account1.credit(300);
                              return null;
                        }}, null);
        try {
              Subject
                    .doAsPrivileged(authenticatedSubject,
                           new PrivilegedExceptionAction() {
                             public Object run() throws Exception {
                               account1.debit(200);
                               return null;
                           }}, null);
        } catch (PrivilegedActionException ex) {
             Throwable cause = ex.getCause();
             if (cause instanceof InsufficientBalanceException) {
             throw (InsufficientBalanceException)ex.getCause();
             }
        }
        try {
            Subject
                  .doAsPrivileged(authenticatedSubject,
                        new PrivilegedExceptionAction() {
                           public Object run() throws Exception {
                            InterAccountTransferSystem
                               .transfer(account1, account2, 100);
                         return null;
            }}, null);
         } catch (PrivilegedActionException ex) {
              Throwable cause = ex.getCause();
              if (cause instanceof InsufficientBalanceException) {
                 throw (InsufficientBalanceException)ex.getCause();
              }
       }
       try {
           Subject
                 .doAsPrivileged(authenticatedSubject,
                       new PrivilegedExceptionAction() {
                           public Object run() throws Exception {
                            InterAccountTransferSystem
                               .transfer(account1, account2,100);
                       return null;
                    }}, null);
       } catch (PrivilegedActionException ex) {
            Throwable cause = ex.getCause();
            if (cause instanceof InsufficientBalanceException) {
               throw (InsufficientBalanceException)ex.getCause();
            }
       }
    }
}
```

Listing 11: Authorization using JAAS: Test.java

```
package banking;

import java.security.*;

public final class BankingPermission extends BasicPermission {
    public BankingPermission(String name) {
```

```
8           super(name);
    }
10
    public BankingPermission(String name, String actions) {
12          super(name, actions);
    }
14  }
```

Listing 12: BankingPermission.java

```
    package banking;
2
    import java.security.AccessController;
4
    public class AccountSimpleImpl implements Account {
6       private int _accountNumber;
        private float _balance;
8
        public AccountSimpleImpl(int accountNumber) {
10          _accountNumber = accountNumber;
        }
12
        public int getAccountNumber() {
14          AccessController.checkPermission(
                new BankingPermission("getAccountNumber"));
16        ...
        }
18
        public void credit(float amount) {
20          AccessController.checkPermission(
                new BankingPermission("credit"));
22        ...
        }
24      ...
```

Listing 13: Authorization using JAAS: AccountImpl.java

The security policy file describes the privileges for the user `testUser`:

```
    grant Principal sample.principal.SamplePrincipal "testUser" {
2       permission banking.BankingPermission "credit";
        permission banking.BankingPermission "debit";
4       permission banking.BankingPermission "getBalance";
        permission banking.BankingPermission "transfer";
6   }
```

Finally, the last file to modify is the `AuthLogging.java`. We need to add, as we did for the authentication, the pointcut `authorizationActivities()`.
When we run the test program, we get the following output:

```
    <credit>
2       <login>
        [SampleLoginModule] user entered user name: testUser
4       [SampleLoginModule] user entered password: testPassword
        [SampleLoginModule] authentication succeeded
6       [SampleLoginModule] added SamplePrincipal to Subject
        <doAsPrivileged>
8   <debit>
        <doAsPrivileged>
10  <transfer>
        <doAsPrivileged>
12          <credit>
            <debit>
14  <transfer>
        <doAsPrivileged>
16          <credit>
            <debit>
18  Exception in thread "main" banking.InsufficientBalanceException: Total balance not sufficient
      at banking.AccountSimpleImpl.debit_aroundBody4(AccountSimpleImpl.java:32)
20    at banking.AccountSimpleImpl.debit(AccountSimpleImpl.java:1)
      at banking.InterAccountTransferSystem.transfer_aroundBody0(InterAccountTransferSystem.java:10)
22    at ...
```

It is obvious that the solution proposed creates some important modifications in the codes. If the number of methods increases then the modifications will increase proportionally. And if we decide to

change the authorization operations then we'll need to modify every methods touched by those changes. Last point but not least, the programmer has to be very careful about the methods which need to be checked and the others.

Aspect-based solution

Let's see what AspectJ-based solution looks like. The key in this solution is the use of the worker pattern. It will save a lot of codes. The worker pattern is described in Chapter 8 of [3].
The abstract aspect we've implemented for the authentication is our base. We have now to add the authorization operations:

- We insert an `around` advice that creates and execute the worker object. This is like in the conventional solution: we call the `Subject.doAsPrivilege()` but the `run()` calls the `proceed()` instead of the specific method like `credit()`, `debit()`.
- A check of permissions (`AccessController.checkPermission()`) is made before any `authOperation()` calls.
- An `AuthorizationException` class is declared.

```
package auth;

import org.aspectj.lang.JoinPoint;
import java.security.*;
import javax.security.auth.Subject;
import javax.security.auth.login.*;
import com.sun.security.auth.callback.TextCallbackHandler;

public abstract aspect AbstractAuthAspect {
    private Subject _authenticatedSubject;
    public abstract pointcut authOperations();

    before() : authOperations() {
         if(_authenticatedSubject != null) {
           return;
         }
      try {
        authenticate();
      } catch (LoginException ex) {
         throw new AuthenticationException(ex);
      }
    }

    public abstract Permission getPermission(
              JoinPoint.StaticPart joinPointStaticPart);

    Object around()
        : authOperations() && !cflowbelow(authOperations()) {
        try {
            return Subject
                    .doAsPrivileged(_authenticatedSubject,
                    new PrivilegedExceptionAction() {
                       public Object run() throws Exception {
                           return proceed();
                       }}, null);
        } catch (PrivilegedActionException ex) {
                throw new AuthorizationException(ex.getException());
        }
    }

    before() : authOperations() {
         AccessController.checkPermission(
            getPermission(thisJoinPointStaticPart));
    }

    private void authenticate() throws LoginException {
         LoginContext lc = new LoginContext("Sample",
                   new TextCallbackHandler());
         lc.login();
         _authenticatedSubject = lc.getSubject();
    }

    public static class AuthenticationException
         extends RuntimeException {
       public AuthenticationException(Exception cause) {
           super(cause);
       }
```

```
58      }
60      public static class AuthorizationException
            extends RuntimeException {
62          public AuthorizationException(Exception cause) {
                super(cause);
64          }
        }
66  }
```

Listing 14: AbstractAuthAspect.java implementing the authorization

Like in the AspectJ-based solution for the authentication, the abstract aspect must be be implemented by a concrete one. The `BankingAuthAspect` is almost identical as the one in the authentication. The pointcut `authOperation()` will capture any execution of `Account.*` or `InterAccoutTransferSystem.*`.
We implement a `getPermission()` method in this concrete aspect. It instanciates the object `BankingPermission()` with the name of the method obtained by the join point's static information. This is like the `AccessController.checkPermission` (see Listing 13) in the conventional solution. The permission scheme is identical as the principal declared in the security policy file.

```
    package banking;
2
    import org.aspectj.lang.JoinPoint;
4   import java.security.Permission;
    import auth.AbstractAuthAspect;
6
    public aspect BankingAuthAspect extends AbstractAuthAspect {
8       public pointcut authOperations()
            : execution(public * banking.Account.*(..))
10          || execution(public * banking.InterAccountTransferSystem.*(..));

12      public Permission getPermission(
            JoinPoint.StaticPart joinPointStaticPart) {
14          return new BankingPermission(
                joinPointStaticPart.getSignature().getName());
16      }
    }
```

Listing 15: BankingAuthAspect.java implementing authorization

Finally, we compile the aspects and run the `Test.java`. We will get almost the same output as the conventional solution. The login appears in a different place (just-in-time policy). The log of each operation occurs before the `doPrivileged()`.

Discussion

This section has proved that AOP can help programmers to develop systems with authorization in an elegant and relatively light way. The conventional solution has shown how heavy the necessary modifications in the existing codes can be. The developer must be sure before changing any line what has to be checked and what not. In a large application, this is a typical source of mistakes and incoherence in the system behaviour.

4.4. Fine tunings

Some points can be customized in the access control solutions. For instance, we have propose an abstract aspect `AbstractAuthAspect()`. This aspect can be implemented by different sub-aspects and used in different contexts.
We could also proposed to separate the authentication and the authorization. Even if our previous solutions are fine in most cases, we might consider a situation where we need an up-front login. This means that the user must be authenticate at the very beginning of the program. As example, we can mention any e-banking system or any system where the user has an account. The access of any resources of the account is authorized only once the user has been identified. To do this with our example, we have to create two pointcuts: one for the authentication and one for the authorization. Then the advices must also be modified accordingly.
We still have the situation where the authorization joint point is reached before the authentication one. Different solutions can be implemented. One possibility is simply to throw an exception if the authorization join point is captured before the user authentication. Checking the status of the `_authenticatedSubject` in the authorization advice would be sufficient.

5. Conclusion

The aim of this paper was to show, step by step, the applicability of AOP for security issues in software systems. For this, we have used one of the most obvious example in term of security: banking system. At first we gave standard or conventional approaches to implement authentication and authorization using JAAS API. Then we have compared these solutions with AspectJ-based solutions.
The differences between both solutions in authentication were not that remarkable. The development time of the aspect-based solution was as long as the conventional one. This would not be the case anymore in a large application. The aspect-based solution has shown qualities that we are all looking for: reusable codes and flexibility.
In the authorization section, the implementation of the conventional solution in the banking example has been already long to code. Many important changes were made within the different classes and almost all of their methods were affected. Again, with the abstract aspect, we have described the checking mechanism only once, then with a concrete aspect, we have defined what had to be checked. The main impact is that no core codes were touched by the changes.
I'm not playing AOP's disciple in this paper but I do admit that studying the particular topic of authentication and authorization (as well as logging) has changed my point of view on AOP. I was not fully convinced by the application and the needs of AspectJ after the readings of the first parts of the book *AspectJ in Action* [3]. But now I would recommend the use of such methodology in applications which show common concerns like logging or security.
I think that OOP and AOP have both their place in modern programming. It is difficult to imagine core or business logic programmed only with aspects. But AOP will help in the limitations that OOP has specifically in crosscuttings concerns.

A. Downloadings and setup

A.1. AspectJ compliler

The compiler AspectJ is available on [1].
The package is a zip file. This archive contains the AJDT feature as a feature plugin and a set of standard plugins. It should be used by anyone who cannot make use of the update site for AJDT. To install it, please unzip it in your eclipse install directory (the same directory in which you can find `eclipse.exe`) - it will correctly place the various components in the right directories below this install directory and next time you start Eclipse, you will have AJDT.

A.2. JAAS API

The examples are using the JAAS API.
If J2SE 1.5.0 is installed on the machine, you get JAAS automatically. Otherwise, this API is available with J2EE environment and must be installed before running the test programs.
The package is available on [6].

A.3. Examples and source codes

The source codes of the different examples are available at:
http://www.manning-source.com/books/laddad/laddad_src_Aspectj-In-Action.zip
The directory `ch10/` in the zip file contains all source codes listed in this paper.
To be able to execute all test programs (`Test.class`) where JAAS is used, we need to add one parameter to the Virtual Machine:
`-Djava.security.auth.login.config=sample_jaas.config`. This is the name of the config file used during the execution.

References

[1] Eclipse.org. The AspectJ project at Eclipse.org. [online]. http://www.ecplipse.org/aspectJ (accessed May 22, 2006).

[2] R. Laddad. Simplify your logging with AspectJ. [Retrieved May 22, 2006, from http://www.developer.com/java/other/article.php/3109831].

[3] R. Laddad. *AspectJ In Action*. Manning, 2003.

[4] S. Microsystem. Simplify your logging with AspectJ. [Retrieved May 22, 2006, from http://www.java.sun.com/j2se/1.4/docs/guide/security/\\jaas/tutorials/GeneralAcnAndAzn.html].

[5] Sun Microsystem. J2SE 1.5. [online]. http://java.sun.com/j2se (accessed May 22, 2006).

[6] Sun Microsystem. Java EE At a Glance. [online]. http://java.sun.com/j2ee (accessed May 22, 2006).

Transaction Management

Lorenzo Clementi
University of Fribourg
Department of Informatics
Bd de Pérolles 90
CH-1700 Fribourg
lorenzo.clementi@unifr.ch

Abstract Aspect Oriented Programming (AOP) is a methodology that, combined with the conventional Object Oriented Programming (OOP), allows to separate the business logic from the crosscutting concerns of a program. Further, the crosscutting concerns can be modularized and reused.
This document discusses the way transaction management is realized using the AOP methodology and makes a comparison with the Enterprise Java Beans (EJB) architecture.

Keywords: Aspect Oriented Programming (AOP), Transaction Management, AspectJ.

1. Introduction

This article analyzes the way AOP deals with transaction management and makes a comparison between the EJB and the AOP solution. Our research is mainly based on the book "AspectJ in Action" by R. Laddad [8], especially Chapter 11, that handles transaction management and Chapter 8, that handles AOP design patterns and idioms.
First, we shortly summarize the concepts of *transaction management* and *aspect oriented programming*; however, it is supposed that the reader has some previous knowledge on these subjects. Then, we outline two transaction management approaches that do no use AOP. The first example is the EJB architecture; the second one is taken from [8] and represents a simple banking system that needs persistence. In order to point out the importance of transactions, we present a JDBC based implementation of the banking system that does not use them. Later, we introduce transaction management using AOP and, in the following paragraph, this solution is improved thanks to AOP design patterns. Finally, the last paragraph discusses benefits and drawbacks of the AOP approach to transaction management, comparing it with the EJB architecture.

1.1. Aspect Oriented Programming

In the same way as object oriented programming allows to modularize the core concerns of a program, aspect oriented programming allows to modularize the crosscutting concerns and to separate them from the business logic. A *crosscutting concern* provides a service used by many modules of the program. Examples of crosscutting concerns are transaction management, logging, caching, and so on. The implementation of crosscutting concerns with the convential object oriented approach has three main drawbacks:

1. the business logic of the program is difficult to understand, because its code is mixed with crosscutting concern code (code tangling)
2. the crosscutting concern code is spread in many modules (code scattering)
3. as a consequence of the first two points, the modification or the addition of a crosscutting concern is difficult and error-prone.

AOP aims to solve these problems, providing a way to modularize crosscutting concerns, thus making them reusable and easier to maintain.

Environnement
The code presented in this article is written in AspectJ, an aspect-oriented extension to the Java™programming language. The IDE used is Eclipse 3.1, including the AJDT (AspectJ Developement Tools) plugin version 1.3.1, which provides the AspectJ version 1.5.1a. Before running the examples, read the information in A.

1.2. Transaction management

A *transaction* is a set of operations that must respect the ACID principle [10]. ACID stands for Atomicity, Consistency, Isolation and Durability; the most important of these properties is atomicity: if one operation fails, then the entire transaction must fail and the database must be brought back to a consistent state[1].
The transaction management is a crosscutting concern, because many modules need to use transactions. In a banking application, for example, there could be a module that needs to connect to the database in order to check a customer's asset and another module that needs to access the database in order to manage the customer's personal data: both of them and, in a real system, many other modules as well, need transaction support.

2. Transaction management without AOP

There exist several transaction management commercial solutions. In this article we focus on Enterprise JavaBeans, a server-side architecture developed by Sun Microsystems™that provides a built-in support for transaction management: the next paragraph outlines this architecture's main features. The following paragraph introduces an example taken from [8] and proposes a conventional solution, without using AOP.

[1]This operation is called rollback.

2.1. Enterprise JavaBeans (EJB)

Enterprise JavaBeans is a set of server-side technologies composed by two main elements [6, 7, 14]:

- the EJB Components, that are reusable software units implementing the application business concerns,
- the EJB Container, that makes system services available to the Components.

This architecture allows to develop a server-side application by combinig already existing EJB Components and implementing the application business logic in new EJB Components. Further, there is no need to worry about low-level concerns, such as transaction management, multi-threading, synchronization, and so on. In fact, these low-level concerns are handled by the EJB Container. Figure 1 shows the global EJB architecture.

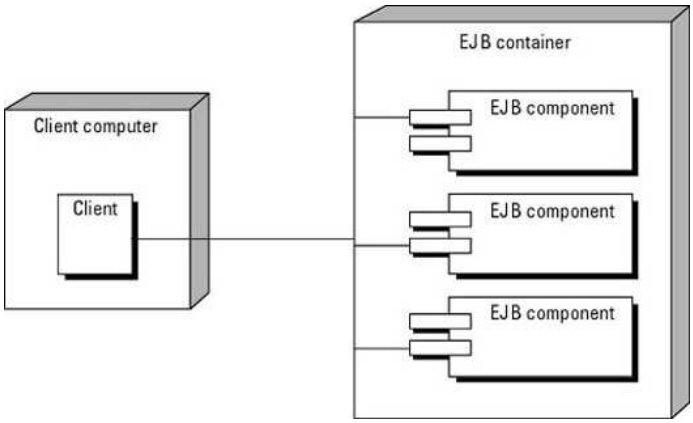

Figure 1.: The container-component architecture in a J2EE application [5].

A standarized programming interface is used to manage the relation between the Container and the Components, enabling the developer to use and combine several modules, even from different vendors [6].
There exist two kinds of transactions [14, 12]:

1. JDBC (Java Database Connectivity) transactions, that are controlled by the Database Management System's transaction manager. This component manages all the database updates and the communication with the Java application is done through the JDBC driver. JDBC transactions are used when only one data source is involed.
2. JTA (Java Transaction API) tansactions, that are controlled by the J2EE application server. These transactions are used when more data sources are involved.

Further, J2EE supports both programmatic and declarative transaction demarcation [12]:

> *The component provider can programmatically demarcate transaction boundaries in the component code with the Java Transaction API. Enterprise beans support declarative transaction demarcation, in which the enterprise bean container automatically starts and completes transactions based on configuration information in the components' deployment descriptor. In both cases, the J2EE platform assumes the burden of implementing transaction management.*

In the programmatic transaction demarcation, the developer uses the methods `begin()`, `commit()` and `rollback()` – defined by the `javax.transaction.UserTransaction` interface – to hardcode the transaction begin and end. For our comparison between EJB and AOP, however, the declarative transaction demarcation is more interesting, because it allows the Container to autonomously execute transaction operations [5]. This is done thanks to a deployment descriptor, a textual file that defines, for each Component method, which transaction operation has to be carried out. A snippet of an assembly-descriptor - taken from [5] - is shown in Listing 1. Declarative transaction demarcation consent to separate the business logic code from the transaction management code.

```
<assembly-descriptor>
  ...
  <container-transaction>
    <method>
      <ejb-name>Bank</ejb-name>
      <method-name>setBalance</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  ...
</assembly-descriptor>
```

Listing 1: An EJB deployment descriptor.

To summarize [12]:

> *J2EE transaction management is transparent to component and application code. A J2EE application server implements the necessary low-level transaction protocols, such as interactions between a transaction manager and resource managers, transaction context propagation, and distributed two-phase commit protocol.*

As we can see from the last quotation, the intent of the Enterprise JavaBeans architecture is very similar to the AOP's one: separating crosscutting concerns (the system services provided by the EJB Container) from business logic (the EJB Components). What are, then, the differences between the two systems? To answer this question, we can affirm that the EJB approach has two shortcomings with respect to the AOP one:

1. it must explicitly declare the action to be executed for each method that needs transaction support,
2. it requires an application server.

Now, let's look closer at how transaction management is handled using aspect oriented programming. Later, in the conclusion, we will consider the EJB - AOP comparison again.

2.2. A simple banking system

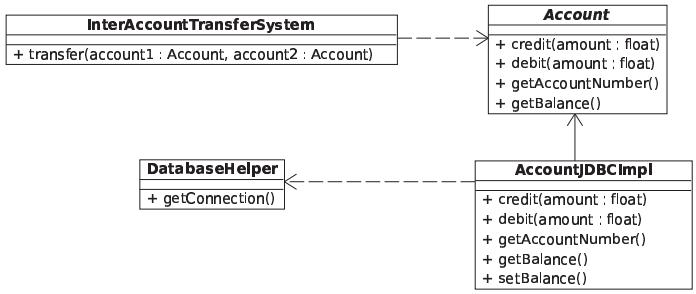

Figure 2.: The UML diagram of the banking system.

Let's go back to Chapter 11 of [8] and look at the example of a simple banking system. Figure 2 shows the UML diagram of a banking system:

- the `Account` interface declares the methods that an account must implement,
- the `InterAccountTransferSystem` class implements the basic operation in our banking system, the transfer of a money amount from an account to another,
- the `AccountJDBCImpl` implements the `Account` interface using `JDBC`,
- the `DatabaseHelper` (see Listing 2) implements a method that returns a `Connection` instance. This class avoids the creation of duplicate connections.

```
public class DatabaseHelper {

  public static Connection getConnection() throws SQLException {
     String url = "jdbc:odbc:bank";
     String user = "user1";
     String password = "password1";
     Connection connection =
    DriverManager.getConnection(url, user, password);
     connection.setAutoCommit(true);
     return connection;
  }
}
```

Listing 2: `DatabaseHelper`

The class `InterAccountTransferSystem` and the interface `Account` define the business logic of the system. The class `AccountJDBCImpl` implements an account business logic but it also contains crosscutting concerns (the connection to the database). The `DatabaseHelper` class does not contain business logic at all. In fact, if the system were to be extended, it is likely that many other modules would use the `DatabaseHelper` class to obtain a connection to the database.
Let's look at how these classes are implemented the conventional way, that is, without using AOP and without transaction management. Listing 2 shows the implementation of the class `DatabaseHelper`, while Listing 3 shows the implementation of the class `AccountJDBCImpl`.
The method `getConnection()` of the `DatabaseHelper` class creates a new connection to the database and returns it to the caller. Note that the auto-commit is set to `true`. This means that, after each update, the data is written to the database; therefore, if an operation fails, all the previous correct updates, will still be commited. Regarding to atomicity, this is the behaviour we want to avoid by mean of transaction management, that will be introduced later.

Concerning the implementation of `AccountJDBCImpl` (Listing 3), we can make two remarks. First, each method that calls `getConnection()` creates a new connection object. In this way, several connections to the same database are created: this reduces the performances of the system, even if it is conceptually correct. Second, note that we introduced an aspect (lines 40 - 44) that softens all the `SQLException`s thrown by the `Account` interface methods. The reason for this softening is that the `Account` methods contain the banking business logic and, therefore, they are not supposed to handle low-level exceptions, such an `SQLException`.

Listing 4 proposes a test scenario. The first two operations (lines 7 and 8) are simple transactions, since they perform only one database update at a time. The `transfer` method, on the other hand, is a *nested transaction*, since it is composed by two operations: debit the first account and credit the second one. Both the operations need to succeed, in order to commit the changes to the database. In this example, the second transfer should produce an error.

Listing 11.4 in [8] introduces a logging aspect that logs account activities (credits and debits), connection activities (committing and rolling back) and database updates. Even if we are going to use the logging aspect output to understand the program's behaviour, we do not deal with its code in this article, since it is not strictly related with transaction management.

2.3. Transaction management: the conventional solution

In the previous paragraph we have presented the case study of a simple banking system. Since the system has to perform a critical operation - transfer money from an account to another - we want to add transaction management to it. This means that, if either the credit or the debit method - called by the transfer method - fails, the transaction manager must rollback the previous updates. The new balance values will be committed only if both credit and debit succeed.
This improvement requires three steps:

1. Switch off the auto-commit mode.-

2. Use the same connection object for all the database updates.

```
public class AccountJDBCImpl implements Account {

  public void credit(float amount)  {
     float updatedBalance = getBalance() + amount;
     setBalance(updatedBalance);
  }

  public void debit(float amount) throws InsufficientBalanceException {
     float balance = getBalance();
     if (balance < amount) {
    throw new InsufficientBalanceException(
       "Total balance not sufficient.");
     }
     else {
     float updatedBalance = balance - amount;
     setBalance(updatedBalance);
     }
  }

  public float getBalance() {
     Connection con = DatabaseHelper.getConnection();
     Statement stmt = conn.createStatement();

     ResultSet rs =
       stmt.executeQuery("select balance from accounts "
       + "where accountNumber = "
       + _accountNumber);

     rs.next();
     float balance = rs.getFloat(1);
     stmt.close();
     conn.close();
     return balance;
  }

  public float setBalance(float balance) throws SQLException {
     // similar to getBalance()
  }

  private static aspect SoftenSQLException {
     declare soft : SQLException
    : execution (* Account.*(..))
    && within(AccountJDBCImpl);
  }
}
```

Listing 3: `AccountJDBCImpl`

```
public class Test {

  public static void main(String[] args) throws Exception {
     Account account1 = new AccountJDBCImpl(1);
     Account account2 = new AccountJDBCImpl(2);

     account1.credit(300);
     account1.debit(200);

     InterAccountTransferSystem.transfer(account1, account2, 100);
     InterAccountTransferSystem.transfer(account1, account2, 100);
  }
}
```

Listing 4: A test scenario

3. Commit the updates only if all operations succeed; otherwise, roll them back.

Switching off the auto-commit mode is trivial. Using the same connection object, on the other hand, can be done in two different ways:

- passing an extra argument to each method that uses a connection;
- using thread-specific storage.

In the first case, a new connection object is created in the `transfer` method and it is then passed as an argument to both `credit` and `debit`. This can be done but, in a more complex system, it would be good to avoid API pollution[2].
The latter option uses the thread-specific storage provided by `ThreadLocal`: when the `DatabaseHelper` is requested a connection, it first checks whether there is one stored in the thread-specific storage. If it is the case, it returns it to the caller; otherwise, it creates a new connection, it stores it in the `ThreadLocal` space and returns it to the caller.

Finally, we have to make sure that only the top-level operation commits the updates to the database. This can be achieved by introducing a variable that takes trace of the call depth: only when this value is zero, the updates are commited.

We have just shown how to add transaction management to our banking system, without using AOP (see code in [8]). However, the conventional solution exposed above has two drawbacks:

1. the transaction management code is scattered in many classes and methods;
2. the code implementing the business logic is tangled with the transaction management code.

These are the typical issues that can be avoided by the use AOP.

3. The first AOP solution

Let's look at the requirements for the aspect oriented version of the banking system:

1. All the updates within a transaction must use the same connection object, that constitutes its context.
2. All the updates must be committed at the top-level and only if all operations succeeded.
3. The solution has to be reusable.

The last requirement is not indispensable but, since AOP is meant to modularize crosscutting concerns, we add it to the core requirements. In practice, these requirements are realized as follows:

Use of the same connection object	We advise the join point in the control-flow of a top-level method, where a connection is created. The connection constitutes the transaction context.
Commit only when the top-level operation completes successfully	The top-level operation is wrapped with an around advice that contains a try/catch block: the operation succeeds only if no exception is thrown. The top-level operations are identified by a pointcut that captures the transacted operations that are not already in the control flow of a transacted operation.
Produce a reusable solution	To produce a reusable solution, abstraction is used: we define an abstract aspect that implements most of the transaction management logic. The subsystems that use it must define the abstract pointcuts declared in this aspect. These pointctus will capture the operations that need transaction management and the join points where a connection is created.

[2] *API pollution*: every method in the execution stack must have extra parameters to pass on the context.

3.1. The abstract aspect

This specification can be translated in two aspects, an abstract one and its concrete implementation, as shown in Listing 5.
Let's analyse Listing 5 in detail.

- `percflow(topLevelTransactedOperation())` (line 2) associates this aspect with each top-level transacted operation. This means that, for each top-level operation, a new instance of this aspect is created; this instance holds the state of the transaction, until it is commited (or rolled back).
- The abstract pointcuts `transactedOperation()` and `obtainConnection()` (lines 4 and 5) capture the operations that need transaction management and that create a new connection respectively. They must be implemented in the concrete subaspect.
- The `topLevelTransactedOperation()` captures the top-level transacted operations. This is done by capturing all the transacted operations not already in the control flow of a transacted operation: `!cflowbelow(transactedOperation())` .
- The around advice to the `topLevelTransactedOperation()` pointcut wraps the transacted operation in a try/catch block: if an exception is thrown, all the updates are rolled back; otherwise, they are committed.
- The around advice to the `obtainConnection()` pointcut (lines 31 - 38) creates a new connection only the first time it is called by a top-level transacted operation. When a connection is required within the same control-flow, it returns the same connection object. This advice also disables the auto-commit mode.
- Finally, we want that only this aspect, and no one else, performs transaction management. For this reason, we use the pointcut `illegalConnectionManagement()` (lines 54 - 60) to capture all the attempts to connect to the database coming from outside the `JDBCTransactionAspect`. The around advice to this pointcut (lines 61 - 64) does not call `proceed()`, which means that no operation is executed.

In fact, there exists two alternatives for the last point. The first option is the one proposed above: a pointcut captures all the illegal attempts to connect to the database and its related advice does not execute them. Another option is to use a policy-enforcement approach. In this case, for each join point captured by the `illegalConnectionManagement()` pointcut, a compile-time error is thrown, forcing the developer to fix the problem.

3.2. The concrete subaspect and the test

All what we have to do at this point, is to extend the abstract aspect - described above - with a concrete aspect that implements the `transactedOperation()` and the `obtainConnection()` pointcuts. Then, we will test the solution with the code of Listing 2 and Listing 3.
The following code snippet shows the `transactedOperation()` pointcut:

```
protected pointcut transactedOperation()
  : execution(* AccountJDBCImpl.debit(..))
  || execution(* AccountJDBCImpl.credit(..))
  || execution(* InterAccountTransferSystem.transfer(..));
```

And the `obtainConnection()` pointcut:

```
protected pointcut obtainConnection()
  : call(Connection DatabaseHelper.getConnection(..));
```

The output of the logging aspect ([8], p. 376) allows us to point out a problem: in case we execute the transfer method but the balance is insufficient, an exception of type `JDBCTransactionAspect.TransactionException` is thrown. However, the method `transfer()` should throw an `InsufficientBalanceException` and this may cause problems to the caller, if it tries to handle the exception. We can solve this problem by using the AOP *exception introduction pattern*[3]. First, the

[3]For more information about AOP design patterns and idioms, see Chapter 8 of [8].

```
public abstract aspect JDBCTransactionAspect
    percflow(topLevelTransactedOperation()) {
    private Connection _connection;
    protected abstract pointcut transactedOperation();
    protected abstract pointcut obtainConnection();

    protected pointcut topLevelTransactedOperation()
  : transactedOperation()
  && !cflowbelow(transactedOperation());

    Object around() : topLevelTransactedOperation() {
  Object operationResult;
  try {
      operationResult = proceed();
      if (_connection != null) {
    _connection.commit();
      }
  } catch (Exception ex) {
      if (_connection != null) {
    _connection.rollback();
      }
      throw new TransactionException(ex);
  } finally {
      if (_connection != null) {
    _connection.close();
      }
  }
  return operationResult;
  }

    Connection around() throws SQLException
  : obtainConnection() && cflow(transactedOperation()) {
  if (_connection == null) {
      _connection = proceed();
      _connection.setAutoCommit(false);
  }
  return _connection;
  }

    public static class TransactionException
  extends RuntimeException {
  public TransactionException(Exception cause) {
      super(cause);
    }
  }

    private static aspect SoftenSQLException {
  declare soft : java.sql.SQLException
      : (call(void Connection.rollback())
         || call(void Connection.close()))
      && within(JDBCTransactionAspect);
  }

    pointcut illegalConnectionManagement()
  : (call(void Connection.close())
     || call(void Connection.commit())
     || call(void Connection.rollback())
     || call(void Connection.setAutoCommit(boolean)))
  && !within(JDBCTransactionAspect);

    void around() : illegalConnectionManagement() {
  // Don't call proceed(); we want to bypass
  // illegal connection management here
  }
}
```

Listing 5: The first AOP solution (the abstract `JDBCTransactionAspect` aspect).

`InsufficientBalanceException` is wrapped with a `TransactionException`; then, as shown in Listing 6, we add an aspect that captures all the methods throwing a `TransactionException`: if the cause of this exception is of the type `InsufficientBalanceException`, the cause is thrown instead of the caught exception.

```
public aspect PreserveCheckedException {
    after() throwing(JDBCTransactionAspect.TransactionException ex)
  throws InsufficientBalanceException
  : call(* banking..*.*(..)
        throws InsufficientBalanceException) {
  Throwable cause = ex.getCause();
  if (cause instanceof InsufficientBalanceException) {
      throw (InsufficientBalanceException)cause;

  } else {
    throw ex;
    }
  }
}
```

Listing 6: An aspect that uses the exception introduction pattern.

With this modification our solution works as expected.

4. An improved AOP solution

The solution we presented in the previous section works fine, if the abstract aspect `JDBCTransactionAspect` has only one concrete subaspect. But consider a situation where you want to provide several concrete subaspects: in our example, you may want to provide two different implementations of the `transactedOperation()` pointcut, one for the class `InterAccountTransferSystem` and one for the class `AccountJDBCImpl`. To do this, we have to use the *participant* and the *worker object creation* patterns.

4.1. Adding the participant pattern

According to the participant pattern structure (see [8], p. 270), we insert in both `AccountJDBCImpl` and `InterAccountTransferSystem` classes a concrete subaspect, inheriting from the `JDBCTransactionAspect` and implementing the abstract pointcuts `transactedOperation()` and `obtainConnection()`. Note that the `InterAccountTransferSystem` class never needs to obtain a connection to the database; however, its nested subaspect must still provide a definition for the `obtainConnection()` pointcut. To get around this problem, we use the following idiom

```
protected pointcut obtainConnection();
```

which defines a pointcut that captures no join points (this is done by omitting the colons symbol).

Everything seems all right now, but it is not. In fact, the abstract aspect `JDBCTransactionAspect` is associated with the control flow of each top-level transacted operation; therefore, a new aspect instance is created each time a `transactedOperation()` join point is captured. This behaviour violates one of the three base requirements of our transaction system (see Section 2.3): with the previous modification, a new connection object is created for both the credit and the debit methods within `transfer()`. This problem can be fixed thanks to the worker object creation pattern.

4.2. The final solution

The problem we have encountered is due to the fact that the connection (or, in other words, the transaction context) is hold by the transaction aspect. We can improve our solution by extracting the connection from the `JDBCTransactionAspect` and creating a special entity that manages it. Let's see in detail how this can be achieved.

1. The `JDBCTransactionAspect` is responsible for the commit and rollback only; we use a new object to store the transaction context.

2. The connection is stored in an object of type `TransactionContext`. We use the worker object creation pattern to create a new `TransactionContext` each time a new transaction is started. The

pattern's worker method is the top-level operation that needs transaction support and it is executed when a new worker object is created.

3. Since we now have a dedicated aspect handling the transaction context, we dissociate the `JDBCTransactionAspect` from the control flow of the `topLevelTransactedOperation()` and we remove its `_connection` field.

Figure 3 shows a diagram sketching part of the banking system structure after introducing the participant and the working object design patterns. Note that, even if the diagram looks very much like an UML diagram, it might not respect the UML convention[4]

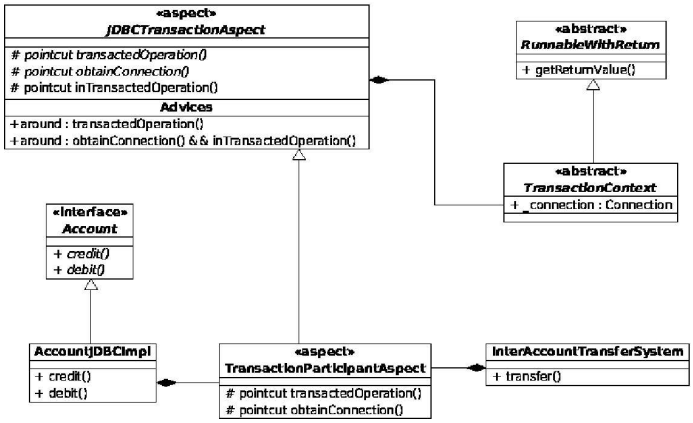

Figure 3.: A diagram sketching part of the banking system structure.

The participant pattern is realized by the `JDBCTransactionAspect` abstract aspect, that declares two abstract pointcuts, `transactedOperation()` and `obtainConnection()`. The `TransactionParticipantAspect` is a concrete aspect that inherits from `JDBCTransactionAspect` and must be implemented by all the participants that share a common feature. In our banking system, this common feature is the need for transaction support and the participants are the classes `AccountJDBCImpl` and `InterAccountTransferSystem`; they both implement a nested `TransactionParticipantAspect`.

The worker object creation pattern, on the other hand, is realized by the classes `RunnableWithReturn` and `TransactionContext`. The following pointcut (Listing 7, lines 11-12),

```
transactedOperation() && !inTransactedOperation(TransactionContext)
```

captures the join points where a transaction begins. Then, according to the pattern structure, an anonymous object of type `TransactionContext` is created. The call to the `run()` method of this object will execute the worker method. Note that the transaction context is created only if there is no transaction in progress for that connection, that means, if the the captured join point is not in the `run()` method control flow.
Lines 36 to 44 of Listing 7 show the use of the *wormhole* design pattern. This pattern allows to pass the context through a method call stack, without having to use additional parameters for each method. In our case, we want the `_connection` object inside the `run()` method to be available from the outside. The following code shows the around advice that realizes the wormhole pattern:

```
Connection around(final TransactionContext context) throws SQLException
  : obtainConnection() && inTransactedOperation(context) {
  if (context._connection == null) {
      context._connection = proceed(context);
      context._connection.setAutoCommit(false);
  }
  return context._connection;
}
```

The AspectJ keyword `this()` - used in the definition of the `inTransactedOperation()` pointcut - collects the context and makes it available inside the advice. In this way, we have created a wormhole that allows other methods to access the connection object inside the `run()` method.

[4]In fact, UML needs to be extended in order to represent new aspect oriented concepts, such as aspects, pointcuts, advices, and so on. However, there is not a wide accepted graphical convention yet [13, 4, 11]; for this reason, the Figure 3 must not be considered an UML diagram, because it could be incorrect.

```
public abstract aspect JDBCTransactionAspect {
    protected abstract pointcut transactedOperation();

    protected abstract pointcut obtainConnection();

    protected pointcut
  inTransactedOperation(TransactionContext context)
  : cflow(execution(* TransactionContext.run())
    && this(context));

    Object around() : transactedOperation()
  && !inTransactedOperation(TransactionContext) {
  TransactionContext transactionContext
      = new TransactionContext() {
        public void run() {
      try {
          _returnValue = proceed();
          if (_connection != null) {
        _connection.commit();
          }
      } catch (Exception ex) {
          if (_connection != null) {
        _connection.rollback();
          }
          throw new TransactionException(ex);
      } finally {
          if (_connection != null) {
        _connection.close();
        }
      }
    }};
  transactionContext.run();
  return transactionContext.getReturnValue();
  }

    Connection around(final TransactionContext context)
  throws SQLException
  : obtainConnection() && inTransactedOperation(context) {
  if (context._connection == null) {
      context._connection = proceed(context);
      context._connection.setAutoCommit(false);
  }
  return context._connection;
  }

    public static abstract class TransactionContext
  extends RunnableWithReturn {
  Connection _connection;
  }

  // illigal connection management,
  // TransactionException and
  // exception softening are unchanged
}
```

Listing 7: The code of the class `JDBCTransactionAspect`, the improved solution.

```
Object around() : transactedOperation()
  && !inTransactedOperation(TransactionContext) {
  TransactionContext transactionContext
      = new TransactionContext() {
        public void run() {
      UserTransaction ut = null;
      try {
          Context ctx = new InitialContext();
          ut = (UserTransaction)
        ctx.lookup("java:comp/ut");
      } catch (NamingException ex) {
          throw new TransactionException(ex);
      }
      try {
          ut.begin();
          _returnValue = proceed();
          ut.commit();
      } catch (Exception ex) {
          ut.rollback();
          throw new TransactionException(ex);
      }
    }};
  transactionContext.run();
  return transactionContext.getReturnValue();
}
```

Listing 8: The transaction aspect's around advice is modified to use a JTA transaction.

4.3. AOP and JTA

Until now we have used a JDBC based connection for our transactions. However, a more complex system - with several data sources - may require the use of JTA. The AOP transaction solution we have presented in the previous paragraph can easily be adapted to suit this requirement. Listing 8 shows the around advice for a JTA based transaction aspect that uses the programmatic transaction demarcation. The begin, the commit and the rollback of the transaction are hardcoded. Note that we do not need to manage the connection object anymore (as in lines 11 - 34 of Listing 7), since this is done by JTA.

5. Conclusion

In this paper we have summarized the Chapter 11 of [8] in order to show how to build an AOP transaction support for a simple banking system. The requirement to produce a general and reusable solution has implied the use of AOP design patterns, making the solution quite complex. Nevertheless, the result is powerful and it allows the developer to apply this transaction management template to other applications, requiring very little modification. Further, the separation of the business logic from the crosscutting concerns is very effective, which makes the code clean and easy to understand. However, a good IDE is essential in order to take trace of "what happens where".

Another goal of this article was to compare the AOP transaction management with the Enterprise JavaBeans solution. We have shown that both AOP and the declarative transaction demarcation in EJB accomplish an effective separation between the business logic code and the transaction management code. The main advantage of AOP over EJB is that it does not require an application server to run. From this point of view, AOP transaction management can be seen as a lightweighted and reusable alternative to EJB transactions. Both the AOP and the EJB approaches require some configuration. For the aspect oriented version - supposing that the transaction management template is provided - two abstract pointcuts must be implemented for each different subsystem and, if necessary, some other adaptations (according to the use of JDBC, JTA or other technologies) . For the EJB declarative demarcation version, on the other hand, a new assembly-descriptor has to be written for each application, but the transaction logic is provided by the framework. We suppose that, for a large application, the pointcuts offer a quicker way to determine which methods need transaction support; this, however, has not been verified in this article.

Section 4.3 shows that the AOP approach can be used in combination with JTA, so that all the transaction inherent code is removed from the business logic, even using the programmatic transaction demarcation. This demonstrates that these two technologies do not exclude each other *a priori*, but

they can also be combined. So, AOP can be seen as a lightweighted alternative to an application server, but it is also possible to realize transaction management using an application server and the aspect oriented methodology. Further, AOP may also be used by application server's developers as a way to modularize the services they offer.

A. Running the examples

All the examples are taken from [8]; the source code is available on the website [9].
The examples have been tested on a Windows XP SP2 machine, using the Eclipse IDE 3.1 with the AJDT plugin version 1.3.1, which provides AspectJ version 1.5.1a (both Eclipse and the plugin are available on the Eclipse website [2]). We also used a MySQL Server database version 5.0 [3].
Note: the examples containing a call to the AspectJ `percflow()` method do not work with the described configuration. The compilation completes successfully, but they produce an error at the runtime. In order to run those examples, we have separately installed the last AspectJ developement build, available on the AspectJ downloads page [1]. It is then possible to compile the classes from the command line and, in this way, they execute correctly.

References

[1] AspectJ download page: it provides the last stable AspectJ release and the last developement build. [online]. http://www.eclipse.org/aspectj/downloads.php (accessed June 7, 2006).

[2] Eclipse community website, which provides a Java IDE and the AspectJ developement tools (AJDT) plugin. [online]. http://www-128.ibm.com/developerworks/java/library/j-what-are-ejbs/part1/ (accessed May 14, 2006).

[3] MySQL Server downloads page. [online]. http://dev.mysql.com/downloads/ (accessed May 14, 2006).

[4] Gefei Zhang. Towards Aspect-Oriented Class Diagrams. [Retrieved May 18, 2006, from http://www.cse.cuhk.edu.hk/~aoasia/workshop/APSEC05/papers/gefei-zhang-aspect.pdf].

[5] B. Goetz. Understanding JTS – The magic behind the scenes. How J2EE containers hide the complexity of transaction management. [online], 2002. http://www-128.ibm.com/developerworks/java/library/j-jtp0410/index.html (accessed May 25, 2006).

[6] Ken Nordby. What are Enterprise JavaBeans components? – Part 1. [online], June 2000. http://www-128.ibm.com/developerworks/java/library/j-what-are-ejbs/part1/ (accessed May 14, 2006).

[7] Ken Nordby. What are Enterprise JavaBeans components? – Part 2. [online], July 2000. http://www-128.ibm.com/developerworks/java/library/j-what-are-ejbs/part2/ (accessed May 14, 2006).

[8] R. Laddad. *AspectJ in action*. Manning, 2003.

[9] R. Laddad. Source code of the examples in the book "AspectJ in Action". [online], 2003. http://www.manning-source.com/books/laddad/laddad_src_Aspectj-In-Action.zip (accessed June 8, 2006).

[10] A. Meier. *Introduction pratique aux bases de données relationnelles*. Springer, 2002.

[11] A. Mohamed M. Kandé, Jörg Kienzle. From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach. [Retrieved May 18, 2006, from http://icwww.epfl.ch/publications/documents/IC_TECH_REPORT_200258.pdf].

[12] T. Ng. Transaction Management. [online]. http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/transactions/transactions.html (accessed May 22, 2006).

[13] A. Omar Aldawud, Tzilla Elrad. UML profile for Aspect-Oriented software developement. [Retrieved May 18, 2006, from http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/submissions/26-aldawud.pdf].

[14] Sanjay Mahapatra. Transaction management under J2EE 1.2. [online]. http://www.javaworld.com/javaworld/jw-07-2000/jw-0714-transaction.html (accessed May 19, 2006).

Implementing Business Rules Using Aspect Oriented Programming

Dominique Guinard
University of Fribourg
Department of Informatics
Bd de Pérolles 90
CH-1700 Fribourg
dominique.guinard@unifr.ch

Abstract This paper was written for a Seminar organized by the Software Engineering Group (`http://diuf.unifr.ch/softeng`) in 2006 at the University of Fribourg. It discusses the use of woven business rules for enterprise systems, focusing on the use of the Aspect Oriented Programming (AOP) paradigm in order to inject rules into both legacy and new business architectures. It presents, in essence, a summary of the well-known book *AspectJ in Action* ([11]) particularly studying chapter 12: *Implementing Business Rules*. Furthermore, it offers additional examples, supplementary considerations about business architectures as well as the author's view point on the use of AOP for modeling and enforcing rules.

Keywords: AOP, Aspect Oriented Programming, Business Rules, Rules engines, JSR 94, Jess, SOA, Service Oriented Architectures.

1. Introduction

Crosscutting concerns, also known as *transversal requirements*, are certainly one of the biggest burden business architectures have to carry. What is more annoying than having to rewrite (or copy/paste) logging facilities for each new module? What is more time-consuming than to re-imagine security each time a new transaction class is designed? As a consequence the list of technologies proposed to fight these cumbersome concerns spread over business architectures is great: Spring, EJB, .Net, Mono, Hibernate, Seaside, Jess, JSR 168, JSR 94, Struts are just a few of the frameworks, libraries and specifications aiming to solve some of the crosscutting concerns issues. Aspect Oriented Programming (AOP) is surfing on that wave. However, AOP is not yet another framework but a new programming paradigm. Complementary to OOP (Object Oriented Programming) it offers to add a generic way of solving the problem of crosscutting concerns.
This paper is based on the book *AspectJ in Action* ([11]), a de-facto standard in the "yet not so well-known" domain of Aspect Oriented Programming. This document is not an essay on AOP but it provides a summary of a particular use of aspects: the modeling of business rules. After a brief introduction to business rules in modern business architectures, a summary of two solutions using aspects as stated in [11] is provided. To conclude, a review of the proposed solutions is offered.

1.1. Introducing SOA and Business Rules

Today's business needs to be more reactive than ever. As a consequence it oughts to rely on agile IT (Information Technology) infrastructures. From a software point of view *composition* and *separation* are the two keywords of the new business architectures. The raise of Service Oriented Architectures is here to prove it: modern business application softwares have to be designed in a very loosely coupled manner, where the actual business is modeled using ad-hoc compositions of various independent services.
While one could think of using AOP to transform POJOs (Plain Old Java Objects) into webservices, this paper focuses on a more precise best practice of a typical SOA: *the modeling of business rules*. After exposing the benefit of these rules in terms of ROI (Return On Investment) and agile management, we will discuss the use of rules modeled as Java Objects (see Subsection 2.4). Eventually we will discuss the use of specialized rules engine to head towards a very clean solution.

1.2. The Need for Business Rules

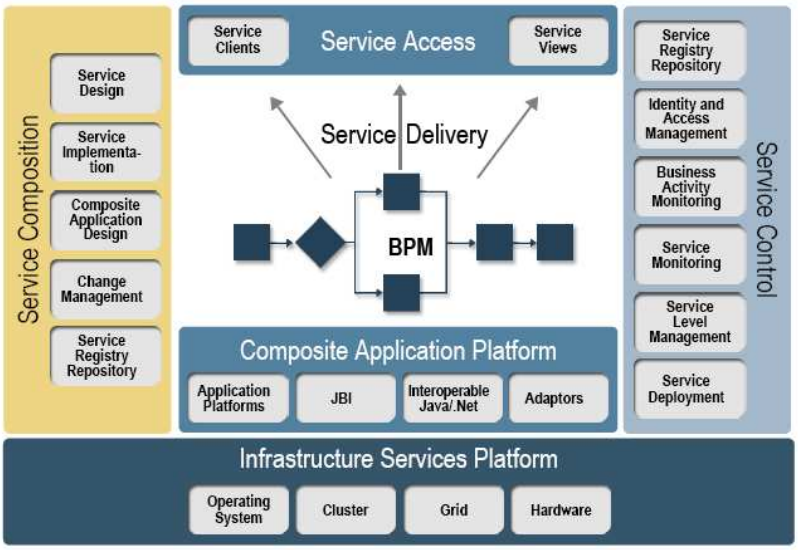

Figure 1.: A typical SOA architecture [13]

Creating a Service Oriented Architecture (see Figure 1) basically requires to split core modules into various independent program units (known as services or web-services), in a manner similar to what

is already done for electronic components. As a consequence, building an application on top of a SOA (theoretically) only requires to *compose and orchestrate services* using a special language such as Business Process Execution Language (BPEL). However, to achieve this ambitious goal the services must be as loosely coupled from the business as possible. In particular no business rule or policy should be embedded (i.e. hard-coded) in the services.
Business rules represent the guidelines for commercial transactions, in [11] it is opposed to the notion of *core business logic* which is the fixed part of the business scheme. Take the example of an online flight booking system. Persisting the booked flight to the database can be considered as a element of the core business logic whereas sending a confirmation per email is not core to the business of the company, but part of the current guidelines.
If the confirmation policy were to be hard-coded into the core system (as it still is for a number of object oriented systems !) any changes decided by the CRM (Customer Relationship Management) team would result into a modification of the system as a whole. For an agile business this latter fact is unsatisfactory both in terms of costs and risk.
Furthermore, any sufficiently big company wants to be able to apply different rules for different clients. For instance two transactions, the first B2C (Business To Customer) the other B2B (Business To Business) may have the exact same core business but not the same terms of contract. Again if the business rules are hard-coded into the application every case will have to be treated in a very clumsy sequence of if-then-elses. This is true even if the core of the transaction remains exactly the same.

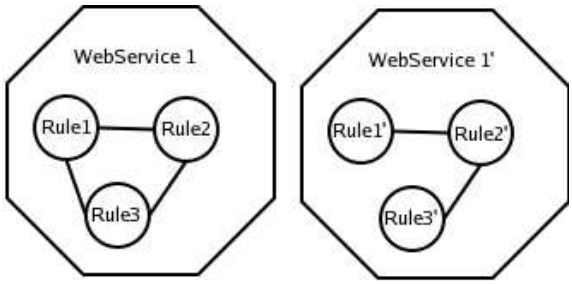

Figure 2.: Webservices using internal rules

To emphasize the need for clearly separated business rules in Service Oriented Architectures as well as in most modern enterprise architectures consider the generic case exposed on Figure 2. On this figure the `Webservice 1` contains internal rules. That is: the business rules are embedded in the software unit called `Webservice 1`. Now, imagine an application wants to use the `Webservice 1` module but needs it with a slightly different rule ordering. As the rules are contained by the service, one needs to rewrite: *(i)* the core of the webservice, even if `Webservice 1`=`Webservice 1'`[1]; *(ii)* all the rules `Rule1' .. Rule3'`; *(iii)* the relations between the rules. On the opposite, imagine the rules are programmed as independent modules as shown on Figure 3. Now, in order to use the module with slightly different rules the programmer only needs to change the relations between the rules. Nevertheless, if the rules are designed using an independent rules engine then no code needs to be re-written in order to setup new relations amongst the rules.
From these examples it should be quite clear that the decoupling of buisness rules from core logic is a good practice for most modern business oriented architectures.

2. Implementing Business Rules

We will now demonstrate the implementation of two solutions based on AspectJ. Whereas the first simply weaves the business rules as concerns modeled with aspects (see Subsection 2.4), the second goes one step further and proposes an implementation using both aspects and a so called *rules engine* (see Subsection 3.3). We first start by exposing a general template for modeling rules using aspects.

[1] Another solution would be to subclass `Webservice 1` if the service is implemented using an Object Oriented language. However, it is worth noting that the subclassing of a service as a whole is not a trivial task.

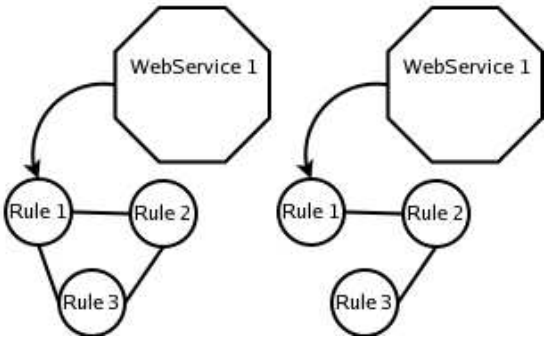

Figure 3.: Webservices using external rules

It is worth noting at this point that the provided examples are programmed in *AspectJ*, a concrete (Java) implementation of the AOP paradigm. Besides, a basic knowledge of the AOP is mandatory to understand this section. Futhermore, all the codes snippets were tested successfully using the Eclipse IDE together with its AspectJ plugin. Appendix A provides more information about the actual setup for running the examples. Eventually, note that some examples were slightly modified with regards to the book ([11]) these are extracted from.

2.1. The Simple Banking Architecture

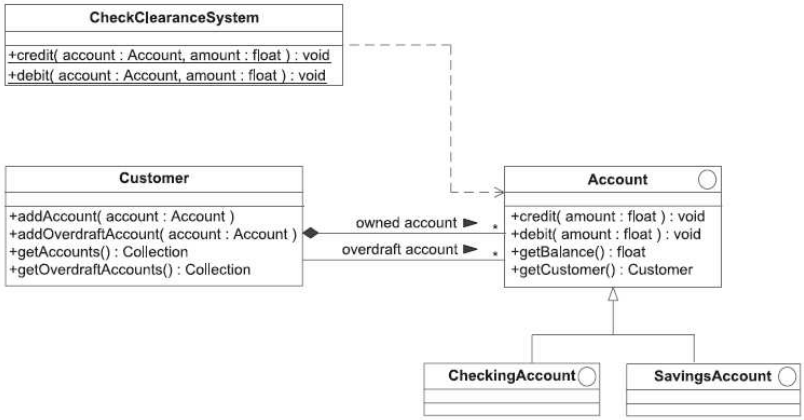

Figure 4.: UML diagram of the simple banking architecture [11]

In order to test the implemented rules and aspects, we will need a small business architecture. This architecture is described on Figure 4. It basically contains the following classes:

- The `Customer` class models a banking customer. It offers various methods for interacting with the `Accounts` belonging to the customer. A `Customer` can have two types of `Accounts`. Overdraft accounts and normal accounts. The formers are a special kind of the latter's and are meant to automatically transfer money to a `CheckingAccount` when an insufficient balance is detected for it.

- The `Account` interface is the common denominator to all the concrete accounts. It basically contains a `credit()` and a `debit()` method. The former is used to add money to the account, the latter is used to take money from the account (from the customer view-point) and can throw an `InsufficientBalanceException` in case an erroneous operation, according to the *business rules*, was attempted.

- The `SavingsAccount` and `CheckingAccount` interfaces are tagging elements used to distinguish the

```
1  package banking;
2
3  public abstract class AccountSimpleImpl implements Account {
4
5    private int _accountNumber;
6    private float _balance;
7    private Customer _customer;
8
9    public AccountSimpleImpl(int accountNumber, Customer customer) {
10     _accountNumber = accountNumber;
11     _customer = customer;
12   }
13
14   public int getAccountNumber() {
15     return _accountNumber;
16   }
17
18   public void credit(float amount) {
19     _balance = _balance + amount;
20   }
21
22   public void debit(float amount)
23   throws InsufficientBalanceException {
24     if (_balance < amount) {
25       throw new InsufficientBalanceException(
26       "Total balance not sufficient");
27     } else {
28       _balance = _balance - amount;
29     }
30   }
31
32   public float getBalance() {
33     return _balance;
34   }
35 }
```

Listing 1: The super-account implementation

two type of accounts our architecture offers. Note that we will change the business rules according to the actual type of account the operation was requested for.

Furthermore, a *concrete* super-account is implemented as stated in Listing 1. The structure of this account is not surprising. It implements the `Account` interface and, thus, implements the `getAccountNumber()`, `getCustomer()`, `getBalance()`, `credit()` and `debit()` methods. The last method is the most interesting since it permits to distinguish what is part of the *business rules* from what is part of the *core business logic*. Indeed consider the rule implemented in the `debit()` method (lines 22 - 30). It corresponds to the following verbose rule:

"When a debit operation is called on an account for which the balance is not sufficient, raise an exception".

At this point, one could consider it as a so called *business rule*, however since this rule is a fundamental constraint (not likely to change sometimes) we make it part of the *core business logic*. It is worth noting that a `AccountSimpleImpl` is not used as an actual account within the system (this is why we declare it as `abstract`). Indeed, we still need to extend it by either a `SavingsAccountSimpleImpl` or a `CheckingAccountSimpleImpl`. This fact, represented in Listing 2, will permit to tag the account in order to distinguish savings accounts from checking accounts.

2.2. Two Business Rules

Now that the scene is set, we need to choose a number of business rules. In order to keep the examples quite easy to understand only two rules are chosen. Additionally, Subsection 3 will explore a third business rule.

Minimum Balance for Savings Accounts The first rule is a very common one for the banking business:

"A savings account should not fall below a predefined amount."

```
package banking;

public class CheckingAccountSimpleImpl
  extends AccountSimpleImpl implements CheckingAccount {

  public CheckingAccountSimpleImpl(int accountNumber, Customer customer) {
    super(accountNumber, customer);
  }

  public String toString() {
    return "CheckingAccount(" + getAccountNumber() + ")";
  }
}
```

Listing 2: An actual savings account

The basic idea behind this rule is to advise the `debit()` method of savings accounts in order to check whether the current operation does not let the account fall below the threshold.

Overdraft Protection The second rule is slightly more complicated and will permit to understand that the power of AOP is not limited to logging and simple transcations:

"If a check can not be cleared and the customer has enough money in an overdraft account, clear the check by transferring the required amount from the overdraft account to the checking account."

The idea here is to pause the current clearance to control whether the missing amount can be transfered from a so called "overdraft account". The current transaction is then resumed and results with: *(i)* an error if no overdraft account was found; *(ii)* an error if the amount on overdraft account was not sufficient; *(iii)* a successful end of transaction if a sufficient amount was found and transfered.

2.3. Various Approaches

Before actually starting to implement the guidelines exposed above, let us compare the different approaches towards a clean implementation of the business rules. The four conceptually most important approaches are listed below.

The If-Then Approach

The first and most simple way of modeling rules is by introducing them as if-then statements into the core modules. This approach is the most straightforward in terms of implementation but also in terms of resulting bugs (!). Indeed, tangling the core modules with business rules often results in poorly reusable architecture. Furthermore, in such a system, extending or modifying the rules implies to rewrite a number of core classes. If this approach may be viable in a research lab it certainly is not for business architectures.

The POJOS Approach

On the way leading to a better solution, programming the rules as independent modules taking the form of POJOs (Plain Old Java Objects) is an improvement but not yet an optimal solution. Indeed, since the rules are usually spread over many modules this approach complicates the core logic and results in a complicated architecture. Maintaining and extending such a system requires a precise knowledge of the scattered business rules which prevents a clean separation of the concerns.

The Business Rules Engine Approach

As seen before the clean separation of the core and rules concerns is of great importance. This point is precisely the goal of rules engines. As this topic will be detailed in the Subsection 3.3 we will not go into further details here. However, as with logging libraries, a rules engine still needs to be called from the core modules. This fact is not optimal since it makes the code depending on an engine vendor or an API.

```
aspect BusinessRuleCoordinator {
  pointcut methodsNeedingBusinessRules(<context...>) : call(...) ...;

  before(<context...>) : methodsNeedingBusinessRules() {
    // Evaluate rules
    // Fire action for matching rules
    // If rule engine is used:
      // 1. Initialize rule engine with the context
      // 2. Run rule engine evaluation
  }
}
```

Listing 3: A template for an aspect that coordinates business rule invocation

The Aspects Approach

The last solution presented here is to use aspects. As seen before, business rules are a typical example of crosscutting concerns. Thus, it is quite naturally that the AOP proposes an elegant solution to the problem.
On the one hand, when using the "POJO and aspects" approach the aspects permit to fully externalize the rules and spread them over various modules, thinking of the scattering in terms of transactions instead of objects. On the other hand, when using a rules engine the aspects offer to use the engine in a manner that is transparent to the core logic. Furthermore, it reduces the dependency to the engine vendor or API to the simple rewriting of one aspect instead of many core modules.

2.4. Implementing Rules Using Aspects

A Template for Modeling Business Rules

This subsection provides a concrete example of how to implement the required rules (as stated in Subsection 2.2). We shall start by a defining a template for modeling and weaving business rules using the aspect oriented programming. Listing 3 presents this template. There is no surprise: the first step consists of capturing joinpoints using a pointcut. As we will apply the rules according to the state of the system we need to capture the context. Thus the `methodsNeedingBusinessRules` pointcut catches it. Usually a (business) rule is meant to be executed *before* the code it refers to. Thus, the advice body is activated `before` further executing the code at the joinpoint. Inside the advice the rules are evaluated against the captured context and the corresponding actions are triggered. Furthermore, in case a rule engine is used (see Subsection 3.3) it is initialized and delegated the task of evaluating the rules.

The Debit Abstract Aspect

As all the precited rules are concerned with debit transactions we start by writing an abstract aspect that will serve as a basis for all the subsequent concrete aspects. Listing 4 presents this module. Its first part uses the AOP mechanism called *introduction* to inject the `getAvailableBalance()` into the `Account` class. Since the available balance will depend on the very nature of the account a generic method is introduced by this abstract aspect. The task of introducing a pertinent method is left to the concrete aspects.
Futhermore, we define a pointcut capturing the execution of `debit` methods together with the concerned account and the amount the customer wants to withdraw.

The Minimum Balance Aspect

We now implement the first concrete aspect that models a business rule, namely the minimum balance aspect. As shown on Listing 5 the scheme of this aspect is quite powerful and yet not very complex (at least in terms of lines of code). This module extends the `AbstractDebitRulesAspect` abstract aspect. It first overrides the getAvailableBlance() method for accounts of type `SavingsAccount` in order for it to fulfill the first rule stated in Subsection 2.2. Then it refines the `debitExecution()` pointcut by applying it only to `SavingsAccount`: `&&this(SavingsAccount)`. Eventually, it defines a `before` advice for the previous pointcut. This method actually enforces the minimal balance rule for `SavingsAccount`s and throws an `InsufficientBalanceException` in the case the conditions are not met.

```
package rule.common;

import banking.*;

public abstract aspect AbstractDebitRulesAspect {
    public float Account.getAvailableBalance() {
  return getBalance();
    }

    public pointcut debitExecution(Account account,
            float withdrawalAmount)
  : execution(void Account.debit(float)
        throws InsufficientBalanceException)
  && this(account) && args(withdrawalAmount);
}
```

Listing 4: An abstract aspect for rules concerned with debit transactions

```
package rule.java;

import rule.common.*;
import banking.*;

public aspect MinimumBalanceRuleAspect extends AbstractDebitRulesAspect {
  private static final float MINIMUM_BALANCE_REQD = 25;

  public float SavingsAccount.getAvailableBalance() {
    return getBalance() - MINIMUM_BALANCE_REQD;
  }

  pointcut savingsDebitExecution(Account account, float withdrawalAmount)
  : debitExecution(account, withdrawalAmount)
  && this(SavingsAccount);

  before(Account account, float withdrawalAmount)
      throws InsufficientBalanceException
  : savingsDebitExecution(account, withdrawalAmount) {
    if (account.getAvailableBalance() < withdrawalAmount) {
      throw new InsufficientBalanceException(
          "Minimum balance condition not met (from Aspect)");
    }
  }
}
```

Listing 5: An aspect for minimum balance checking

```
package rule.java;

import java.util.*;
import banking.*;
import rule.common.*;

public aspect OverdraftProtectionRuleAspect extends AbstractDebitRulesAspect {
  pointcut checkClearanceTransaction()
  : execution(* CheckClearanceSystem.*(..));

  pointcut checkingDebitExecution(Account account, float withdrawalAmount)
  : debitExecution(account, withdrawalAmount)
  && this(CheckingAccount);

  before(Account account, float withdrawalAmount)
      throws InsufficientBalanceException
  : checkingDebitExecution(account, withdrawalAmount)
  && cflow(checkClearanceTransaction()) {
    if (account.getAvailableBalance() < withdrawalAmount) {
      performOverdraftProtection(account, withdrawalAmount);
    }
  }

  private void performOverdraftProtection(Account account,
      float withdrawalAmount) throws InsufficientBalanceException {
    float transferAmountNeeded = withdrawalAmount
        - account.getAvailableBalance();
    Customer customer = account.getCustomer();
    Collection overdraftAccounts = customer.getOverdraftAccounts();
    for (Iterator iter = overdraftAccounts.iterator(); iter.hasNext();) {
      Account overdraftAccount = (Account) iter.next();
      if (overdraftAccount == account) {
        continue;
      }
      if (transferAmountNeeded < overdraftAccount.getAvailableBalance()) {
        overdraftAccount.debit(transferAmountNeeded);
        account.credit(transferAmountNeeded);
        return;
      }
    }
    throw new InsufficientBalanceException(
        "Insufficient funds in overdraft accounts");
  }
}
```

Listing 6: An aspect for overdraft protection

The Overdraft Protection

We go on with the implementation of the second business rule: the *overdraft protection*. Again, the concrete aspect for this rule extends the `AbstractDebitRulesAspect` as show on Listing 6. As this aspect is slightly more complicated than the previous one we will describe it in more details. First of all the overdraft protection is available only when the customer uses the clearance system. In order to enforce this guideline a new pointcut is created: `checkClearanceTransaction`. Next we refine, once again, the `debitExecution()` pointcut in order for it to capture debit transactions on checking accounts only: `&& this(CheckingAccount)`.

The `before()` advice starting on line 15 advises the methods for debit transactions that occurred in the control flow of `checkClearanceTransaction`. That is it will advice all the debit transactions issued on a check clearance system (see lines 8 - 9). This advice is also responsible for checking whether the amount in the current account is big enough to cover the debit request. If this is the case then there is nothing to attempt with the overdraft accounts. On the contrary, if the amount is not big enough, the advice will call the `performOverdraftProtection()` method.

This latter method (line 24) is the core of the overdraft protection rule. It basically tries to get the missing amount from one of the overdraft accounts the customer owns. If it fails an `InsufficientBalanceException` will be raised just as for any invalid debit request. On the contrary, if the method finds enough money in one of the overdraft accounts (partial withdrawal from multiple overdraft accounts is not implemented here) it will credit the current account (i.e. transfer the money from one account to the other). As a consequence the transaction will be achieved in a completely transparent manner (from the user view-point).

It is worth noting at this point that the power of AOP for this issue does not only come from the clean separation. Indeed, using classical OOP detecting the fact that the debit request comes from a clearance

```
package banking;

import java.io.*;

public class Test {
  public static void main(String[] args) throws Exception {
    Customer customer1 = new Customer("Customer1");
    Account savingsAccount = new SavingsAccountSimpleImpl(1, customer1);
    Account checkingAccount = new CheckingAccountSimpleImpl(2, customer1);
    customer1.addAccount(savingsAccount);
    customer1.addAccount(checkingAccount);
    customer1.addOverdraftAccount(savingsAccount);

    savingsAccount.credit(1000);
    checkingAccount.credit(1000);

    savingsAccount.debit(500);
    savingsAccount.debit(480);

    checkingAccount.debit(500);
    checkingAccount.debit(480);
    checkingAccount.debit(100);

    CheckClearanceSystem.debit(checkingAccount, 400);
    CheckClearanceSystem.debit(checkingAccount, 600);
  }
}

aspect LogInsufficientBalanceException {
  pointcut methodCall() : call(void *.debit(..))
  && within(Test);

  void around() : methodCall() {
    try {
      proceed();
    } catch (InsufficientBalanceException ex) {
      System.out.println(ex);
    }
  }
}
```

Listing 7: A class for testing the aspects

system would not be an easy task. Using the Java call stack is one solution but it is unreliable because many virtual machines are optimized (Just In Time, HotSpots, EJB pool optimizers, etc.) and thus, their call stack is not reflecting the exact order of the instructions in the code[2].

Testing the Rules

A test for the rules is now needed. Listing 7 presents this test method. This script file creates a number of accounts and performs both legal and illegal (in terms of the defined business rules) operations on these.
Additionally, this testing script implements a good example of the so-called *exception softening*. Indeed, the internal aspect (lines 29 - 40) declared in this class catches the exceptions that may be raised (by `debit()` operations) within the `main()` method.
Lastly, we analyze the output of the test ran together with a logging aspect (see Appendix A.5). The first lines (7 - 17) are not surprising, they reflect perfectly legal operations performed on well-balanced accounts:

```
[AccountSimpleImpl.credit(..)] SavingsAccount(1) 1000.0
Before: 0.0
After: 1000.0
[AccountSimpleImpl.credit(..)] CheckingAccount(2) 1000.0
Before: 0.0
After: 1000.0
[AccountSimpleImpl.debit(..)] SavingsAccount(1) 500.0
Before: 1000.0
After: 500.0
```

The balance for the `SavingsAccount` is now $500 whereas for the `CheckingAccount` it is $1000. The next instruction (line 18) is an attempt to debit the `SavingsAccount` of $480. Since the `MinimumBalanceRuleAspect` woven with the core logic is checking the debit transaction against a mini-

[2]See [14] for an example of what kind of stack "inconsistencies" an optimized virtual machine may generate.

mum residual balance of $25 for the `CheckingAccount`s (see Subsubsection 2.4), the operation fails and an exception is raised:

```
[AccountSimpleImpl.debit(..)] SavingsAccount(1) 480.0
Before: 500.0
After: 500.0
banking.InsufficientBalanceException: Minimum balance condition not met (from Aspect)
```

Note that since the aspect raises and exception the transaction is `aborted` (not executed at all to be more precise since the advice is a `before` advice) and the balance remains the exact same.
We go on with lines 20 and 21:

```
[AccountSimpleImpl.debit(..)] CheckingAccount(2) 500.0
Before: 1000.0
After: 500.0
[AccountSimpleImpl.debit(..)] CheckingAccount(2) 480.0
Before: 500.0
After: 20.0
```

The first operation is no surprise since the `CheckingAccount` still contains $1000. The second instruction (line 19) is, however, more interesting. Since we apply the *Minimum Balance Rule* only to `SavingsAccount`s (see snippet `&& this(SavingsAccount)` for the `savingsDebitExecution` pointcut) this account is allowed to go below the threshold of $25 and no exception is raised.
The output of line 22 is interesting as well. As the *overdraft protection rule* is meant to be activated only when passing through the clearance system it is not triggered here. As a consequence the request results in an `InsufficientBalanceException` *rose by the core logic* of the banking system:

```
[AccountSimpleImpl.debit(..)] CheckingAccount(2) 100.0
Before: 20.0
After: 20.0
banking.InsufficientBalanceException: Total balance not sufficient
```

Eventually, the overdraft protection logic is tested. On line 24 an attempt to widthdraw $400 from the `CheckingAccount` (balance: $20) triggers the overdraft protection rule. The system attempts to retrieve: $400 - 20 = 380$ dollars from an overdraft account. Since the `SavingsAccount` was set as such (see line 12 of the test script) $380 are transfered to the `CheckingAccount` and the transaction can be successfully terminated:

```
[CheckClearanceSystem.debit(..)] CheckingAccount(2) 400.0
Before: 20.0
    [AccountSimpleImpl.debit(..)] CheckingAccount(2) 400.0
    Before: 20.0
        [AccountSimpleImpl.debit(..)] SavingsAccount(1) 380.0
        Before: 500.0
        After: 120.0
        [AccountSimpleImpl.credit(..)] CheckingAccount(2) 380.0
        Before: 20.0
        After: 400.0
    After: 0.0
After: 0.0
```

The last operation (line 25) is an attempt to withdraw $600 from the `CheckingAccount`. Since this account has now a balance of $0 the overdraft protection rule is activated once again. However, because the overdraft account (i.e. the `SavingsAccount`) has a balance of $120 only the protection rule fails to transfer enough money to the `CheckingAccount` and raises an exception:

```
[CheckClearanceSystem.debit(..)] CheckingAccount(2) 600.0
Before: 0.0
    [AccountSimpleImpl.debit(..)] CheckingAccount(2) 600.0
    Before: 0.0
    After: 0.0
After: 0.0
banking.InsufficientBalanceException: Insufficient funds in overdraft accounts
```

This example already shows how one can implement relatively complicated business rules without modifying the core logic of an enterprise application.

3. Extending the Example

In order to fully understand the power of the AOP for modeling business rules we will now extend our application with two new business rules. Note that, unlike the other examples, this code was developped by the author of the paper and is thus not part of the package you can download from the official reference book's website.

```java
/** File: YouthAccount.java */
package banking;

public interface YouthAccount extends Account {

}
```

Listing 8: The flagging interface for a youth account

```java
/**
 * File: YouthAccountSimpleImpl.java
 * This account represents the concrete implementation of an account for young
 * customers.
 * Author: Dominique Guinard
 * Mail: dominique.guinard 'a t' unifr.ch
 * Web: http://www.gmipsoft.com/unifr
 * University of Fribourg, Switzerland
 */

package banking;

public class YouthAccountSimpleImpl extends AccountSimpleImpl implements
    YouthAccount {

  public YouthAccountSimpleImpl(int accountNumber, Customer customer) {
    super(accountNumber, customer);
  }

  public String toString() {
    return "YouthAccount(" + getAccountNumber() + ")";
  }
}
```

Listing 9: The concrete implementation of a youth account

3.1. New Core Elements

For this extension we introduce a new kind of account namely the `YouthAccount` an account for the young people. This account is programmed as a POJO extension of `AccountSimpleImpl` implementing the flagging interface `YouthAccount`. Both the concrete implementation and the interface for this account are available on Listing 9 and Listing 8.

3.2. New Business Rule and Aspects

We now want to introduce two new buiness rule that we will model using an aspect. The rules are as follow:

"The maximal widthdrawal *at once* is: for youth accounts: $200, for savings accounts: $1000, for checking accounts: unlimited."

Now, to model these rules we simply need to create a new aspect with new pointcuts and advices as shown in Listing 11. Additionnaly, we modify the `AbstractDebitRulesAspect` in order to introduce a new method: `Account.getMaximumWithdrawal()` (see Listing 10) into all the available accounts.

Although the rules are different, the syntax of this aspect is very similar to what we encountered in the previous subsections and thus will not be further detailed. However, the key point with this extension is to realize that while adding these new rules *we did not have to change anything within the core business logic*. The power of AOP for business rules resides in this small but yet *very* important difference.

```java
/* File: AbstractDebitRulesAspect.java */
  public float Account.getMaximumWithdrawal() {
    return getAvailableBalance(); /* default implementation */
```

Listing 10: The new method to introduce into all the available accounts

```
/**
 * File: MaximalWithdrawalRuleAspect.aj
 * This aspect models a business rule used to check, for the cases of
 * special accounts (youth accounts, savings accounts),
 * whether the withdrawal does not overcomes a fixed amount.
 * Author: Dominique Guinard
 * Mail: dominique.guinard 'a t' unifr.ch
 * Web: http://www.gmipsoft.com/unifr
 * University of Fribourg, Switzerland
 */

package rule.java;

import banking.Account;
import banking.IncorrectWithdrawalException;
import banking.InsufficientBalanceException;
import banking.SavingsAccount;
import banking.YouthAccount;
import banking.CheckingAccount;
import rule.common.*;

public aspect MaximalWithdrawalRuleAspect extends AbstractDebitRulesAspect {
  private static final float MAXIMUM_WITHDRAWAL_FOR_YOUTH = 200;

  private static final float MAXIMUM_WITHDRAWAL_FOR_SAVINGS = 1000;

  public float SavingsAccount.getMaximumWithdrawal() {
    if ((getBalance() - MAXIMUM_WITHDRAWAL_FOR_SAVINGS) < 0) {
      /*
       * current balance is smaller than the max withdrawal amount, return
       * balance
       */
      return getBalance();
    } else {
      return MAXIMUM_WITHDRAWAL_FOR_SAVINGS;
    }
  }

  public float YouthAccount.getMaximumWithdrawal() {
    if ((getBalance() - MAXIMUM_WITHDRAWAL_FOR_YOUTH) < 0) {
      /*
       * current balance is smaller than the max withdrawal amount, return
       * balance
       */
      return getBalance();
    } else {
      return MAXIMUM_WITHDRAWAL_FOR_YOUTH;
    }
  }

  pointcut specialAccountsDebitExecution(Account account,
      float withdrawalAmount)
    : debitExecution(account, withdrawalAmount)
    && (this(YouthAccount) || this(SavingsAccount));

  before(Account account, float withdrawalAmount)
      throws InsufficientBalanceException
    : specialAccountsDebitExecution(account, withdrawalAmount) {
    if (account.getMaximumWithdrawal() < withdrawalAmount) {
      throw new InsufficientBalanceException(
          "\nAttempting to withdraw an amount bigger than the authorized" +
          " amount for this account: " + withdrawalAmount +
            "\nMaximal authorized withdrawal at once: " +
            account.getMaximumWithdrawal() + "\n");
    }
  }
}
```

Listing 11: An aspect for special rules on withdrawals

3.3. Implementing Rules Using Aspects and a Rule Engine

The previous sections exposed the use of AOP to weave rules modeled as aspects into an existing application. This practice is already quite clean, however most modern business application softwares model rules using dedicated systems. These are the so-called *rules engines*. This subsection firstly provides an introduction to the benefits of rules engines and their use in Service Oriented Architectures, furthermore it exposes an example of the AspectJ/Jess (rules engine) combination heading towards a very clean solution for modeling business rules.

The Benefits of Rules Engines

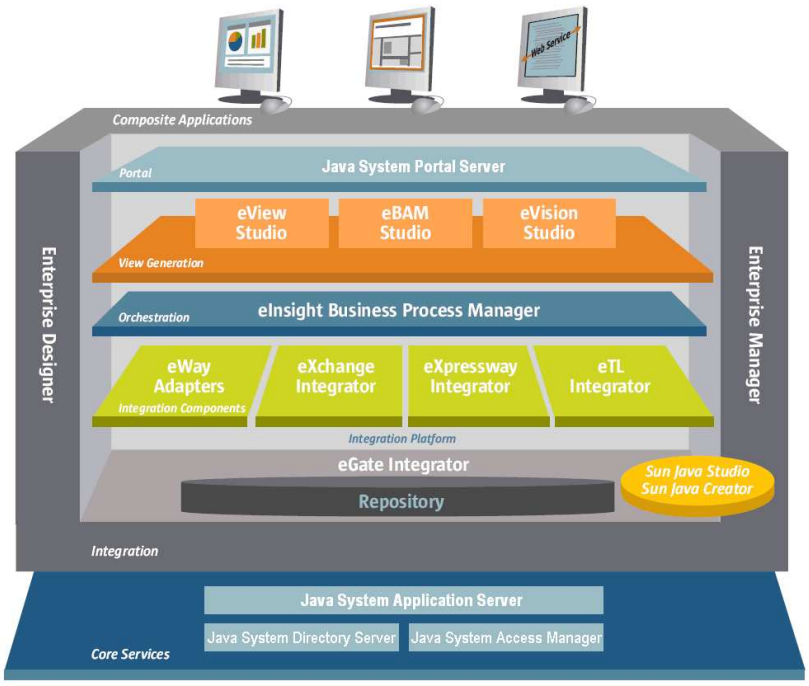

Figure 5.: The Java Composite Application Platform Suite [13]

Nowadays, when building an enterprise application (i.e. an application software supporting business transactions and models) a pertinent choice is to build it on a Service Oriented Architecture basis. These architectures are in fact a number of *best practices* for modular, distributed and easily extensible business applications. As stated in Subsection 1.2, the main goal of service oriented systems is to be as loosely coupled to the state of a business at time t as possible. In particular, the services should be independent from the business rules. Without this latter fact, composition and reuse of service modules is rather difficult.
This explains why most of the enterprise stacks (aka composite application platforms) include rules engines. Let us consider two concrete examples (from the business world) of such softwares. To start with consider Figure 5. This is the Java Composite Application Platform Suite. A well-known enterprise stack provided by SeeBeyond Inc. and Sun Microsystems Inc. This SOA solution is used to build strong business applications such as banking softwares. It offers various layers of abstractions in order to make the programmed modules as reusable as possible. In particular the orchestration layer is the most interesting for this paper. It is composed of *(i)* a BPEL (Business Process Execution Language) engine, used to compose (orchestrate) webservices; *(ii)* a rules engine, used to model business rules.
Together these components permit to compose and reuse the services. Let us consider three services:

- a debit service that operates on credit cards accounts;
- a printing service;
- an invoice emitting service.

Using the BPEL engine one can compose these three services. Firstly, we use the debit service which sends information to the invoice emitting service which, in turn, transmit the invoice to the printing service. Now, while this is the flow for a correct debit transaction it is not for an erroneous one. The decision whether the transaction is correct or not relies on *business rules*. With the proposed stack these rules are modeled using the rules engine which is called by the BPEL engine (or the webservices themselves) at various points to actually execute the rules.
The second example emphasizes the fact that rules engines are not only applied to the banking business but are common to all the SOAs. Figure 6 presents an example of a rules engine used for the management of the supply chain. Here the engine permits to weave retailing rules into the BPEL composition of services. For instance the amount to charge for the shipping of goods could be adapted to the client by a business rule.

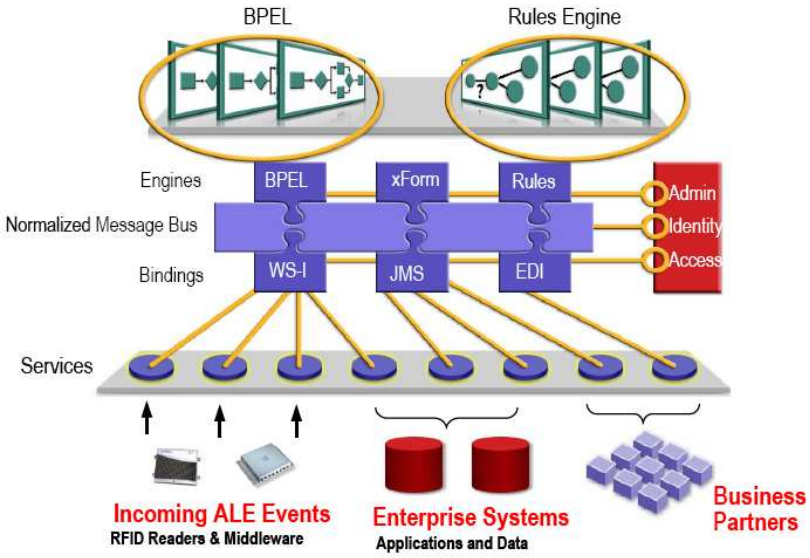

Figure 6.: An example of SOA using a Rule Engine [12]

These two concrete examples are here to emphasize the use of such engine for real companies. We will now briefly introduce the mechanisms of rules engine, focusing on Jess, a well-known engine.

Introduction to Rules Engines

As for almost every component of a business stack, rules engines are standardized through a Java Specification Request (JSR) from the Java Community Process (JCP). Developed by most of the majors of the business software (BEA Systems, Sun Microsystems, Unisys, ILOG, etc), the JSR 94 standard was designed to ensure the use of a common API amongst the vendors of rules engines. This fact is really important as it permits the companies to be free to change the engine vendor anytime they would like to.
Rules engines built according to JSR 94 basically function as exposed on Figure 7. When using an engine, the rules and the core business logic are completely separated. The sequence of processes for applying the rules to the current transactions is as described below:

1. The rules are expressed using a dedicated language (a LISP-like language in the case of Jess, see Figure 12 for an example).

2. The engine parses the rules expressed using a dedicated language.

3. The business logic calls the engine and asks it to invoke the parsed rules on the current context.

In order for the engine to be aware of the context of invocation a special zone between the core and the engine is set: the *working memory*. The elements within this zone are called *facts*. In the example of a

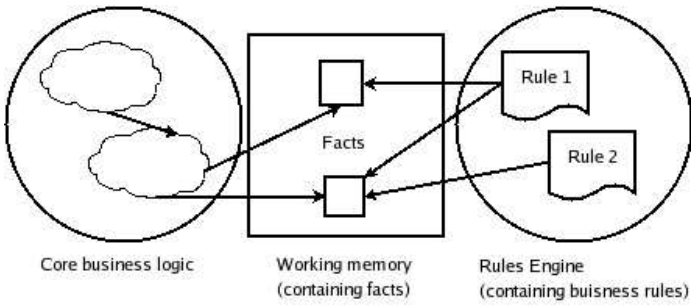

Figure 7.: Collaboration between the rules engine and the core business logic

banking application the facts would include the concerned accounts as well as the withdrawal amount. In the case of a supply chain it would be the detected goods and corresponding clients.
Furthermore, it is important to note that the evaluation of the rules by the engine is optimized. That is the engine uses a special algorithm (often a variation of the Rete algorithm, see [8]) to run the rules faster than any procedural system. Thus, the use of rules engines does not only improve the separation of concerns, it also optimizes the rules' evaluation.

Rules Engines and Aspects

As seen before, introducing a rules engine permits a strong separation between the rules and better performances. However, most of the time[3], using this scheme the core business logic still needs to:

1. Store the context (*facts in the working memory*) that may be useful to the rules (using `ruleEngine.store()` instructions).
2. Fire the engine from each module to which rules may apply.

This is where aspects can be of great use. Introducing them in this scheme leaves the core business logic totally unaware of the rule engine. The responsability of both populating the context with facts and firing the engine is left to the aspects. This latter solution is shown on Figure 8. We will now

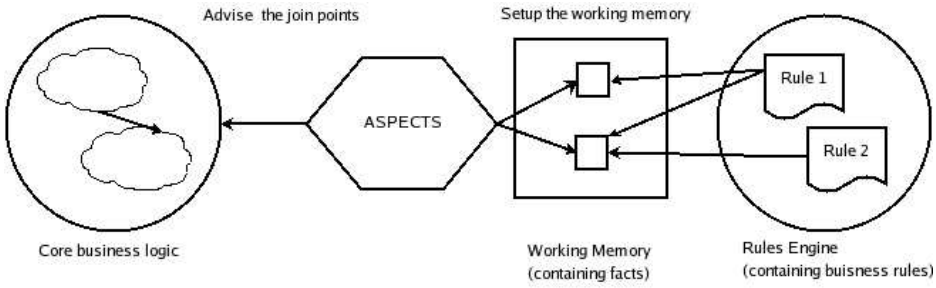

Figure 8.: Using a rules engine and aspects

demonstrate this use by re-modeling the example of Subsection 2.4 in order to use a rules engine.

Expressing the Rules

As mentioned before this example uses the Jess (Java Expert System Shell, see [6]) engine. As this software is JSR 94 compliant it should not cause real troubles to adapt it to some other engines or engines' vendors (such as BEA Systems, Seebeyond & Sun Microsystems or ILOG).
For modeling business rules Jess offers a dedicated LISP-like language. Thus, as a starting point, we model our two business rules (see Subsection 2.2) using this language. Figure 12 provides a transcription of these rules that is to be written to a `.clp` file. Note that we will not provide a full study of these rules

[3]This is not the case if the system is based on an enterprise stack capable of firing rules automatically, see Section 4.

as this implies a good knowledge of the Jess Rules language. A full description of this language (version 7.0) is available on [7].
The first lines (1-15) are used to define the environment to be used by the rules. `bind` functions are used to assign a value to a variable, whereas the `assert` function permits to define a fact. As it can be observed, we are only allowed to work on a limit set of objects, namely only with the objects that where placed in the working memory by the core business logic (or the aspect in our case). These facts, fetched from the working memory, are:

- The account, named `current-account`.
- The withdrawal amount, named `transaction-amount`.
- And a boolean stating whether we are currently in a check clearance transaction, named `checkClearanceTransaction`.

We will see later how these objects are stored to the working memory by the aspect.
The remaining of the listing actually models the rules. We start by the minimum balance rule on lines 16-23. It uses the facts initialized before to check whether the requested amount (`transaction-amount`) overcomes the available amount (line 19). If it is the case an exception is thrown. It is worth noting at this point that nothing needs to be changed conceptually, only the syntax is different.
This fact is also true for the overdraft protection rule, which is scripted on lines 23 to 45. Which achieves the same goal as the `performOverdraftProtection` method in Listing 6. The only difference (besides the syntax) between these rules and the rules using Java only is that the former are already detecting the type of account we are performing the logic one. Indeed, the fine granularity of the pointcuts in the Java and Aspects implementation is replaced here by the instructions `(type savings)` and `(type checking)`. This latter fact is positive as the deciding on what account to apply what rule is conceptually more related to the rule than to the pointcuts.

The Rule Invocation Aspect

Now that our rules are expressed we need to invoke them. As said before, without aspects one needs to invoke them from the core logic. We will, however, use an aspect that advise the join points on the business logic and setup the working memory for the rules engine. The aspect we develop for this purpose is described on Listing 13. We start by initializing the engine with our rules script in the aspect's constructor (lines 11-20). Next, we advise the debit and check clearance transactions (lines 25-43) to call the `invokeRules()` method. In the case of the `checkClearanceDebitExecution` pointcut we invoke it with the `isCheckClearance` flag set to `true` so that the rule engine knows where the method has been called from. To be more precise, this latter flag is used on line 27 of Listing 12.
Eventually, the `invokeRules()` method is interacting with the engine. It starts by *populating the working memory* of the rule engine with objects from the current context in a serie of `_debitRete.store(NAME, VALUE)` instructions (lines 48-52). The next step is to fire the engine by invoking `_debitRete.reset()` and `_debitRete.run()`. Finally, since Jess encapsulates all the business exceptions triggered by the rules into `JessException`s the `invokeRules()` method unwraps them and throws the corresponding `InsufficientBalanceException`s.
Since the rules remain the exact same, running the test script will result in the exact same output as for Subsection 2.4.
Important note: If you install the examples from the official website together with Jess 7 (see Appendix A.3 you will need to patch `Test.java` file. The patching procedure is available in Appendix A.1.

4. Conclusion

"Designing software oriented software is hard, designing *reusable* object oriented software is even harder"[3]

Building reusable software components just as we build reusable hardware components (transistors, gates, etc.) is certainly the dream of every Object Oriented programmer. But the way to genuine reusability is long and filled with obstacles. However, since the dawn of the Service Oriented Architectures takes reusability at a mandatory level it is becoming a real need.

```
(deftemplate account (slot availBalance) (slot type))
(defrule account-existance
  (test (neq (fetch current-account) nil))
  =>
  (bind ?account-object (fetch current-account))
  (bind ?account-avail (call ?account-object getAvailableBalance))
  (if (instanceof ?account-object banking.SavingsAccount) then
    (bind ?account-type savings)
  else (if (instanceof ?account-object banking.CheckingAccount) then
    (bind ?account-type checking)))
  (assert (account (type ?account-type)
    (availBalance ?account-avail)))
  (assert (transaction-amount (fetch transaction-amount)))
  (assert (isCheckClearance (fetch checkClearanceTransaction)))
)
(defrule minimum-balance
  (account (availBalance ?account-avail) (type savings))
  (transaction-amount ?amount)
  (test (< ?account-avail ?amount))
  =>
  (throw (new banking.InsufficientBalanceException
      "Minimum balance condition not met"))
)
(defrule overdraft-protection
  (account (availBalance ?account-avail) (type checking))
  (transaction-amount ?amount)
  (isCheckClearance TRUE)
  (test (< ?account-avail ?amount))
  =>
  (bind ?account-object (fetch current-account))
  (bind ?customer (call ?account-object getCustomer))
  (bind $?overdraft-accounts
    (call (call ?customer getOverdraftAccounts) toArray))
  (bind ?transfer-amount (- ?amount ?account-avail))
  (foreach ?overdraft-account $?overdraft-accounts
    (bind ?overdraft-avail
    (call ?overdraft-account getAvailableBalance))
    (if (< ?transfer-amount ?overdraft-avail) then
      (call ?overdraft-account debit ?transfer-amount)
      (call ?account-object credit ?transfer-amount)
      (return)
    )
  )
  (throw (new banking.InsufficientBalanceException
  "Insufficient funds in overdraft accounts. (according to specified JESS rules)"))
)
```

Listing 12: The rules expressed using Jess' LISP-like language

```
package rule.jess;

import jess.*;
import banking.*;
import rule.common.*;

public aspect RuleEngineBasedDebitRulesAspect extends AbstractDebitRulesAspect {
  private static final float MINIMUM_BALANCE_REQD = 25;

  Rete _debitRete = new Rete();

  public RuleEngineBasedDebitRulesAspect() {
    try {
      _debitRete.executeCommand("(batch rule/jess/debitRules.clp)");
    } catch (JessException ex) {
      System.err.println(ex);
    }
  }

  public float SavingsAccount.getAvailableBalance() {
    return getBalance() - MINIMUM_BALANCE_REQD;
  }

  before(Account account, float withdrawalAmount)
      throws InsufficientBalanceException
  : debitExecution(account, withdrawalAmount) {
    invokeRules(account, withdrawalAmount, false);
  }

  pointcut checkClearanceTransaction()
  : execution(* CheckClearanceSystem.*(..));

  pointcut checkClearanceDebitExecution(Account account,
      float withdrawalAmount)
  : debitExecution(account, withdrawalAmount)
  && cflow(checkClearanceTransaction());

  before(Account account, float withdrawalAmount)
      throws InsufficientBalanceException
  : checkClearanceDebitExecution(account, withdrawalAmount) {
    invokeRules(account, withdrawalAmount, true);
  }

  private void invokeRules(Account account, float withdrawalAmount,
      boolean isCheckClearance) throws InsufficientBalanceException {
    try {
      _debitRete.store("checkClearanceTransaction", new Value(
          isCheckClearance));
      _debitRete.store("current_account", account);
      _debitRete.store("transaction_amount", new Value(withdrawalAmount,
          RU.INTEGER));
      _debitRete.reset();
      _debitRete.run();
    } catch (JessException ex) {
      Throwable originalException = ex.getCause();
      if (originalException instanceof InsufficientBalanceException) {
        throw (InsufficientBalanceException) originalException;
      }
      System.err.println(ex);
    }
  }
}
```

Listing 13: The aspect interfacing the rules engine and the core logic

While re-usability may only seem elegant to lab programmers it becomes a matter of great money when talking about business architectures. Thus, it is no suprise that the idea of decoupling business rules from the core business logic seduced many companies and teams of software developers. Yet, the use of POJOs only to model rules is often not satisfactory enough because the core logic still needs to be aware of the rules to call. This is where AOP enters the game, proposing to wave rules (modeled as Aspects) into a business logic that is completely unaware of the current business guidelines.
While this solution enforces a clean separation of the concerns it still models the rules as sequences of if-then statements. This fact makes it a viable solution for small architectures but a wasteful one (in terms of optimization) for bigger systems.
Rules engines go one step beyond as they offer: *(i)* to isolate completely the rules' definition and execution from the core logic *(ii)* to optimize (when compared Java's if-then statements) the execution of the business rules. Modeling rules with engines is a clean way of doing it, yet not the best. Indeed, with such engines the core logic still needs to populate the working memory with the current context (facts) and fire the engine when needed. Again, AOP proposes a solution to this problem. By using pointcuts and aspects the systems can fire and populate the engine without modifying any existing class. The core logic remains totally unaware of the rules engine which permits a very clean separation of the concerns and prevents a company to depend on a particular engine vendor.
While the value of aspects in business architectures is clear, these are not the only way to achieve a clean separation of the crosscutting concerns. Modern service oriented enterprise architectures often relies on a *business stack* (see Subsection 3.3). Most of these complex set of softwares embed, amongst other components, both orchestration (e.g. BPEL) and rules engines. Using these systems one is able to compose services and trigger rules without the awareness of core components, resulting in the same clean separation as when using aspects.
Thus, AOP in the business field must face a number of competing products from powerful (both in terms of market and technology) competitors. As a consequence aspects are an outsider technology. Nevertheless, this technique has a great advantage over the market stars: aspects are lightweight while business stacks are quite heavy. This fact may seem insignificant but the growing success of Spring (see [9]), the lightweight framework for J2EE applications, is here to prove the opposite.

A. Setting up the Examples

In order to use the various examples presented in this paper you should install the software components listed below.

A.1. Installing the Examples

The examples can be downloaded for free from [2]. In this archive the folder `Ch12` contains the examples used in the paper. The `section12.5` folder contains the examples for Subsection 2.4 while `section12.7` contains the files for Subsection 3.3. To create AspectJ Eclipse projects from these files do as follow:

1. Create a new AspectJ project.
2. Import the files from the filesystem into your project (File → Import)
3. Rightclick on the projects root and select: AspectJ Tools → Convert Files Extensions and select OK.
4. Eventually run the project. You should build and run the whole project, not just the main in order for all the Aspects to be woven with the Java files.

Important note: If you install the examples using Jess from the official website together with Jess 7 (see Appendix A.3) you will need to patch `Test.java` file. To achieve this simply copy the code below and paste it in order to replace the existing `catch` clause of the `RuleEngineBasedDebitRulesAspect.aj` file:

```
catch (JessException ex) {
    // this is the modified instruction —>
      Throwable originalException = ex.getCause();
      if (originalException
    instanceof InsufficientBalanceException) {
    throw
        (InsufficientBalanceException)originalException;
```

```
8         }
          System.err.println(ex);
10    }
```

A.2. Installing Aspect J

First of all install the AspectJ compiler available from [1]. Additionally it is recommended (while not mandatory) to download and install the AspectJ plugin for the Eclipse IDE (AJDT). Note that this plugin is shipped with the AspectJ compiler. Thus, installing it will also install this latter. Setting up the plugin is quite easy. Download and unzip it in Eclipse's root folder. The plugin will be automatically installed and started the next time you launch the IDE.

A.3. Installing Jess 7 (Beta)

The next component to install is Jess (rules engine). However if you want to try the solution with no rules engine (i.e. with aspects only, see 2.4) you do not need to install Jess. Note that Jess is not a free software. You may however request a free license for educational/personal use on [6].
As for AspectJ an Eclipse plugin for Jess is available. To install it perform the following steps:

- Download Jess 7.0b7 (or later) from [5].
- Unzip the archive.
- Copy the content of the Eclipse folder (contained in the Jess 7 archive) to Eclipse's root folder.
- Start Eclipse.
- Import the Jess library (`libjess.jar`) as a Java Build Path library into your existing AspectJ project.

A.4. Running the Examples

To run one of the examples run the project as a whole not just the main file. If you run the file containing the main method only, the aspects will not be woven with the Java files.

A.5. Logging Aspect for Testing the Rules

The code listed on Listing 14 is the aspect used for logging (and testing) the business rules contained in this paper. It is to be used together with the aspects and test methods presented in the previous sections.

```
2  package banking;

4  import logging.*;

6  public aspect LogAccountActivities extends IndentedLogging {
       declare precedence : LogAccountActivities, *;
8
       pointcut accountActivity(Account account, float amount)
10   : ((execution(void Account.credit(float))
       || execution(void Account.debit(float)))
12     && this(account)
       && args(amount))
14     || (execution(void CheckClearanceSystem.*(Account, float))
       && args(account, amount));
16
       protected pointcut loggedOperations()
18   : accountActivity(Account, float);

20     void around(Account account, float amount)
     : accountActivity(account, amount) {
22   try {
         System.out.println("[" +
24           thisJoinPointStaticPart.getSignature().toShortString()
               + "] " + account + " " + amount);
26       System.out.println("Before: " + account.getBalance());
         proceed(account, amount);
28   } finally {
         System.out.println("After: " + account.getBalance());
30   }
     }
32 }
```

Listing 14: The logging aspect for testing the rules

References

[1] Home of the Aspectj Project. [Retrieved June 6, 2006, from http://www.eclipse.org/aspectj/].

[2] Examples from the AspectJ Book. [Retrieved June 5, 2006, from http://www.manning-source.com/books/laddad/laddad_src_Aspectj-In-Action.zip].

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.

[4] D. Guinard. Radio Frequency Identification: Evaluation of the Technology Supporting the Development fo an Assets Tracking Application. Bachelor thesis, Department of Informatics of the University of Fribourg, Switzerland, September 2005. [Retrieved December 22, 2005, from http://diuf.unifr.ch/softeng/student-projects/completed/guinard/download/report_rfid_guinard_unifr.pdf].

[5] Download Page of Jess 7. [Retrieved June 5, 2006, from http://www.jessrules.com/jess/software/].

[6] Jess, the Rule Engine for the Java Platform. [Retrieved June 4, 2006, from http://herzberg.ca.sandia.gov/jess/].

[7] Jess Language Basics. [Retrieved June 4, 2006, from http://www.jessrules.com/jess/docs/70/basics.html].

[8] R. Johnson. A fast match algorithm for the many pattern to many object pattern match problem. *Artificial Intelligence*, 1982.

[9] R. Johnson. Introduction to the Spring Framework. *The Server Side*, May 2005. [Retrieved June 5, 2006, from http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework].

[10] JSR 94: The Java Rule Engine API. [Retrieved June 4, 2006, from http://www.jcp.org/en/jsr/detail?id=94].

[11] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[12] O. Liechti and D. Guinard. RFID Middleware: Ensuring Qualities of Service for the Internet of Objects. In *Software Engineering Today - Building Better Software Faster and Smarter, May 9-10 2006, Zuerich, Switzerland*, pages 403–414. SIGS Datacom, 2006. http://www.sigs.de/download/set_06/ (accessed June 8, 2005).

[13] Sun's Vision for the SOA Platform. 2006.

[14] RFE: Tail Call Optimization. [Retrieved June 3, 2006, from http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4726340].

AOP Tools Comparison

Fabrice Bodmer
Université de Fribourg
Département d'Informatique
Bd de Pérolles 90
CH-1700 Fribourg
fabrice.bodmer@unifr.ch

Timothée Maret
Université de Fribourg
Département d'Informatique
Bd de Pérolles 90
CH-1700 Fribourg
timothee.maret@unifr.ch

Abstract Ce rapport entre dans le cadre d'un séminaire de Master du groupe de recherche en Génie Logiciel de l'Université de Fribourg. L'objectif de ce séminaire est de familiariser ses participants au paradigme de la programmation orientée aspect. L'objet de ce document est l'étude comparative des trois solutions dominantes dans le monde Java pour la programmation AOP, à savoir: AspectJ, JBoss AOP et Spring AOP.

Mots clefs: AOP, AspectJ, JBoss, Spring, AspectWerkz.

1. Introduction

Ce rapport entre dans le cadre d'un séminaire de Master du groupe de recherche en Génie Logiciel de l'Université de Fribourg. L'objectif de ce séminaire est de familiariser ses participants au paradigme de programmation orientée aspect.
Le séminaire est basé principalement sur le livre *AspectJ in Action* de *Ramnivas Laddad* [11] dont certains chapîtres sont présentés en profondeur par les participants. Dans ce rapport, nous allons nous concentrer sur une comparaison des différentes implémentations et des outils de développement permettant d'utiliser l'AOP en Java.

1.1. Article de référence

Notre référence principale pour établir cette comparaison est l'article en deux parties *AOP tools comparison*[5, 6] écrit par *Mik Kersten*. Cet article compare les quatre principaux outils permettant d'utiliser la programmation orientée aspect en Java. Ces outils sont *AspectJ*, *JBoss*, *AspectWerkz* et *Spring*.
La comparaison est effectuée principalement sur la mise en oeuvre des aspects, la syntaxe et les possibilités sémantiques des langages, l'intégration dans les IDE et la documentation.

1.2. Critères de comparaison

Depuis la publication de cet article au début du mois de décembre 2005, des évolutions importantes sont apparues dans la pluspart de ces outils. Le projet *AspectWerkz* a été inclu dans *AspectJ* et ne sera plus développé. Le *Framework Spring* a évolué vers une version majeure qui renforce sa compatibilité avec *AspectJ*. Nous ne suivront donc pas tous les points de comparaison définis dans l'article et nous allons présenter les différentes possibilités d'utilisation et de combinaison de ces outils.

Les points que nous jugeons importants à faire ressortir de ces différents outils sont listés ci-dessous.

Mise en oeuvre des aspects Les aspects peuvent êtres mis en oeuvre de trois manières différentes. La première consiste à *étendre le langage* Java pour déclarer les aspects dans des fichiers spécifiques. La deuxième utilise des *fichiers* `XML`, tandis que la dernière est basée sur les *annotations* de Java 1.5.
Les annotations permettent d'ajouter des méta-données aux classes et aux méthodes. Syntaxiquement, elles sont précédées du caractère `@`. Elles peuvent êtres considérées comme des types particuliers de modificateurs. Ainsi, elle peuvent êtres disposées à tous les emplacements où les modificateurs classiques peuvent être utilisés. Les annotations utilisées par les différents outils sont des annotations sans paramètres qui servent uniquement au marquage. Les annotations sont toujours disponibles dans les fichiers `.class` après la compilation [14].

Fonctionnalités du langage Les mécanismes de base des langages AOP sont la définition de *Pointcut* permettant de sélectionner des points dans le code et d'y appliquer les méthodes définies dans les *Advice*, ainsi que la possibilité de définir de nouveaux membres et de définir les parents d'une classe. Tous les langages étudiés fonctionnent selon ces mécanismes. Cependant, ils permettent différents niveaux de précision et fonctionnalités pour ces concepts de base.

Modes de weaving L'opération de weaving consiste à injecter le code des aspects dans les classes constituant la logique métier. Cette opération peut soit être effectuée dynamiquement en modifiant par exemple le chargeur de classe ou en utilisant les *Proxy dynamiques*, soit statiquement par un compilateur spécifique.

Intégration avec les IDE Les IDE vont nous permettre d'éditer les aspects. Ils offrent une quantité variable de mécanismes d'aide à l'édition.

Domaines d'applications Les outils testés peuvent être utilisés dans deux cas de figures, le développement d'applications classiques, et le développement d'applications destinées à un serveur d'application.

Pour illustrer les domaines d'application des outils, nous tenterons d'implémenter le même programme d'exemple dans chacun d'eux. L'exemple est tiré du livre de *R. Laddad* et met en oeuvre un aspect

permettant de logger toutes les exécutions de méthodes et les exécutions de constructeurs dans un code. Les classes implémentant la logique métier ne vont pas varier tout au long des exemples et sont disponibles en annexe.

2. Présentation des outils

2.1. AspectWerkz

AspectWerkz est un des premiers langage AOP apparu pour le Java, développé par BEA. L'évolution de ce langage l'a rendu de plus en plus proche de *AspectJ* tout en gardant une mise en oeuvre différente. *AspectWerkz* supporte un style de développement basé sur les annotations et des fichiers de description `XML`, et permet le weaving dynamique. Ces spécificités le rendent complémentaire au langage *AspectJ* c'est pouquoi ces deux langages ont fusionné pour donner *AspectJ 5* une évolution majeure du projet d'IBM. Le développement du langage *AspectWerkz* est arrêté à la version 2.0.
Bien que le projet n'évoluera plus, les archives contenant les différentes versions du projet peuvent êtres téléchargées à l'adresse http://dist.codehaus.org/aspectwerkz/[7]

2.2. AspectJ

Ce langage est développé par IBM et supporte une large palette de fonctionnalités permettant de mettre en oeuvre la programmation orientée Aspect en Java dans des application classiques. Au départ, la réalisation d'un programme AOP était basée uniquement sur une extension du langage Java permettant d'écrire les aspects, et un compilateur permettant d'effectuer l'opération de weaving et/ou la compilation. Ce compilateur est disponible sous forme de binaire ou sous forme de classe, ce qui permet de l'intégrer aisément dans des projets[1]. Une évolution importante de ce projet fut la version 1.5 terminée le 20 décembre 2005 et issue de la fusion avec le projet *AspectWerkz*.
Rebaptisée *AspectJ 5* cette version n'étend pas les fonctionnalités du langage, mais offre une nouvelle méthode de mise en oeuvre des aspects par annotations, héritées du projet *AspectWerkz*. Le compilateur `ajc` a aussi bénéficié de cette fusion et la partie permettant d'effectuer le weaving dynamiquement a été améliorée [8].
Cette nouvelle version supporte toutes les nouvelles extensions du langage Java 1.5. tel que les annotations, la généricité, les types énumérés, etc. [1]

Les fonctionnalités du langage *AspectJ* sont très larges. Les Pointcut peuvent êtres associés à l'exécution et l'appel de méthodes et de constructeurs, à l'accès aux champs de classes, la gestion des exception, le control de flux, les classes et les expressions conditionnelles. Les Advice peuvent êtres spécifié avant, après, après le retour, après une exception, ou autour de l'exécution d'une méthode. *AspectJ* dispose de la possibilité de déclarer des warnings ou des erreurs de compilation.

L'outil principal dont dispose *AspectJ* pour éditer les aspects est le plugin *ajdt*. Il permet de gérer et d'éditer des projets *AspectJ* dans *Eclipse*. Les fonctionnalités principales de ce plugin sont la mise en évidence de la syntaxe propre au langage, la détection des erreurs syntaxiques ainsi que le pointage par des flêches des Joinpoint dans le code, qui sont concernés par des Pointcut. Cette fonctionnalité est particulièrement importante et doit permettre d'éviter un grand nombre d'erreur. Le plugin fonctionne avec des aspects décrits avec des annotations ou avec la syntaxe propre à *AspectJ*. Dans le premier cas, le support est moins bon car les erreurs de syntaxes ne sont pas détectées. La Figure 1 nous montre un aperçu du plugin *adjt* lors de l'utilisation.
Le weaver du compilateur ajc fonctionne avec des fichiers `.class` en entrée et produit des fichiers `.class` dans lesquels le bytecode correspondant aux aspects a été ajouté. Cette opération peut être effectuée à trois étapes différentes dans le processus de développement d'une application. Ces trois possibilités sont listées ci-dessous.

- Compile-time weaving
- Post-compile weaving
- Load-time weaving (LTW)

[1] Le plugin Eclipse ajdt utilise la version Java du compilateur.

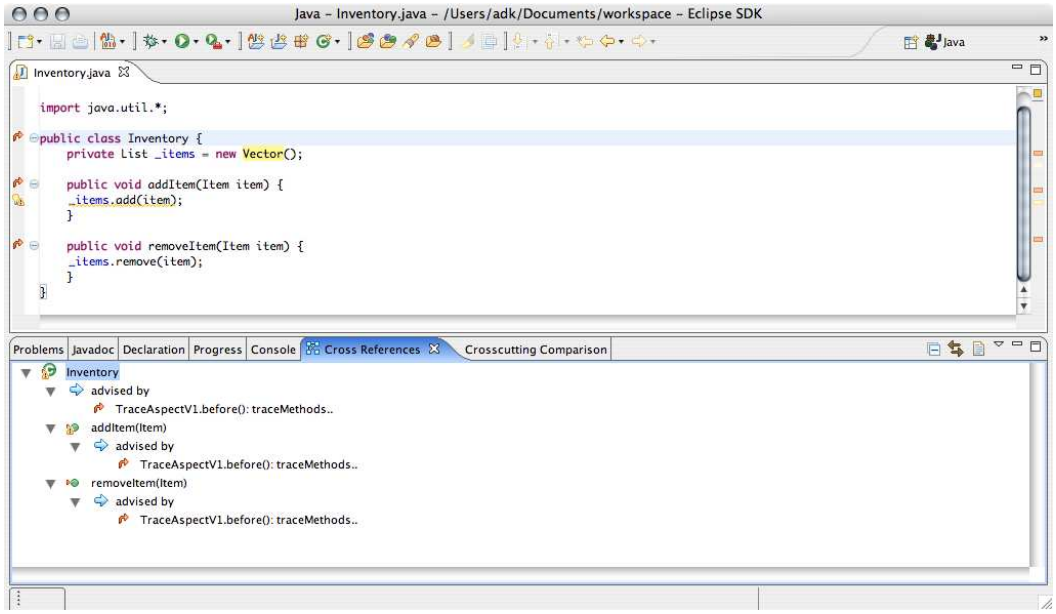

Figure 1.: Aperçu du plugin ajdt en action.

Exemples d'applications

Les trois modes d'utilisation du compilateur ajc vont êtres démontrés dans les exemples suivants. Le Listing 1 montre la classe de test qui sera utilisé pour tous ces trois exemples.

```
//Listing 5.5 A test class

public class Test {
    public static void main(String[] args) {
  Inventory inventory = new Inventory();
  Item item1 = new Item("1", 30);
  Item item2 = new Item("2", 31);
  Item item3 = new Item("3", 32);
  inventory.addItem(item1);
  inventory.addItem(item2);
  inventory.addItem(item3);
  ShoppingCart sc = new ShoppingCart();
  ShoppingCartOperator.addShoppingCartItem(sc, inventory, item1);
  ShoppingCartOperator.addShoppingCartItem(sc, inventory, item2);
    }
}
```

Listing 1: Listing `Test.java` [11]

Compile-time weaving Cette méthode est la plus simple à mettre en oeuvre c'est une méthode statique qui utilise uniquement les fichiers sources du programme. Le compilateur ajc est utilisé et va procéder en deux opérations, la première consistant à compiler l'application, la deuxième consistant à effectuer l'opération d'injection du code correspondant aux aspects.
Le code de l'aspect est listé ci-dessous. Nous pouvons remarquer qu'il est contenu dans un fichier portant l'extension `.java` mais dont le contenu n'est pas une classe mais un aspect.

```
import org.aspectj.lang.*;

public aspect TraceAspectV1 {
    pointcut traceMethods()
  : (execution(* *.*(..))
     || execution(*.new(..))) && !within(TraceAspectV1);

    before() : traceMethods() {
  Signature sig = thisJoinPointStaticPart.getSignature();
  System.out.println("Entering ["
        + sig.getDeclaringType().getName() + "."
        + sig.getName() + "]");
    }
```

```
14 }
```

Listing 2: Listing `TraceAspectV1.java`

Les commandes suivantes sont utilisées pour compiler et lancer l'application. Nous avons ajouté le binaire ajc dans notre `PATH` et nous ajoutons les librairies dans le `CLASSPATH` manuellement.

```
ajc -classpath .:/Applications/AspectJ1.5/lib/aspectjrt.jar -1.5 *.java
java -classpath .:/Applications/AspectJ1.5/lib/aspectjrt.jar Test
```

La sortie du programme est listée ci-cessous. Nous pouvons remarquer que toutes les exécution de méthodes, ainsi que tous les exécution de constructeurs sont biens loggées, comme défini dans l'aspect.

```
Entering [Test.main]
Entering [Inventory.<init>]
Entering [Item.<init>]
Entering [Item.<init>]
Entering [Item.<init>]
Entering [Inventory.addItem]
Entering [Inventory.addItem]
Entering [Inventory.addItem]
Entering [ShoppingCart.<init>]
Entering [ShoppingCartOperator.addShoppingCartItem]
Entering [Inventory.removeItem]
Entering [ShoppingCart.addItem]
Entering [ShoppingCartOperator.addShoppingCartItem]
Entering [Inventory.removeItem]
Entering [ShoppingCart.addItem]
```

Post-compile weaving Cette méthode permet d'injecter du code associé à un aspect lorsque les fichiers sont déjà compilés. Les fichiers peuvent êtres contenus soit dans des `.class` soit dans des fichiers `.jar`. L'ensemble des annotations *@AspectJ* est utilisé. En utilisant ces annotations, le programme pourra être compilé avec un compilateur classique, puis l'opération de weaving sera effectuée soit dynamiquement dans un class-loader modifié, soit statiquement. La syntaxe des annotations et leur mise en oeuvre est décrite dans la documentation [9].
Nous listons ci-dessous, le fichier implémentant l'aspect du programme de test en utilisant les annotations.

```
import org.aspectj.lang.*;
2 import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
4 import org.aspectj.lang.annotation.Pointcut;

6 @Aspect
public class TraceAspectV1Annotations {
8
    @Pointcut("execution(* *.*(..)) || execution(*.new(..)) && !within(TraceAspectV1Annotations)")
10    void traceMethods() {}

12    @Before("traceMethods()")
    public void process(JoinPoint.StaticPart thisJoinPointStaticPart) {
14      Signature sig = thisJoinPointStaticPart.getSignature();
      System.out.println("Entering ["
16            + sig.getDeclaringType().getName() + "."
            + sig.getName() + "]");
18    }
}
```

Listing 3: Listing `TraceAspectV1Annotations.java`

Le code est contenu dans une classe normale et nous pouvons remarquer que la structure globale ne change pas. Les annotations *@Aspect*, *@Pointcut* et *@Before* sont utilisées pour remplacer la syntaxe Java spécifique à *AspectJ*. Dès lors, le code pourrait se compiler sans problème avec le compilateur `javac`, cependant l'exécution ne donnerait aucun résultat car l'opération de weaving ne serait pas encore effectuée. Dans Eclipse, les mécanismes indiquant par des flêches, les points ou seront insérés le code sont toujours disponibles. Cependant les tests syntaxiques sont moins puissants car les erreurs syntaxiques dans les pointcut ne sont pas détectées.
Les commandes suivantes sont utilisées pour compiler et lancer l'application. Nous avons ajouté le binaire ajc dans notre `PATH` et nous ajoutons les librairies dans le `CLASSPATH` manuellement.

```
javac -g -classpath .:/Applications/AspectJ1.5/lib/aspectjrt.jar *.java
ajc -classpath .:/Applications/AspectJ1.5/lib/aspectjrt.jar -1.5 -inpath . -aspectpath .
java -classpath .:/Applications/AspectJ1.5/lib/aspectjrt.jar Test
```

La sortie du programme correspond au listing de l'exemple précédent ce qui montre que le programme a toujours le comportement désiré.

Load-Time Weaving Cette méthode permet d'injecter le code associé aux aspect, durant la phase de chargement de classe dans le class loader. Le chargeur de classe est adapté par un agent gérant la modification des classes lorsqu'elles sont chargées. *AspectJ* a toujours supporté un mode load-time weaving dans son framework, cependant depuis la version 1.5, l'intégration est plus aisée et le développement se base uniquement sur un fichier de configuration `aop.xml`. Ce fichier contient la déclaration des aspects devants êtres appliqués sur les classes. Il est aussi possible de réaliser des aspects abstraits directement dans le fichier `XML`. Cette fonctionnalité permet par exemple d'adapter un aspect en modifiant uniquement la définition des *Pointcut* dans le fichier `XML`, sans toucher à un code Java. *AspectJ 5* définit plusieurs méthodes pour faire du weaving dynamique:

- Activation par des agents. Ces agents sont spécifiques au JRE utilisé. Il existe des agents pour les JVM compatibles avec les spécifications de SUN, et les JVM de JRockit.
- Lancement par la commande *aj* qui va exécuter le programme dans un environnement Java 1.4 ou suppérieur.
- Ecrire notre propre chargeur de classes en utilisant les interfaces fournies et permettant d'effectuer les opérations de weaving.

Nous allons développer un exemple mettant en oeuvre les agents. Nous modifions l'aspect du Listing 2 pour le rendre abstrait en rendant le *Pointcut* abstrait. Le Listing 4 montre la définition abstraite de l'aspect.

```
import org.aspectj.lang.*;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public abstract class TraceAspectV1AbstractAnnotations {

    @Pointcut
    abstract void traceMethods();

    @Before("traceMethods()")
    public void process(JoinPoint.StaticPart thisJoinPointStaticPart) {
      Signature sig = thisJoinPointStaticPart.getSignature();
      System.out.println("Entering ["
             + sig.getDeclaringType().getName() + "."
             + sig.getName() + "]");
    }
}
```

Listing 4: Listing `TraceAspectV1Abstract.java`

Nous définissons ensuite le fichier `META-INF/aop.xml` [2] dans lequel nous allons réaliser l'aspect abstrait en spécifiant le *Pointcut*. La structure des fichiers de description est composé de deux parties principales, un élément qui définit l'ensemble des aspects à êtres utilisés durant la phase de weaving, et un élément contenant les différentes configurations du weaver. Nous avons ajouté une option pour le weaver afin qu'il n'affiche pas les messages de Warning.

```
<aspectj>
    <aspects>
        <!-- definition de l aspect -->
        <concrete-aspect name="TraceAspectV1" extends="TraceAspectV1AbstractAnnotations">
            <pointcut name="traceMethods" expression="execution(* *.*(..)) || execution(*.new(..))" />
        </concrete-aspect>
    </aspects>
    <weaver options="-nowarn" />
```

[2] Par défaut, le fichier de configuration doit être disposé dans le répertoire `META-INF` et se nommer `aop.xml`

```
</aspectj>
```

Listing 5: Listing aop.xml

Pour utiliser ce mode de weaving, la librairie `aspectjweaver.jar` doit être ajoutée pour l'exécution. Cette librairie contient le weaver, les agents, le chargeur de classe, ainsi qu'une DTD pour parser le fichier de configuration. Les commandes pour la compilation et l'exécution sont listées ci-dessous. Pour l'exécution il faut spécifier le chemin vers la librairie `aspectjweaver.jar`.

```
javac -g -classpath .:/Applications/AspectJ1.5/lib/aspectjrt.jar *.java
java -javaagent:/Applications/AspectJ1.5/lib/aspectjweaver.jar
     -classpath .:/Applications/AspectJ1.5/lib/aspectjrt.jar Test
```

La sortie du programme correspond au listing du premier exemple, ce qui montre que le programme a toujours le comportement désiré.

Fonctionnement du weaver ajc

Le fonctionnement du weaver ajc n'est pas clairement défini dans la littérature. L'article *Advice Weaving in AspectJ* [3] décrit cependant globalement l'implémentation de l'opération de weaving dans *AspectJ 1.1* et présente des tests de performances. Nous avons utilisé le décompilateur jad [10] pour observer le code généré. Nous listons uniquement les fichiers décompilés de `Item.class` et `TraceAspectV1.class` car les autres fichiers n'apportent pas plus d'information.

Nous pouvons observer sur le Listing 7 que comme présenté dans l'article, l' *Advice* de l'aspect est convertit en une méthode correspondante avec les mêmes paramètres et l'ajout du paramètre `thisJoinPointStaticPart` car cet objet est utilisé dans le code de la méthode. Nous pouvons remarquer que la méthode possède le nom modifié `ajc$before$TraceAspectV1$1$b314f86e`. Ces attributs vont permettre de référencer les *Pointcut* associés à cet *Advice* et de stocker des informations additionnelles associées à l'objet `thisJoinPointStaticPart`.
Nous pouvons remarquer que l'aspect est implémenté en utilisant le design pattern *Singleton* dont l'instance locale est accédée par la méthode statique `aspectOf()`. L'initialisation de cette instance est effectuée au chargement de la classe dans un bloc `static`.

Le Listing 6 nous permet d'observer le code injecté dans cette classe en fonction de l'aspect. L'aspect définit qu'avant chaque exécution de méthode ou de constructeur, l'*Advice* doit être exécuté. Nous pouvons vérifier que la méthode est bien appelée comme première instruction de chaque méthode et du constructeur.
Au chargement de la classe, une instance de `thisJoinPointStaticPart` est créé pour chaque point d'exécution ou sera appelé l'advice. Ces instances contiennent les informations sur le type de *Join Point* et les informations associées.

```
import org.aspectj.runtime.reflect.Factory;
public class Item{
    private java.lang.String _id;
    private float _price;
    private static final org.aspectj.lang.JoinPoint.StaticPart ajc$tjp_0;
    private static final org.aspectj.lang.JoinPoint.StaticPart ajc$tjp_1;
    private static final org.aspectj.lang.JoinPoint.StaticPart ajc$tjp_2;
    private static final org.aspectj.lang.JoinPoint.StaticPart ajc$tjp_3;
    static {
        org.aspectj.runtime.reflect.Factory factory = new Factory("Item.java", java.lang.Class.forName("
            Item"));
        ajc$tjp_0 = factory.makeSJP("constructor-execution", factory.makeConstructorSig("1--Item-java.lang.
            String:float:-id:price:--"), 7);
        ajc$tjp_1 = factory.makeSJP("method-execution", factory.makeMethodSig("1-getID-Item---java.lang.
            String-"), 12);
        ajc$tjp_2 = factory.makeSJP("method-execution", factory.makeMethodSig("1-getPrice-Item---float-"),
             16);
        ajc$tjp_3 = factory.makeSJP("method-execution", factory.makeMethodSig("1-toString-Item---java.lang
            .String-"), 20);
    }
    public Item(java.lang.String id, float price){
        TraceAspectV1.aspectOf().ajc$before$TraceAspectV1$1$b314f86e(ajc$tjp_0);
        _id = id;
        _price = price;
    }
    public java.lang.String getID(){
```

```
22          TraceAspectV1.aspectOf().ajc$before$TraceAspectV1$1$b314f86e(ajc$tjp_1);
            return _id;
24      }
        public float getPrice(){
26          TraceAspectV1.aspectOf().ajc$before$TraceAspectV1$1$b314f86e(ajc$tjp_2);
            return _price;
28      }
        public java.lang.String toString(){
30          TraceAspectV1.aspectOf().ajc$before$TraceAspectV1$1$b314f86e(ajc$tjp_3);
            return "Item: " + _id;
32      }
    }
```

Listing 6: Listing `Item.class`

```
    import java.io.PrintStream;
2   import org.aspectj.lang.NoAspectBoundException;
    import org.aspectj.lang.Signature;
4   public class TraceAspectV1{
        private static java.lang.Throwable ajc$initFailureCause;
6       public static final TraceAspectV1 ajc$perSingletonInstance;
        static {
8           try{
                TraceAspectV1.ajc$postClinit();
10          }
            catch (java.lang.Throwable throwable){
12              ajc$initFailureCause = throwable;
            }
14      }
        public TraceAspectV1(){}
16      public void ajc$before$TraceAspectV1$1$b314f86e(org.aspectj.lang.JoinPoint.StaticPart
             thisJoinPointStaticPart){
            org.aspectj.lang.Signature sig = thisJoinPointStaticPart.getSignature();
18          java.lang.System.out.println("Entering [" + sig.getDeclaringType().getName() + "." + sig.getName()
                 + "]");
        }
20      public static TraceAspectV1 aspectOf(){
            if (ajc$perSingletonInstance == null)
22              throw new NoAspectBoundException("TraceAspectV1", ajc$initFailureCause);
            else
24              return ajc$perSingletonInstance;
        }
26      public static boolean hasAspect(){
            return ajc$perSingletonInstance != null;
28      }
        private static void ajc$postClinit(){
30          ajc$perSingletonInstance = new TraceAspectV1();
        }
32  }
```

Listing 7: Listing `TraceAspectV1.class`

2.3. Spring Framework

Spring [15] est un *Framework* open source défini comme un conteneur léger [2]. Le *Framework* offre les fonctionnalités d'un serveur d'application tout en garantissant une très grande souplesse de développement grâce à sa conception modulaire. Cet environnement permet de créer des applications serveurs basées sur des *POJOS*[3] [2].
La Figure 2 nous montre les sept principaux modules du *Framework*. Le *Framework* est basé sur le module "Spring Core" qui contient le container *IoC* [4]. Ce conteneur va permettre de gérer les objets créés lors de l'exécution. Ces objets sont nommés *bean* en références au *JavaBean* de *J2EE*. Le conteneur est paramétré par un fichier `XML` permettant de spécifier comment instancier, configurer et assembler les *beans*.
Pratiquement, les modules sont distribués sous forme de `.jar` que nous pouvons inclure dans le `CLASSPATH` en fonction des besoins et des dépendances imposées. Nous allons porter notre attention sur le module *Spring AOP* associé au module *Spring Core*.
Une évolution majeure de ce *Framework* est la version 2.0 lancée durant le mois de décembre 2006. Cette nouvelle version apporte un grand nombre d'améliorations dans tous les modules dont la plus marquante est l'introduction de nouveaux `namespace` permettant de simplifier l'écriture des fichiers de configuration. Pour ce rapport, nous utilisons la version `2.0 m5` lancée le 2 juin 2006, qui fournit une documentation

[3] Plain Old Java ObjectS
[4] Inversion Of Control

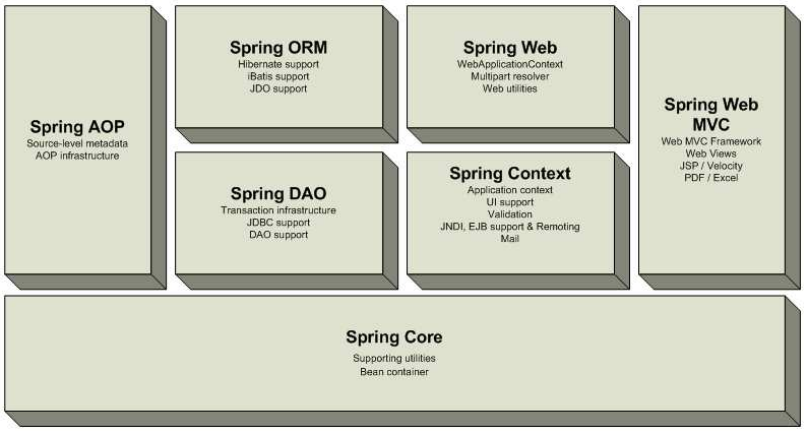

Figure 2.: Aperçu des modules du Framework Spring. [15]

complète. Nous avons téléchargé le fichier `spring-framework-2.0-m5-with-dependencies.zip [53Mo]` contenant toutes les classes, les sources du *Framework*, la documentation ainsi que de nombreuses libraires utilitaires développées dans d'autres projets. La documentation est également disponible sur le web [15].

Spring AOP

L'objectif des développeurs du module *Spring AOP* n'est pas de fournir un ensemble complet des fonctionnalités disponibles dans la programmation orientée aspect. Leur objectif est de fournir une des possibilités permettant de répondre à la plus-part des besoins dans ce domaine et de garantir une compatibilité avec des outils plus complets. Concrètement, les Pointcut peuvent uniquement définir un ensemble de Joinpoint associés à l'exécution de méthodes. Les évènements sur les champs de classes (set,get), les constructeurs, le flux, etc. disponibles dans *AspectJ* ne sont pas disponibles. L'injection de nouveaux champs et la possibilité de définir les parents de la classes sont disponbiles. Les applications utilisant *Spring AOP* sont écrites entièrement en Java et aucune phase de compilation n'est nécessaire car *Spring AOP* permet uniquement le weaving dynamique.

La compatibilité avec le langage et les outils d'*AspectJ* est donc développée, tout en restant optionnelle. *Spring 2.0* a introduit deux nouvelles manières permettant de définir des aspects.

- Définition des aspects dans le fichier `XML` de configuration en utilisant les nouveaux `namespace`. Ce procédé est conseillé pour les applications utilisant une version antérieure à *Java 5*
- Définition des aspects par les annotations introduites par *@AspectJ*. Ce procédé est conseillé pour les applications utilisant *Java 5* et permet d'utiliser les fichiers sources avec le compilateur et les weaver dynamique d'*AspectJ*.

Ces deux nouvelles méthodes utilisent la syntaxe des *pointcut* définie par *AspectJ* et utilisent les librairies d'*AspectJ 5* pour parser les *pointcut* et pour effectuer les correspondances avec les *joinpoint*. Lorsque *Spring AOP* est utilisé pour le weaving, les fonctionnalités sont toujours limitées bien que la mise en oeuvre est similaire à *AspectJ*. Lorsque les annotations sont utilisées, il est possible d'utiliser le weaver d'*AspectJ* et le compilateur *ajc* pour profiter des fonctionalités étendues d'*AspectJ* associées au conteneur *Spring*. Cette association permet de créer de puissantes application destinées au serveur d'application *Spring*.

Spring AOP implémente l'*AOP* grâce aux fonctionnalités de *Proxy Dynamique* offerte par le *JDK* [5] soit par la librairie *CGLIB* [6]. Les *Proxy Dynamiques* sont des objets qui implémentent plusieurs interfaces dynamiquement. Les appels de méthodes sur un objets *Proxy* sont délégués automatiquement sur des

[5] http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html
[6] http://cglib.sourceforge.net/

instances gérées par l'objet *Proxy*. Ce mécanisme est totalement transparent pour un objet qui utilise une méthode d'un objet *Proxy*.
Dans *Spring*, chaque objet sur lequel des pointcut vont avoir un effet vont êtres instanciés sous forme de *AOP Proxy* par le conteneur *IoC* et chaque appel de méthode pourra être intercepté pour exécuter le code à injecter.
Par défaut, les *Proxy Dynamiques* du *JDK* seront utilisés. Ces *Proxy Dynamiques* ne peuvent représenter que des interfaces et non pas des classes qui n'implémentent pas d'interface. Dans ce cas, la librairie *CGLIB* peut être utilisée.

Les fonctionnalités du langage *Spring AOP* sont très restreintes. Les Pointcut ne peuvent peuvent êtres associés qu'à l'exécution de méthodes non statiques et au control de flux.

Spring AOP ne dispose pas d'un outil performant permettant d'éditer les aspects. Un projet ayant pour but la création d'un plugin pour *Eclipse* est cependant en cours. Les fonctionnalités sont actuellement limitées à la reconnaissance de la syntaxe du fichier de configuration `XML` et à la gestion de projets *Spring*. Cependant, lorsque nous l'avons installé, cette fonctionnalité s'est avérée n'apporter aucune aide pour le développement.

Exemples d'applications

Le premier exemple montre l'utilisation de *Spring AOP* seul, tandis que le deuxième nous montre l'association entre le container *Spring* et *AspectJ*. Nous ne montrons pas d'exemple de mise en oeuvre de *Spring AOP* utilisant *AspectJ* pour les Joinpoint ou les nouveaux `namespace` car les fonctionnalités sont toujours limitées au langage de *Spring AOP*, les améliorations portent uniquement sur la syntaxe.

Pour ces deux exemples, la librairie *commons-logging.jar* doit être ajoutée au `CLASSPATH` pour l'exécution car elle est implicitement utilisée par le *Framework Spring* pour logger ses activités.

Spring AOP Cet exemple utilise uniquement *Spring AOP* pour le weaving et l'implémentation. Le Listing 8 nous montre la classe implémentant l'interface `MethodBeforeAdvice`. Dans cet exemple, l'objet `JoinPoint` disponible dans *AspectJ*, n'est naturellement pas disponible. Mais la méthode qui sera sélectionnée par le JoinPoint sera passée en paramètre par le *Framework* et va permettre de retirer les informations désirées pour le logging.

```
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class Trace implements MethodBeforeAdvice{
  public void before(Method method, Object[] arg1, Object arg2) throws Throwable {
    System.out.println("Entering ["
      + method.getDeclaringClass() + "."
      + method.getName() + "]");
  }
}
```

Listing 8: Listing `TraceBase.java`

Le Listing 9 nous montre la classe de test utilisée. Cette classe doit être modifiée par rapports aux exemples d'*AspectJ* pour que les objets soient obtenus en passant par le container. La première étape est donc de créer le container en fonction du fichier de configuration.

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring {
  public static void main(String[] args) {
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("./loggingaspectbase.xml");
    Inventory inventory = (Inventory)context.getBean("inventory");
    Item item1 = (Item)context.getBean("item1");
    Item item2 = (Item)context.getBean("item2");
    Item item3 = (Item)context.getBean("item3");
    ShoppingCart sc = (ShoppingCart)context.getBean("sc");
    inventory.addItem(item1);
    inventory.addItem(item2);
    inventory.addItem(item3);
    ShoppingCartOperator.addShoppingCartItem(sc, inventory, item1);
    ShoppingCartOperator.addShoppingCartItem(sc, inventory, item2);
  }
```

```
}
```

Listing 9: Listing `TestSpring.java`

Le Listing 10 nous montre le fichier de configuration nécessaire pour implémenter cet exemple. Il contient la définition des *Proxy* que nous utilisons dans l'application. Ces beans sont des *Proxy* qui gèrent deux objects particulier, la cible qui est l'instance gérée par le *Proxy* et l'intercepteur qui est un bean englobant l'advice (méthode à exécuter) et un pattern permettant de sélectionner. Le langage utilisé pour définir les méthodes à capturer est basé sur des `REGEXP`. Le pattern spécifié correspond au pattern le moins sélectif possible car il permet de sélectionner toutes les exécutions de méthodes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- template from http://www.programmersheaven.com/2/AOP-Spring-Instro-Part1-P2 -->
<beans>
  <!--CONFIG-->
  <bean id="inventory" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
      <ref local="beanInventory"/>
    </property>
    <property name="interceptorNames">
      <list>
        <value>trace</value>
      </list>
    </property>
    <property name="optimize"><value>true</value></property>
  </bean>
  <bean id="item1" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
      <ref local="beanItem1"/>
    </property>
    <property name="interceptorNames">
      <list>
        <value>trace</value>
      </list>
    </property>
    <property name="optimize"><value>true</value></property>
  </bean>
  <bean id="item2" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
      <ref local="beanItem2"/>
    </property>
    <property name="interceptorNames">
      <list>
        <value>trace</value>
      </list>
    </property>
    <property name="optimize"><value>true</value></property>
  </bean>
  <bean id="item3" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
      <ref local="beanItem3"/>
    </property>
    <property name="interceptorNames">
      <list>
        <value>trace</value>
      </list>
    </property>
    <property name="optimize"><value>true</value></property>
  </bean>
  <bean id="sc" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
      <ref local="beanSc"/>
    </property>
    <property name="interceptorNames">
      <list>
        <value>trace</value>
      </list>
    </property>
    <property name="optimize"><value>true</value></property>
  </bean>
  <!--CLASS-->
  <bean id="beanInventory" class="Inventory"/>
  <bean id="beanItem1" class="Item"/>
  <bean id="beanItem2" class="Item"/>
  <bean id="beanItem3" class="Item"/>
  <bean id="beanSc" class="ShoppingCart"/>
  <!-- Advisor pointcut definition for before advice -->
  <bean id="trace" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <ref local="theTrace"/>
```

```
    </property>
72  <property name="pattern">
      <value>.*</value> <!-- a . matches everything except \n -->
74  </property>
  </bean>
76 <!--ADVICE-->
  <bean id="theTrace" class="Trace"/>
78 </beans>
```

Listing 10: Listing `loggingaspectbase.xml`

Les commandes suivantes peuvent êtres utilisées pour compiler et exécuter l'application. Les librairies suivantes doivent êtres ajoutée dans le `CLASSPATH`.

- `cglib-nodep-2.1_3.jar` Les classes de notre application n'implémentent pas d'interface.
- `spring.jar` Cette archive contient la plus part des modules[7] du *Framework Spring*.

```
javac -g -classpath .:/Applications/spring/dist/spring.jar *.java
java -classpath .:/Applications/spring/dist/spring.jar:
                  /Applications/spring/lib/jakarta-commons/commons-logging.jar:
                  /Applications/spring/lib/cglib/cglib-nodep-2.1_3.jar TestSpring
```

Aprés l'exécution du programme, nous obtenons la sortie suivante sur la sortie standard[8].

```
...
Entering [class Inventory.addItem]
Entering [class Inventory.addItem]
Entering [class Inventory.addItem]
Entering [class Inventory.removeItem]
Entering [class ShoppingCart.addItem]
Entering [class Inventory.removeItem]
Entering [class ShoppingCart.addItem]
```

Nous pouvons remarquer que les méthodes statiques ne sont pas capturées, ce qui s'explique par le fait qu'il n'existe pas de *Proxy* associé à ces appels de méthode. Nous pouvons remarquer que les appels aux constructeurs ne sont pas non plus capturés.

Cette mise en oeuvre est la plus basique disponible avec ce *Framework*. Elle a le désavantage d'imposer un long fichier de configuration et d'utiliser des REGEXP pour la définition des méthodes à sélectionner.

Spring/AspectJ Cet exemple montre l'utilisation combinée du container *Spring* et du weaver dynamique d'*AspectJ*. Les classes de l'application ne sont toujours pas modifiées. L'aspect implémenté avec des annotations peut être réutilisé sans modifications, il correspond au Listing 3. Les classes de l'application sont misent dans un package particulier, ce qui nous permet de spécifier au weaver dynamique de ne traiter que les classes de ce package. Ainsi, nous évitons de capturer tous les appels de méthodes générés par le conteneur *Spring*.
Cette mise en oeuvre se base sur deux fichiers de configuration `XML`. Le Listing 11 montre le contenu du fichier permettant de configurer le weaver en spécifiant les aspects à injecter ainsi que les classes sur lesquelles les pointcut seront testés. Le Listing 12 permet de spécifier les beans qui seront instanciés par le *Framework*. Un bean est créé pour chaque instance nécessaire dans la classe de test.
La classe de Test est similaire au Listing 9, seul le nom du fichier de configuration `XML` change.

```
<!DOCTYPE aspectj PUBLIC
2   "-//AspectJ//DTD//EN"    "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>
4   <weaver>
      <include within="ch.unifr.aop.dynamic..*"/>
```

[7] Spring est composé de nombreux jar correspondants aux différentes fonctionnalités désirées. Ces jars sont disponibles dans le répertoire `dist/modules/spring-*.jar`
[8] Les logs effectués par le *Framework* ont été enlevés et remplacés par "..." pour une plus grande clarté

```
  </weaver>
  <aspects>
    <aspect name="ch.unifr.aop.dynamic.TraceAspectV1Annotations"/>
  </aspects>
</aspectj>
```

Listing 11: Listing `aop.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop.xsd">
  <bean id="inventory" class="ch.unifr.aop.dynamic.Inventory"/>
  <bean id="item1" class="ch.unifr.aop.dynamic.Item"/>
  <bean id="item2" class="ch.unifr.aop.dynamic.Item"/>
  <bean id="item3" class="ch.unifr.aop.dynamic.Item"/>
  <bean id="sc" class="ch.unifr.aop.dynamic.ShoppingCart"/>
</beans>
```

Listing 12: Listing `loggingaspect.xml`

Les commandes suivantes peuvent êtres utilisées pour compiler et exécuter l'application. La librairie `cglib-nodep-2.1_3.jar` n'a pas besoin d'être ajoutée puis-ce que le weaving sera effectué par *AspectJ*.

```
javac -classpath .:/Applications/spring/lib/aspectj/aspectjrt.jar:
                   /Applications/spring/dist/spring.jar -g ch/unifr/aop/dynamic/*.java
java -javaagent:/Applications/spring/lib/aspectj/aspectjweaver.jar
     -classpath .:/Applications/spring/lib/aspectj/aspectjrt.jar:
                  /Applications/spring/dist/spring.jar:
                  /Applications/spring/lib/jakarta-commons/commons-logging.jar
                  ch.unifr.aop.dynamic.Test
```

Aprés l'exécution du programme, nous obtenons la sortie suivante sur la sortie standard.

```
Entering [ch.unifr.aop.dynamic.Test.main]
...
Entering [ch.unifr.aop.dynamic.Inventory.<init>]
Entering [ch.unifr.aop.dynamic.Item.<init>]
Entering [ch.unifr.aop.dynamic.Item.<init>]
Entering [ch.unifr.aop.dynamic.Item.<init>]
Entering [ch.unifr.aop.dynamic.ShoppingCart.<init>]
Entering [ch.unifr.aop.dynamic.Inventory.addItem]
Entering [ch.unifr.aop.dynamic.Inventory.addItem]
Entering [ch.unifr.aop.dynamic.Inventory.addItem]
Entering [ch.unifr.aop.dynamic.ShoppingCartOperator.addShoppingCartItem]
Entering [ch.unifr.aop.dynamic.Inventory.removeItem]
Entering [ch.unifr.aop.dynamic.ShoppingCart.addItem]
Entering [ch.unifr.aop.dynamic.ShoppingCartOperator.addShoppingCartItem]
Entering [ch.unifr.aop.dynamic.Inventory.removeItem]
Entering [ch.unifr.aop.dynamic.ShoppingCart.addItem]
```

Nous pouvons constater que le listing correspond au listing obtenus lors des tests avec *AspectJ*. Le weaving s'est opéré de manière dynamique par le weaver d'*AspectJ*, sur des objets instanciés par le conteneur *Spring*. Cet exemple démontre l'intégration poussée des deux technologies.

2.4. JBoss AOP

JBoss, entreprise récemment acquise par *Red Hat Inc*, est connue pour son serveur d'application J2EE *open-source*. Entre autres apports aux technologies *Java*, cette entreprise a développé un environnement permettant le support de la programmation orientée aspect en java, de manière autonome dans un environnement de programmation quelconque, ou étroitement intégré au serveur d'application *JBoss*

Application Server.

JBoss étant une entreprise à vocation commerciale, et donc désireuse de satisfaire sa clientèle en offrant des produits dont la pérennité est assurée à long terme, elle a choisi de développer la couche orientée aspect en se conformant au langage Java tel que spécifié par *Sun*, autrement dit, sans toucher à la syntaxe du langage. Pour exécuter, compiler, éditer, etc. un programme java s'appuyant sur *JBoss AOP*, les outils standards conviennent.

En outre, il convient de préciser que *JBoss AOP* n'est pas seulement un environnement, mais également un ensemble d'aspects réutilisables. Sont fournis notamment des aspects pour la gestion de la mise en cache, de la communication asynchrone, des transactions, des aspects liés à la sécurité, etc.

Implémentation des aspects

Les possibilités offertes par JBoss AOP pour la formalisation des *join points* et l'implémentation des aspects sont similaires à celles offertes par AspectJ. Dans cette section les possibililtés offertes par JBoss AOP sont décrites et comparées à AspectJ. On pourra ainsi se faire une idée de la richesse du produit analysé.

Pointcuts Les *pointcuts* identifient les *join points* dans le flux du programme. Un *pointcut* est composé d'une expression décrivant la structure que l'on souhaite identifier dans le flux du programme. Ainsi, comme pour AspectJ, il existe des expressions (signatures) identifiant des types, des méthodes, des constructeurs, et des champs. La syntaxe de ces expressions ne change que peu par rapport à celle employée dans AspectJ. Selon la documentation de référence officielle [4], elle semble toutefois quelque peu moins riche. Ainsi il manque par exemple le caractère de remplacement `+` remplaçant une interface ou une sous-classe quelconque. En revanche, les caractères de remplacement suivants sont communs à AspectJ et JBoss AOP:

- `*` Remplace zéro ou plusieurs caractères.
- `..` Remplace zéro ou plusieurs paramètres.

Il existe toute une palette de *pointcuts* permettant de lier un point identifié par une expression (signature) dans le flux du programme. Le tableau 2.4 dresse la liste des *pointcuts* disponibles dans JBoss AOP. On peut remarquer que comparé à AspectJ, il y en a moins. Manquent notamment les *pointcuts* basés sur le contrôle du flux du programme (control-flow pointcuts). Les mêmes possibilités existent mais c'est n'est pas un *pointcut* de base. Il n'est en revanche pas possible de gérer les exceptions directement par la déclaration du *pointcut*, le traitement des exceptions ne peut se faire que dans le corps de l'*advice*.

Pointcut	Expression
`execution`	methode, constructeur
`get`	champ
`set`	champ
`field`	champ
`all`	type
`call`	methode, constructeur
`within`	type
`withincode`	methode, constructeur
`has`	methode, constructeur
`hasfield`	champ

Table 1.: Liste des *pointcuts* disponibles dans JBoss AOP

Les *pointcuts* peuvent être composés. Plusieurs opérateurs logiques sont disponibles:

- `!` le non logique.
- `AND` l'opérateur binaire ET.
- `OR` l'opérateur binaire OU.
- `(`, `)` les parenthèses pour grouper les expressions.

La déclaration des *pointcuts* peut se faire selon deux manières:

- dans un document `XML` dédié, ou
- dans la classe implémentant l'aspect, en tant qu'annotation.

La première variante est celle qui semble être la manière recommandée par JBoss. L'utilisation d'un fichier de configuration des *pointcuts* respecte la philosophie des serveurs d'application où l'on met en place un maximum de choses dans des fichiers de configuration, pour diminuer la complexité de l'implémentation. De plus, un des objectifs de JBoss étant de pouvoir fournir des aspects prédéfinis à greffer sur une application tierce, le but est de pouvoir le faire sans modifier l'aspect lui-même. Le listing 13 montre la définition des *pointcuts* dans le cadre de l'exemple avec le *shopping cart*.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<aop>
    <aspect class="TraceAspect" scope="PER_VM"/>
    <bind pointcut="!within(TraceAspect) AND execution(*->new(..))">
        <advice aspect="TraceAspect" name="constructorLogging"/>
    </bind>
    <bind pointcut="!within(TraceAspect) AND execution(public void *->*(..))">
        <advice aspect="TraceAspect" name="methodLogging"/>
    </bind>
</aop>
```

Listing 13: Listing jboss-aop.xml contenant la définition des *pointcuts*

La deuxième variante exploite les annotations. Dans ce cas, les *pointcuts* sont déclarés dans le code java, au sein même de l'implémentation des *advices* de l'aspect. Le listing 14 montre le même exemple que précédemment, mais cette fois en utilisant les annotations.

```
import org.jboss.aop.joinpoint.MethodInvocation;
import org.jboss.aop.joinpoint.ConstructorInvocation;
import org.jboss.aop.Bind;
import org.jboss.aop.Aspect;
import org.jboss.aop.advice.Scope;

@Aspect (scope = Scope.PER_VM)
public class TraceAspect {

  @Bind (pointcut="!within(TraceAspect) AND execution(*->new(..))")
  public Object methodLogging(MethodInvocation invocation) throws Throwable {
    /* advice implementation */
  }

  @Bind (pointcut="!within(TraceAspect) AND execution(*->new(..))")
  public Object constructorLogging(ConstructorInvocation invocation) throws Throwable {
    /* advice implementation */
  }
}
```

Listing 14: Listing de TraceAspect.java contenant la définition des *pointcuts* à l'aide d'annotations

Les annotations permettent non seulement de définir les *pointcuts*, ils sont également parfaitement intégrés dans le *matching* des expressions. Ainsi, l'exemple d'expression suivant *match* toute méthode nommée `methodName` déclarée dans une classe annotée `javax.ejb.Entity` et prenant une instance de la classe `String` comme argument.

```
void @javax.ejb.ENtity->methodName(java.lang.String)
```

Dans le reste de cette section, l'accent sera porté sur le configuration des aspects par `XML`. D'une part, parce que les possibilités de l'une et de l'autre manière sont les mêmes et d'autre part parce que la technique *XML* semble être préférée.

Advices L'*advice* est la méthode appelée lorsqu'un certain *join point* est exécuté. Une analogie utilisée dans la documentation officielle de JBoss AOP est que l'*advice* est un gestionnaire d'événement, l'événement étant l'occurrence d'un *jointpoint* dans le flux du programme.

Dans JBoss AOP, l'aspect est une classe java. Les méthodes de la classe sont les *advices*. De manière surprenante, une classe implémentant un aspect ne doit même pas implémenter une interface ou une classe abstraite, alors qu'une telle méthode doit:

1. prendre comme argument une instance de la classe `Invocation` ou une de ses classes filles,
2. pouvoir lever une exception du type `Throwable`, et
3. retourner une instance d'un objet.

Le listing 15 montre l'implémentation de l'aspect de *logging* de notre exemple du *shopping cart*.

```
import org.jboss.aop.joinpoint.MethodInvocation;
import org.jboss.aop.joinpoint.ConstructorInvocation;

public class TraceAspect {

  public Object methodLogging(MethodInvocation invocation) throws Throwable {

    System.out.println("Method: "+invocation.getMethod().getName());

    // Here we invoke the next in the chain
    return invocation.invokeNext();
  }

  public Object constructorLogging(ConstructorInvocation invocation) throws Throwable {

    System.out.println("Constructor: "+invocation.getConstructor().getName());

    // Here we invoke the next in the chain
    return invocation.invokeNext();
  }
}
```

Listing 15: Listing de TraceAspect.java implémentant l'aspect de *logging*

AspectJ prévoit plusieurs types d'*advices* là où JBoss AOP n'en prévoit qu'un. En effet, il est nécessaire de pouvoir préciser quand il faut exécuter l'*advice*. Avant, après ou avant et après le *join point* ? AspectJ traite ce problème en prévoyant les mots clés adéquats dans la syntaxe ajoutée au langage java. En JBoss AOP, c'est dans l'implémentation de l'aspect qu'il s'agit de gérer quand on exécute l'*advice*. Dans le listing 15, on peut voir que les méthodes implémentant les *advices* retournent explicitement la "prochaine instruction". On peut ainsi, en utilisant les structures standards du langage Java — en l'occurence les blocs `try` et `finally`— englober le *pointcut* d'un *advice* exécuté avant, après ou les deux. Le listing 16 montre comment implémenter ces *advices*.

```
public Object methodLogging(MethodInvocation invocation) throws Throwable {
  try {
    System.out.println("Method: " + invocation.getMethod().getName());

    // Here we invoke the next in the chain
    return invocation.invokeNext();
  } finally {
    System.out.println("End of method: "
        + invocation.getMethod().getName());
  }
}
```

Listing 16: Equivalent des *advices* `before`, `after` et `around` dans JBoss AOP

Contexte du join point Pour accéder aux données du contexte du *join point*, JBoss AOP diffère sensiblement d'AspectJ. La force des *advices* de ce dernier est que les paramètres provenant du contexte sont déclarés et passés à l'implémentation de l'*advice*. En JBoss AOP, toute implémentation d'un *advice* reçoit une référence sur le contexte sous la forme du paramètre de type `Invocation`. Dans le listing 17, on obtient l'objet cible par l'argument invocation, que l'on convertit par casting dans le type d'origine.

```
public Object priceConsult(Invocation invocation)
    throws Throwable {
  Item item = (Item)invocation.getTargetObject();
  System.out.println("The price was asked for item: "+item.getID());

  return invocation.invokeNext();
}
```

Listing 17: Obtenir la référence sur l'instance d'un objet dont on modifie un champ

Autres possibilités

JBoss AOP supporte d'autres fonctionnalités définissables dans le fichier de "configuration" ou comme annotation. Les plus importantes sont brièvement décrites ci-dessous.

Introduction La notion d'introduction permet de forcer une classe Java existante d'implémenter une interface donnée. La désignation de la classe peut se faire nommément ou à l'aide d'une expression plus complexe.

```
<introduction class="MaClasse">
  <interfaces>java.io.Serializable</interfaces>
</introduction>
```

Listing 18: Pour forcer la classe *MaClasse* à implémenter *Serializable*.

```
<introduction expr="has(* *->*(..)) OR class(MaClasse.*)">
  <interfaces>java.io.Serializable</interfaces>
</introduction>
```

Listing 19: Pour forcer les classes identifiées par l'expression à implémenter *Serializable*.

Le *mixin* ajoute la possibilité d'ajouter le code java à une classe:

```
<introduction class="MaClasse">
  <interfaces>java.io.Serializable</interfaces>
  <class>MaClasseSerialisable</class>
  <construction>new MaClasseSerialisable(this)</construction>
</introduction>
```

Listing 20: Modification d'une classe à l'aide d'un *mixin*.

Dans le listing 20, on modifie le classe `MaClasse` tout d'abord en la forçant à respecter l'interface `Serializable`, puis on définit son nouveau type et pour finir on écrit le code java permettant de créer une instance de la classe modifiée.

cflow-stack Nous avons vu, plus haut, lors de l'étude des *advices* qu'il n'existe pas directement d'*advice* permettant le contrôle du flux. Il existe cependant une construction nommée `cflow-stack` permettant de définir une suite d'appels, qui si elle est détectée dans le flux du programme, déclenche l'*advice* que l'on peut lier à cette construction.

```
<cflow-stack name="recursive">
  <called expr="void MaClasse->UneMethode(..)"/>
  <called expr="void MaClasse->UneMethode(..)"/>
  <not-called expr="void MaClasse->UneMethode(..)"/>
</cflow-stack>
```

Listing 21: Définition d'un `cflow-stack`.

Le listing 21 montre une construction permettant de lier un *advice* qui sera déclenché si la méthode `UneMethode()` s'appelle récursivement une fois, mais pas plus.

Précédence La notion de précédence est également gérée.

"Hot Deployment"

JBoss AOP permet d'ajouter et de supprimer des *advices* de manière dynamique, lors de l'exécution. Cette possiblité peut s'avérer très intéressante pour appliquer des aspects de test sur un système en production. Il est particulièrement simple de procéder de la sorte dans le cadre du serveur d'application. Une API est prévue pour effectuer de telles opérations dynamiquement depuis le code. Lorsque JBoss AOP est utilisé dans le cadre de JBoss AS[9], intervenir sur la configuration des aspects est très simple, il suffit de modifier ou de remplacer le fichier `jboss-aop.xml`.

```
AdviceBinding binding;
try {
  binding = new AdviceBinding("!within(TraceAspect) AND execution(public void *->*(..))", null);
  binding.addInterceptor(TraceInterceptor.class);
  AspectManager.instance().addBinding(binding);
} catch (ParseException e) {
  e.printStackTrace();
}
```

Listing 22: Ajout d'un *advice de manière dynamique*

Compilation et exécution

JBoss AOP applique les aspects au code du programme en modifiant le *bytecode* java. Cette opération est nommée *weaving*. Il peut avoir lieu à trois moment:

Lors de la précompilation Les classes sont *instrumentées* préalablement à l'exécution par un compilateur d'aspect.

Lors du chargement du programme Les classes sont *instrumentées* lors du premier chargement de celles-ci.

Lors de l'exécution Les *join points* sont "détectés" lors de l'exécution et le programme et alors modifié à ce moment là. Cette manière de faire a évidemment des incidences sur les performances d'exécution, en revanche, elle ouvre la voie au déploiement "à chaud" des aspects. Ce mode de fonctionnement fait probablement partie d'un des objectifs initiaux de l'implémentation AOP de JBoss, car elle a tout son sens dans le cadre de l'utilisation des aspects dans le serveur d'application J2EE *JBoss AS*. On peut ainsi enclencher et déclencher des aspects sur un service en production.

Pour le modification du *bytecode*, c'est la librairie *Javassist* ([12], [13]) qui est utilisée par le weaver pour manipuler la classe compilée. Cet environnement permet de manipuler du *bytecode* sans jamais devoir se préoccuper de sa forme. En effet, le programme est abstrait dans une structure de données que l'on peut manipuler à l'aide d'une `API`.

L'exécution d'un programme précompilé est exécuté la machine virtuelle java standard, invoquée par la commande `java`. Il est nécessaire de définir le `classpath` pour avoir accès aux librairies jboss-aop, le fichier de configuration `XML` et la classe principale:

```
java -cp=<classpath> -Djboss.aop.path=jboss-aop.xml MainClass
```

L'exécution d'un programme aspectisé au chargement ou durant l'exécution est gérée par un agent java (depuis Java 1.5). Motif de la commande pour le weaving au chargement:

```
java -cp=<classpath> -Djboss.aop.path=jboss-aop.xml -javaagent:jboss-aop-jdk50.jar MainClass
```

Motif de la commande pour le weaving durant l'exécution:

```
java -cp=<classpath> -Djboss.aop.path=jboss-aop.xml
-javaagent:jboss-aop-jdk50.jar=-hotSwap MainClass
```

[9] Serveur d'application J2EE.

L'environnement fournit des tâches `ant` et des scripts pour simplifier l'invocation de ces commandes, le nombre de paramètres à fournir étant plutôt important. Le listing 23 montre la sortie de l'exécution de notre programme de test avec l'aspect de *logging*.

```
Method: main
Constructor: Inventory
Constructor: Item
Constructor: Item
Constructor: Item
Method: addItem
End of method: addItem
Method: addItem
End of method: addItem
Method: addItem
End of method: addItem
Constructor: ShoppingCart
Method: addShoppingCartItem
Method: removeItem
End of method: removeItem
Method: addItem
End of method: addItem
End of method: addShoppingCartItem
Method: addShoppingCartItem
Method: removeItem
End of method: removeItem
Method: addItem
End of method: addItem
End of method: addShoppingCartItem
End of method: main
```

Listing 23: Sortie de l'exécution du programme de test avec l'aspect de *logging*

2.5. Décompilation d'une classe aspectisée

Le listing 28 (disponible dans les annexes) montre la classe `ShoppingCartOperator` décompilée après avoir été compilée avec `javac` puis précompilée avec `aopc` (le précompilateur de JBoss AOP). On passe de 15 lignes à 311 lignes, soit un facteur 20 !

2.6. Environnement de développement

JBoss fourni un environnement de développement sous forme de plugin pour Eclipse. Il offre une certaine assistance pour la définition des *pointcuts*. En outre, il propose deux vues:

Vue "Aspect Manager" C'est une vue d'ensemble des entités définies dans le fichier de configuration des aspects `jboss-aop.xml` et permet un accès rapide à leur implémentation.

Vue "Advised Members" Indique les *join points* trouvés dans la classe courante.

Les annotations pour définir les *pointcuts* ne sont pas encore gérées. Comparé à l'environnement de développement, le niveau de support du langage et de ses possibilités n'est pas à la hauteur.

3. Comparaison des outils

3.1. Syntaxe

La manière de déclarer les aspects était à l'origine une des principales différences parmi les outils que nous étudions: AspectJ pour l'extension de la syntaxe du langage java et JBoss pour la définition des *join points* dans un document `XML`. Au fil des versions, un alignement vers le haut à eu lieu ; les trois outils permettent une déclaration des aspects à l'aide des annotations ou d'un document `XML`[10]. La figure 4 donne un aperçu de la syntaxe utilisée par ces outils. AspectJ reste toutefois le seul à étendre la syntaxe du langage Java, ce qui procure certains avantages:

- Validation statique des aspects (*pointcuts* & *advices*).

[10]Dans le cas d'AspectJ, la définition des aspects dans un document `XML` n'est possible qu'avec le weaving au *loadtime*.

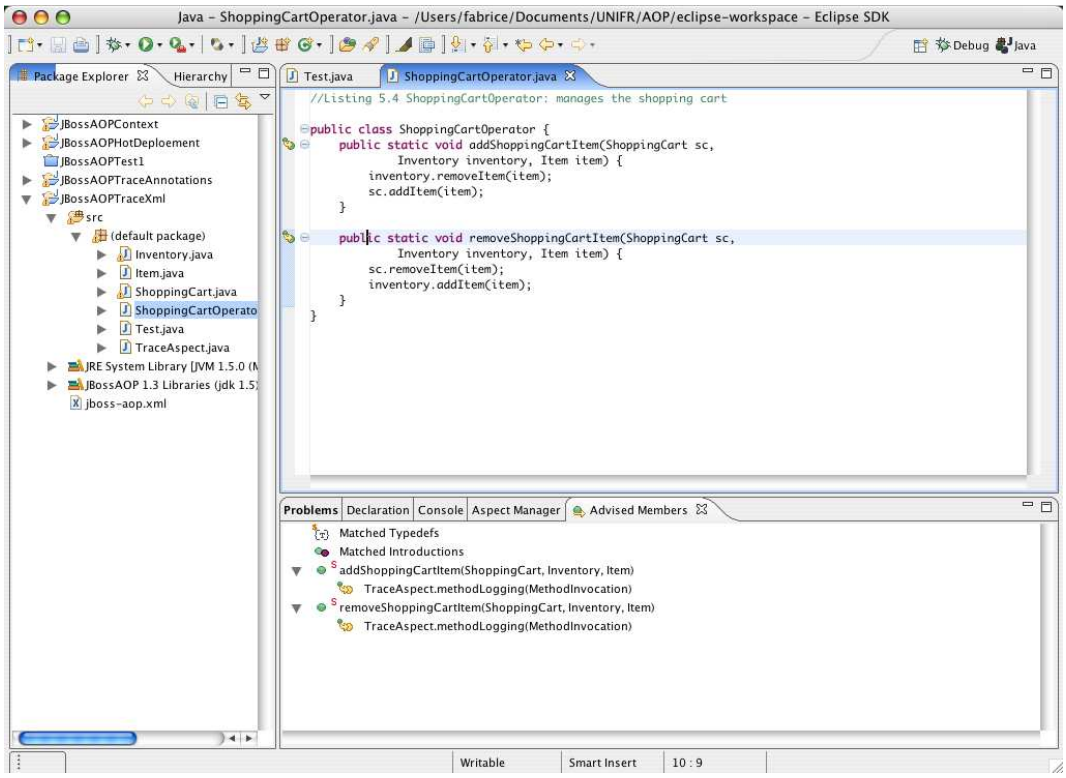

Figure 3.: Plugin JBoss IDE pour Eclipse

- Syntaxe concise, bien que parfois laborieuse.
- Hormis les mots clés supplémentaires, la déclaration et l'implémentation des aspect respectent les principes de la syntaxe Java.
- La définition des *pointcuts* et l'implémentation des aspects sont localisés au même endroit.
- Moins de redondance d'information entre la définition des *join points* et l'implémentation des aspects et de leurs *advices* → moins de code à écrire pour arriver au même résultat.
- Excellent support dans l'environnement de développement (Eclipse plugin).
- Les attributs du contexte "déclenchant" doivent être déclarés dans le *pointcut* de manière typée (pas de casting hasardeux). Les *pointcuts* sont des entités java à part entière.

Et certains désavantages:

- Pour beaucoup de développeurs, il est plus aisé de se conformer à une grammaire `XML` que d'apprendre à maîtriser une extension du langage Java.
- Le fait d'être tributaire du compilateur AspectJ. On n'a aucune garantie que, dans un avenir plus ou moins lointain, le compilateur `ajc` reste en phase avec les évolutions du langage Java.
- Le "hot deployment" n'est pas possible.

Les annotations permettent de définir les *pointcuts* côte à côte avec l'implémentation des *advices*, sans pour autant étendre la syntaxe Java. Nous ne voyons pas d'avantage majeur à les utiliser, autrement que dans les expressions de *matching* des *pointcuts* eux-même. A moins, bien sûr, que l'on souhaite déclarer les *pointcuts* directement dans le corps de l'implémentation des aspects avec JBoss AOP.

	Aspect declaration	Inter-type declarations	Advice bodies	Pointcuts	Static enforcement	Configuration
AspectJ	Annotation, code or XML[1]		Java method or code	Code or string value	Declare, error/warning	aop.xml[1]
JbossAOP	Annotation or XML		Java method	String value	-	jboss-aop.xml
SpringAOP						springconfig.xml

[1] loadtime weaving only

Figure 4.: Comparaison de la syntaxe des outils étudiés

La défintion des aspects et des *join points* dans un fichier de configuration `XML` apporte des avantages qui peuvent être forts intéressants selon les cas. Le fait de pouvoir faire du *hot deployment* avec les aspects peut être vu comme un avantage décisif dans le cas de leur utilisation dans des systèmes en production, nous pensons en particuliers aux applications reposant sur un serveur d'application tel que JBoss... Les fichiers `XML` permettent aussi de configurer les aspects, sans toucher au code et sans recompilation, particulièrement utile pour déployer des aspects prédéfinis.

3.2. Sémantique

Les trois outils étudiés partagent les mêmes principes. La notion de *join point* est la même partout. Tous trois définissent un langage pour décrire ces *join points*. La richesse du vocabulaire conditionne la précision avec laquelle les points dans le flux du programme peuvent être identifiés. Les figures 5 et 6 dressent une vue d'ensemble de la richesse de définition qu'offres les outils.

	Join point kinds and kinded pointcuts				Kindless pointcuts		
	Invocation	Initialization	Access	Exception handling	Control flow	Containment	Conditional
AspectJ	{Method, constructor, advice} x {call, execution}	Instance, static, preinit	Field get/set	Handler	Cflow, cflowbelow	Within, withincode	If
JbossAOP		Instance		(via advice)	(via specified call stack)	Within, withincode, hasmethod/field, all	(via dynamic cflow)
SpringAOP	Non-static method execution	-	-	(via advice)	Cflow	Within, this, target, args	Custom pointcut

Figure 5.: *Join points* et *pointcuts*

On observe que AspectJ et JBoss AOP offrent tous deux des possibilités similaires, avec un léger avantage pour le premier. Spring AOP limite de manière intentionnelle la palette des *pointcuts*, notamment pour des raisons de facilité d'adoption de la part des utilisateurs. Plus les *pointcuts* peuvent être définis précisement, moins de traitement devra être accompli dans l'implémentation de l'*advice* venant se greffer au *join point*.

AspectJ est le seul environnement à offrir un *pointcut* conditionnel `if`, permettant, déjà au niveau de la définition du *join point*, de pouvoir tester un "membre" du contexte du point de jointure décidant du déclenchement ou pas de l'*advice*.

AspectJ prévoit plusieurs types d'*advices*, selon s'il doit s'exécuter avant, après ou autour du point de jointure. Avec JBoss AOP, c'est au développeur de gérer cet aspect-là. Il en résulte un code plus compliqué. Le cas extrême étant une situation où les méthodes et les classes d'un programme sont organisées et nommées de manière à rendre possible l'exposition aux *pointcuts*. A l'opposé, plus les *join*

	Pointcut matching	Pointcut composition	Advice forms	Dynamic context	Instantiated per (scope)	Extensibility
AspectJ	Signature, type, pattern, subtypes, wild card, annotation	&&, ||, !	Before, after, after returning, after throwing, around	This, target, args (all statically typed)	Vm, target, instance, cflow/below	Abstract pointcuts
JbossAOP	Signature, instanceof, wild card, annotation		Around	Via reflective access	Vm, class, instance, join point	Overriding, advice bindings
SpringAOP	Regular expression	&&, ||	Before, after returning, around, throws		Class, instance	

Figure 6.: *Pointcuts* et *advices*

points sont subtiles plus le risque de leurs mauvaise utilisation augmente, tout en rendant le langage plus difficile à appréhender.

	Source	Compiler	Checking	Weaving	Deployment	Run
AspectJ	Extended .java, .xml	Incremental aspectj compile	Full static checking	Compile and load-time	Static deployment	Plain Java program
JbossAOP	Plain .java, .xml	Java compile, post processing	Minimal static checking	Runtime interception, compilation	Hot deployable	Plain Java program, invoked & managed
SpringAOP		Java compile	-	Runtime interception		Invoked & managed

Figure 7.: Environnements de développement

3.3. Weaving

Le weaving — l'opération consistant à insérer le code des *advices* aux *join points* — peut être effectué selon trois modes:

1. Lors de la compilation,
2. lors du chargement du programme,
3. durant l'exécution.

Pour cela, les trois outils comparés différent sensiblement. AspectJ supporte les deux premiers modes, JBoss AOP les supporte les trois, tandis que Spring AOP se focalise sur le troisième mode. Le weaving durant l'exécution permet le déploiement "à chaud" d'aspects. C'est une qualité hautement appréciable dans le cadre d'un serveur d'application tel que JBoss AS[11].

3.4. Support des IDE

La qualité de l'offre d'environnements de développement diffère largement. Tous les outils sont des plugins pour Eclipse. Nous n'avons pas évalué des outils développés par des tiers.

[11] JBoss Application Server, serveur d'application J2EE.

Spring AOP Il existe un plugin Spring qui n'apporte rien pour le développement des aspects, tout au plus apporte t-il une assistance pour l'édition du fichier de configuration `XML`.

AspectJ L'environnement prévu pour AspectJ offre un très bon support pour le développement des aspects. Le balisage des *join points* est excellent et la vérification statique de la syntaxe est irréprochable. L'autocomplétion est fonctionnelle, mais avec les annotations. Seules ombres au tableau, certaines fonctionalités d'Eclipse ne se voient pas supportées, notamment les fonctions de *refactoring*.

JBoss AOP L'environnement prévu pour JBoss AOP apporte une certaine assistance pour la définition des *pointcuts* dans le fichier `XML`, mais on se pose la question s'il ne vaut pas mieux les coder "à la main". Le "balisage" des *joinpoints* ne donne pas l'impression d'être exhaustif. Le plugin est pour ainsi dire inutile si l'on utilise les annotations.

4. Conclusion

Il n'est pas possible de désigner un des outils étudiés comme étant le meilleur. Ils ont chacun leurs qualités et leurs défauts. Le choix de l'un ou l'autre produit devra se faire en fonction des contraintes et la nature du projet.

AspectJ semble devenir un standard de fait. Concernant les serveurs d'application, nous avons vu que les fonctionnalités de Spring AOP sont relativement limitées. Nous avons cependant vu qu'il pouvait s'accorder efficacement avec *AspectJ* pour offrir un conteneur léger pleinement fonctionnel du point de vue de l'AOP.

L'exploration de ces différents outils nous a permis d'observer les nombreuses évolutions et fusions que la technologie Java AOP vit. L'évolution montre que les modes de mise en oeuvre des aspects basées sur des fichiers *XML* et les annotations sont adoptés par chaque outil et deviennent par conséquent de moins en moins un facteur de comparaison.

A. Logique métier du programme de test

Les codes source des exemples utilisés dans ce rapport sont directement tirés de du livre de R. Laddad [11] et constituent l'exemple du chapitre 5.4.1 illustrant le logging par aspect. Les sources des fichiers d'implémentation de l'aspect et de la classe de test sont donnés au fil du rapport.

```
//Listing 5.1 The Item class: models an item that can be purchased

public class Item {
    private String _id;
    private float _price;

    public Item(String id, float price) {
  _id = id;
  _price = price;
    }

    public String getID() {
  return _id;
    }

    public float getPrice() {
  return _price;
    }

    public String toString() {
  return "Item: " + _id;
    }
}
```

Listing 24: Listing `Item.java`

```
//Listing 5.4 ShoppingCartOperator: manages the shopping cart

public class ShoppingCartOperator {
    public static void addShoppingCartItem(ShoppingCart sc,
            Inventory inventory,
```

```
            Item item) {
  inventory.removeItem(item);
  sc.addItem(item);
    }

    public static void removeShoppingCartItem(ShoppingCart sc,
                Inventory inventory,
                Item item) {
  sc.removeItem(item);
  inventory.addItem(item);
    }
}
```

Listing 25: Listing `ShoppingCartOperator.java`

```
//Listing 5.2 The ShoppingCart class: models a shopping cart

import java.util.*;

public class ShoppingCart {
    private List _items = new Vector();

    public void addItem(Item item) {
  _items.add(item);
    }

    public void removeItem(Item item) {
  _items.remove(item);
    }

    public void empty() {
  _items.clear();
    }

    public float totalValue() {
  // unimplemented... free!
  return 0;
    }
}
```

Listing 26: Listing `ShoppingCart.java`

```
//Listing 5.3 The Inventory class: models the shop inventory

import java.util.*;

public class Inventory {
    private List _items = new Vector();

    public void addItem(Item item) {
  _items.add(item);
    }

    public void removeItem(Item item) {
  _items.remove(item);
    }
}
```

Listing 27: Listing `Inventory.java`

B. Extrait de la classe `ShoppingCartOperator` weavée avec JBoss AOP (puis décompilée)

```
// Decompiled by Jad v1.5.8c. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
// Decompiler options: packimports(3)
// Source File Name:   ShoppingCartOperator.java

import /* ... */

public class ShoppingCartOperator
    implements Advised
{
    public static class addShoppingCartItem_8618279472599727500 extends MethodInvocation
        implements Untransformable
    {

```

```
        public Object invokeNext()
16          throws Throwable
        {
18          if(super.currentInterceptor < super.interceptors.length)
            {
20              try
                {
22                  Object obj = super.interceptors[super.currentInterceptor++].invoke(this);
                    return obj;
24              }
                catch(Throwable throwable)
26              {
                    throw throwable;
28              }
                finally
30              {
                    super.currentInterceptor--;
32              }
            } else
34          {
                ShoppingCartOperator.ShoppingCartOperator$addShoppingCartItem$aop(arg0, arg1, arg2);
36              return null;
            }
38      }

40      public void setArguments(Object aobj[])
        {
42          super.arguments = aobj;
            Object obj = aobj[0];
44          arg0 = (ShoppingCart)obj;
            Object obj1 = aobj[1];
46          arg1 = (Inventory)obj1;
            Object obj2 = aobj[2];
48          arg2 = (Item)obj2;
        }
50
        public Object[] getArguments()
52      {
            if(super.marshalledArguments != null)
54          {
                Object aobj[] = super.getArguments();
56              setArguments(aobj);
                return aobj;
58          }
            if(super.arguments != (Object[])null)
60          {
                return (Object[])super.arguments;
62          } else
            {
64              super.arguments = new Object[3];
                super.arguments[0] = arg0;
66              super.arguments[1] = arg1;
                super.arguments[2] = arg2;
68              return super.arguments;
            }
70      }

72      public Invocation copy()
        {
74          addShoppingCartItem_8618279472599727500 addshoppingcartitem_8618279472599727500 = new
                addShoppingCartItem_8618279472599727500(super.interceptors, super.methodHash, super.
                advisedMethod, super.unadvisedMethod, super.advisor);
            addshoppingcartitem_8618279472599727500.arguments = super.arguments;
76          addshoppingcartitem_8618279472599727500.metadata = super.metadata;
            addshoppingcartitem_8618279472599727500.currentInterceptor = super.currentInterceptor;
78          addshoppingcartitem_8618279472599727500.instanceResolver = super.instanceResolver;
            addshoppingcartitem_8618279472599727500.arg0 = arg0;
80          addshoppingcartitem_8618279472599727500.arg1 = arg1;
            addshoppingcartitem_8618279472599727500.arg2 = arg2;
82          return addshoppingcartitem_8618279472599727500;
        }
84
        public ShoppingCart arg0;
86      public Inventory arg1;
        public Item arg2;
88
        public addShoppingCartItem_8618279472599727500(MethodInfo methodinfo, Interceptor ainterceptor[])
90      {
            super(methodinfo, ainterceptor);
92      }

94      public addShoppingCartItem_8618279472599727500(Interceptor ainterceptor[], long l, Method method,
            Method method1, Advisor advisor)
        {
96          super(ainterceptor, l, method, method1, advisor);
        }
```

```
98          public addShoppingCartItem_8618279472599727500(Interceptor ainterceptor[])
100         {
                super(ainterceptor);
102         }

104         public addShoppingCartItem_8618279472599727500()
            {
106         }
        }
108
        public static class removeShoppingCartItem_8750204760336507093 extends MethodInvocation
110         implements Untransformable
        {
112         /* ... */
        }
114

116     public ShoppingCartOperator()
        {
118     }

120     public static void ShoppingCartOperator$addShoppingCartItem$aop(ShoppingCart sc, Inventory inventory,
             Item item)
        {
122         inventory.removeItem(item);
            sc.addItem(item);
124     }

126     public static void ShoppingCartOperator$removeShoppingCartItem$aop(ShoppingCart sc, Inventory inventory
             , Item item)
        {
128         sc.removeItem(item);%Ending the appendix
\renewcommand{\thesection}{\arabic{section}}
130 %%%%%%%%%%%%%%%%%%%%
            inventory.addItem(item);
132     }

134     public Advisor _getAdvisor()
        {
136         return aop$classAdvisor$aop;
        }
138
        public InstanceAdvisor _getInstanceAdvisor()
140     {
            synchronized(this)
142         {
                if(_instanceAdvisor == null)
144                 _instanceAdvisor = new ClassInstanceAdvisor(this);
                return _instanceAdvisor;
146         }
        }
148
        public void _setInstanceAdvisor(InstanceAdvisor instanceadvisor)
150     {
            synchronized(this)
152         {
                _instanceAdvisor = (ClassInstanceAdvisor)instanceadvisor;
154         }
        }
156
        public static void addShoppingCartItem(ShoppingCart shoppingcart, Inventory inventory, Item item)
158     {
            MethodInfo methodinfo = (MethodInfo)aop$MethodInfo_addShoppingCartItem8618279472599727500.get();
160         Interceptor ainterceptor[] = methodinfo.getInterceptors();
            if(ainterceptor != (Object[])null)
162         {
                addShoppingCartItem_8618279472599727500 addshoppingcartitem_8618279472599727500 = new
                     addShoppingCartItem_8618279472599727500(methodinfo, ainterceptor);
164             addshoppingcartitem_8618279472599727500.arg0 = shoppingcart;
                addshoppingcartitem_8618279472599727500.arg1 = inventory;
166             addshoppingcartitem_8618279472599727500.arg2 = item;
                addshoppingcartitem_8618279472599727500.setAdvisor(aop$classAdvisor$aop);
168             addshoppingcartitem_8618279472599727500.invokeNext();
            } else
170         {
                ShoppingCartOperator$addShoppingCartItem$aop(shoppingcart, inventory, item);
172         }
        }
174
        public static void removeShoppingCartItem(ShoppingCart shoppingcart, Inventory inventory, Item item)
176     {
            /* ... */
178     }

180     public static ShoppingCartOperator ShoppingCartOperator_new_$aop()
```

```
    {
182     ConstructorInfo constructorinfo = (ConstructorInfo)aop$constructorInfo_0.get();
        Interceptor ainterceptor[] = constructorinfo.getInterceptors();
184     if(ainterceptor != (Interceptor[])null)
        {
186         ShoppingCartOperator_0OptimizedConstructorInvocation
                shoppingcartoperator_0optimizedconstructorinvocation = new
                ShoppingCartOperator_0OptimizedConstructorInvocation(ainterceptor);
            shoppingcartoperator_0optimizedconstructorinvocation.setConstructor(constructorinfo.
                getConstructor());
188         shoppingcartoperator_0optimizedconstructorinvocation.setAdvisor(aop$classAdvisor$aop);
            return (ShoppingCartOperator)shoppingcartoperator_0optimizedconstructorinvocation.invokeNext();
190     } else
        {
192         return new ShoppingCartOperator();
        }
194 }

196 private static ClassAdvisor aop$classAdvisor$aop = AspectManager.instance().getAdvisor(Class.forName("
        ShoppingCartOperator"));
    protected transient ClassInstanceAdvisor _instanceAdvisor;
198 private static WeakReference aop$MethodInfo_addShoppingCartItem8618279472599727500;
    private static WeakReference aop$MethodInfo_removeShoppingCartItem8750204760336507093;
200 private static WeakReference aop$constructorInfo_0;

202 }
```

Listing 28: Extrait du listing d'une classe weavée décompilée

References

[1] Adrian Colyer. Introducing AspectJ 5. [online].
http://www-128.ibm.com/developerworks/java/library/j-aopwork8/ (accessed Mai 26, 2006).

[2] Erik Gollot. Introduction au framework Spring. [online].
http://ego.developpez.com/spring/ (accessed Juin 9, 2006).

[3] Erik Hilsdale & Jim Hugunin. Advice Weaving in AspectJ. *Aspect-Oriented Software Development*, 2004. [Retrieved Mai 27, 2006, from http://hugunin.net/papers/aosd-2004-cameraReady.pdf].

[4] labs.jboss.com. *JBoss AOP Reference Documentation*. JBoss Inc. / Redhat Inc., 2003.
http://labs.jboss.com/portal/jbossaop/docs/1.5.0.GA/docs/aspect-framewo%rk/reference/en/html/index.html.

[5] Mik Kersten. AOP tools comparison, Part 1. [online].
http://www-128.ibm.com/developerworks/java/library/j-aopwork1/ (accessed Mai 25, 2006).

[6] Mik Kersten. AOP tools comparison, Part 2. [online].
http://www-128.ibm.com/developerworks/java/library/j-aopwork2/ (accessed Mai 25, 2006).

[7] Palo Alto Research Center. AspectJ and AspectWerkz to Join Forces. [online].
http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/aj5announce.html (accessed Mai 25, 2006).

[8] Palo Alto Research Center. AspectJ Documentation. [online].
http://www.eclipse.org/aspectj/docs.php (accessed Mai 25, 2006).

[9] Palo Alto Research Center. Development Kit Developer's Notebook. [online].
http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html (accessed Mai 25, 2006).

[10] Pavel Kouznetsov. Jad - the fast JAva Decompiler.
http://www.kpdus.com/jad.html.

[11] Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

[12] Shigeru Chiba. Javassist (Java Programming Assistant).
http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[13] C. D. . S. J. . E. Sort. Aspect Oriented Programming in JBoss 4. Master thesis, IT University of Copenhagen, February 2004.

[14] Sun Microsystems, Inc. Annotations. [online].
http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html (accessed Mai 25, 2006).

[15] www.springframework.org. Spring Framework v 2.0 M5. [online].
http://www.springframework.org/ (accessed Juin 5, 2006).

Program

Friday, March 24, 2006

10:15 Welcome, General Information, Organisation	Patrik Fuhrer and Jacques Pasquier

Friday, April 28, 2006

14:00 Common background milestone	Patrik Fuhrer and Jacques Pasquier

Friday, May 19, 2006

14:00 Authentication and Authorization	Pascal Bruegger
15:30 Coffee Break	
15:50 Implementing Thread Safety	Lawrence Michel

Friday, June 9, 2006

14:00 Transaction Management	Lorenzo Clementi
15:15 Coffee Break	
15:30 Implementing Business Rules Using AOP	Dominique Guinard

Friday, June 23, 2006

14:00 AOP Tools Comparison	Fabrice Bodmer and Timothée Maret
16:30 Conclusion	Patrik Fuhrer and Jacques Pasquier

Participants

Bodmer, Fabrice	University of Fribourg, Switzerland
Bruegger, Pascal	University of Fribourg, Switzerland
Clementi, Lorenzo	University of Fribourg, Switzerland
Guinard, Dominique	University of Fribourg, Switzerland
Maret, Timothée	University of Fribourg, Switzerland
Michel, Lawrence	University of Fribourg, Switzerland