PaDA: A Pattern for Distribution Aspects

Sérgio Soares*
Universidade Católica de Pernambuco
Departamento de Estatística e Informática
Centro de Informática
Universidade Federal de Pernambuco

Paulo Borba†
Centro de Informática
Universidade Federal de Pernambuco

Abstract

This paper presents a pattern that provides a structure for implementing distribution using AOP — aspect-oriented programming. The main goal is to achieve better separation of concerns avoiding tangled code (code with different concerns interlacing to each other) and spread code (code regarding one concern scattered in several units of the system). Therefore, system modularity, and hence, maintainability and extensibility are increased. The paper also presents an example of distribution aspects using AspectJ, an aspect-oriented extension to Java.

Intent

PaDA (Pattern for Distribution Aspects) provides a structure for implementing distribution code by achieving better separation of concerns. This is obtained through the use of aspect-oriented programming [7]. It increases system modularity, and hence, maintainability and extensibility.

Context

When implementing a distributed system that requires high modularity, meaning that the system should be independent of the distribution concern. To achieve better separation of concern we should use aspect-oriented programming by applying *PaDA*. An aspect defines a crosscutting concern, for example, distribution, which is automatically woven to a system changing its original behavior. Therefore, the system should be implemented in a programming language that has an aspect-oriented like extension. Examples of these languages with the respective aspect-oriented extensions are the following:

- Java — AspectJ [10], HyperJ [12], DemeterJ [11], Composition Filters [3];

*Supported by CAPES. Emails: scbs@cin.ufpe.br, sergio@dei.unicap.br

†Partially supported by CNPq, grant 521994/96-9. Email: phmb@cin.ufpe.br

- C++ — AspectC [6], Composition Filters [3];
- Smalltalk — AspectS [9], Composition Filters [3].

Problem

Tangled code (code with different concerns interlacing to each other) and spread code (code regarding one concern scattered in several units of the system) decrease system modularity. Therefore, maintainability and extensibility are also decreased.

Forces

To distribute a system, *PaDA* balances the following forces:

- Remote communication. The communication between two components of a system should be remote in order to allow several clients accessing the system, considering that the user interface is the distributed part of the system.
- API independence. The system should be completely independent of the communication API and middleware to facilitate system maintenance, as communication code is not tangled with business or user interface code. This also allows changing the communication API without impacting other system code.
- A same system can use different middleware at the same time. This would allow, for instance, two clients accessing the system, one using RMI and the other CORBA.
- Dynamical middleware changing. The system should allow changing the middleware without changing or recompiling its source code.
- Facilitate functional tests. Functional tests are easier by testing the system with its local version; therefore, distribution code errors will not affect the tests.

Solution

In order to solve the problem previously presented, *PaDA* uses aspect-oriented programming [7] to define distribution aspects [16] that can be woven to the system core source code. This separation of concern is achieved by defining aspects to implement a specific concern. After identifying and implementing the crosscutting concerns of a system, they can be automatically composed (woven) with the system source code, resulting on the system version with the required concerns.

Figure 1 illustrates the aspectual decomposition, which identifies the crosscutting concerns of a system, and the aspectual recomposition, or weaving, which composes the identified concerns with the system to obtain the final version with the required functions. In our case, *PaDA* defines just one concern (distribution), which is implemented by three aspects, as we show in next section.

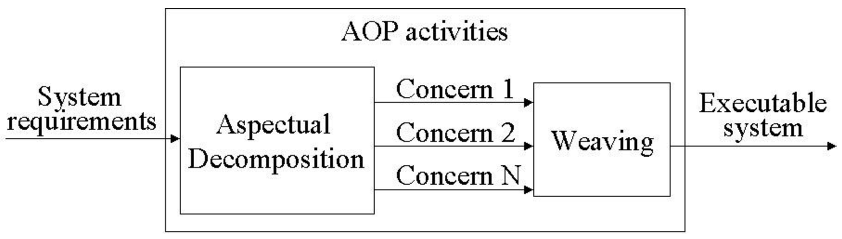

Figure 1: AOP development phases.

Structure

The *PaDA* pattern defines three aspects: one to crosscut the target component (server), another to crosscut the source components (client classes), and the third crosscuts both target and source component to provide the exception handling, as shown in Figure 2. In fact, the third aspect defines a concern that is crosscutting to distribution itself, namely exception handling. In fact, the `ServerSide` aspect might crosscuts others classes that are return types or arguments type of the target component methods.

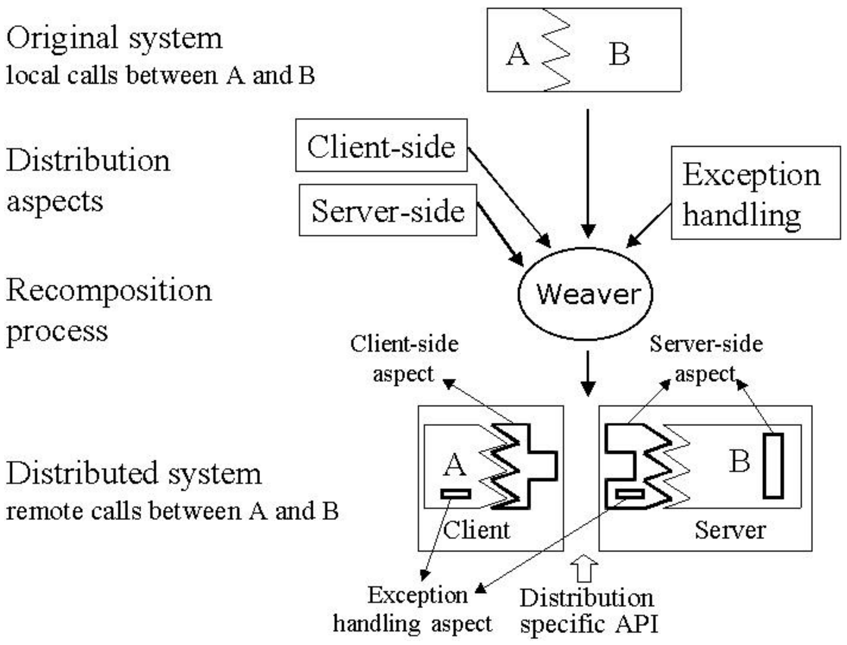

Figure 2: *PaDA*'s structure.

Figure 3 presents a UML class diagram that shows the aspects and their with the components to allow them to be remotely accessed. In that figure, `TargeComponent` is the one to be remotely accessed by instances of `SourceComponent`.

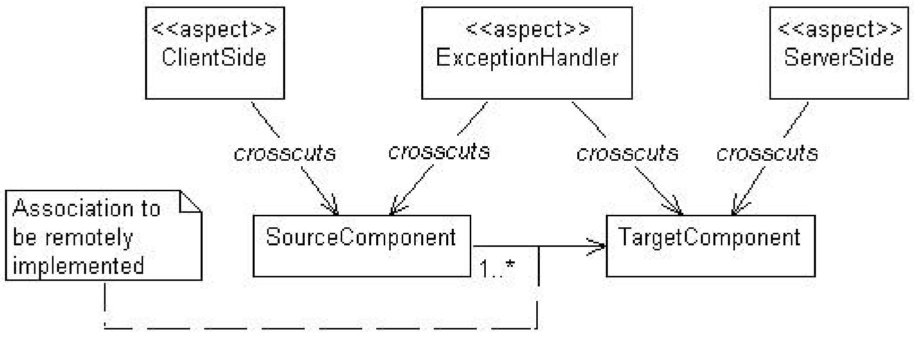

Figure 3: *PaDA* class diagram.

Dynamics

Figure 4 shows a sequence diagram of what is the original system behavior: a `SourceComponent` instance makes local calls to some methods of a `TargetComponent` instance.

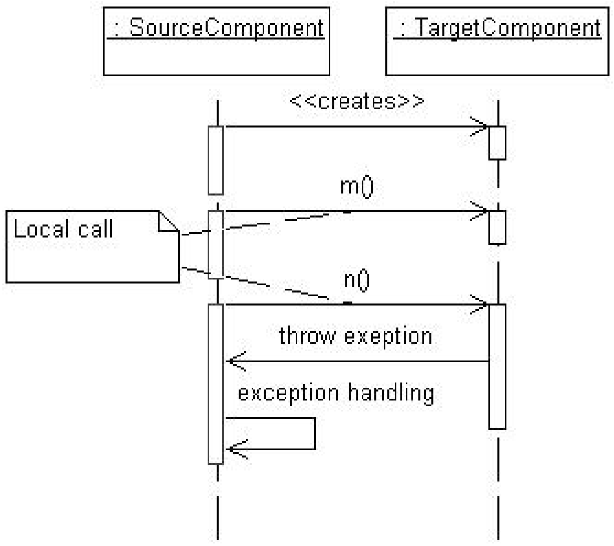

Figure 4: Original system behavior.

Figure 5 is the sequence diagram that states what is the behavior after weaving the aspects to the system: `SourceComponent` local calls are intercepted by the `ClientSide` aspect that gets the reference to the remote instance and redirect the local call to it. Note that the `ServerSide` aspect creates and makes the remote instance (a `TargetComponent` instance) available to response remote calls.

If the remote call raises an exception, like in the `n` method call, the `ExceptionHandler` aspect wraps the exception to an unchecked one and throws it, in the server-side. Note that the message that wraps and throws the unchecked exception is a message to the `ExceptionHandler` aspect itself, because the aspect is also responsible for catching the unchecked exception providing the necessary handling in the client-side (`SourceComponent`).

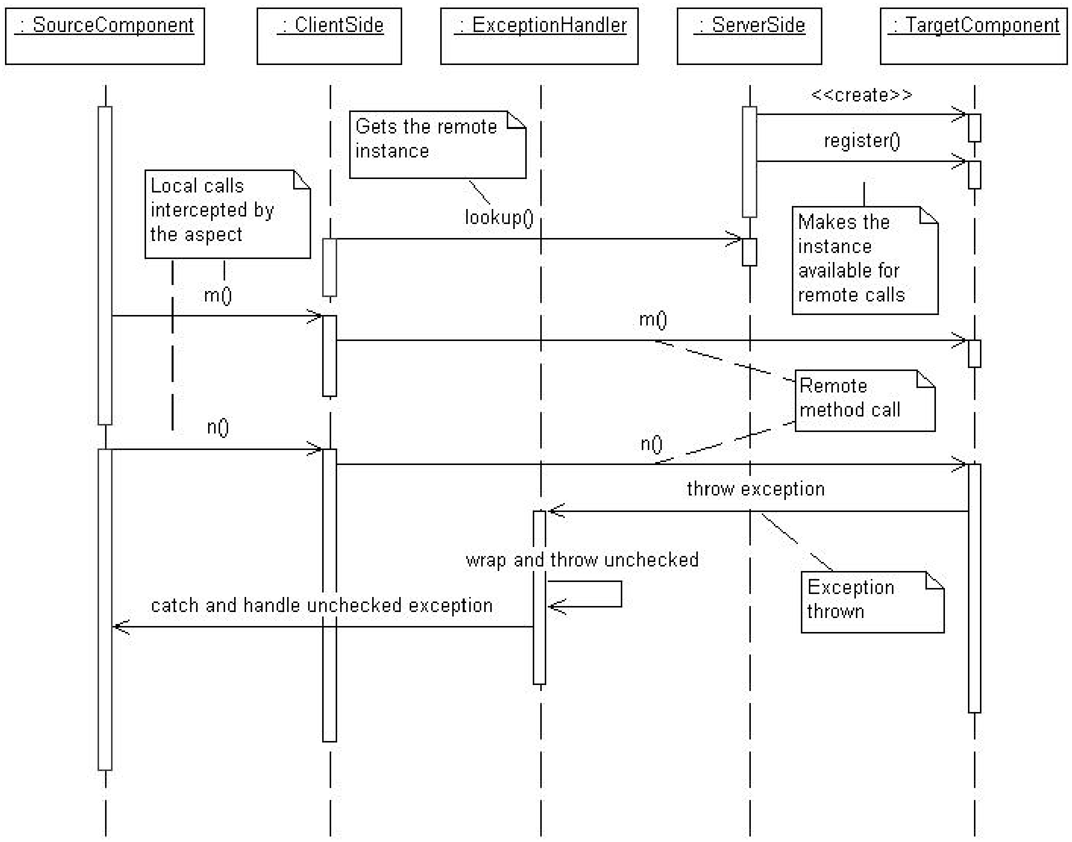

Figure 5: System behavior after applying *PaDA*.

Consequences

The *PaDA* pattern has several benefits:

- *Distributed Implementation.* The pattern provides remote communication between two components of a system;
- *Modularity. PaDA* structures the distribution code in aspects, which is completely separated of the system code, making the system source code API-independent;
- *Maintenance and extensibility.* As the distribution code is completed separated of the system code, changing the communication API is simpler and has no impact in the system code. The programmers should just write another distribution aspects, to the new specific API, or change the aspects already implemented to correct errors and them woven it to the original system source code.
- *Incremental implementation. PaDA* allows incremental implementation [15]. The system can be completely implemented and tested before implementing the distribution aspects, since the distribution aspects are separated from the system source code. This abstraction increases productivity, since the programmers should not take care about distribution problems. This incremental implementation also allows requirements validation without the impact of distribution.

- *Additional separation of concern. PaDA* structure defines exception handling as a crosscutting concern, which is not done by object-oriented programming techniques. Therefore, the exception handling can be changed without impacting in the original system source code and in the distribution aspects, or in others aspects that can be implemented, as the system requires.

- *Facilitate testing of functional requirements.* Tests of the functional requirements can be done easily if made using the system without the distribution. The full separation of concerns preserves the original system source code. This means that the distribution aspect is added to the system just if the composition process (weaving) is executed. Therefore, to obtain the monolithic system, just use the original source code, or remove the distribution aspects from the weaving, in case of the need of another concern, like data management.

The *PaDA* pattern also has the following liabilities:

- *New programming paradigm.* The pattern uses a new programming technique that implies in learning a new programming paradigm to use the pattern. Another impact of being a new programming paradigm is regarding the separation of code that usually was together in the same module. The programmer of the functional requirements cannot see the resultant code that will implement the required concern, decreasing code legibility.

- *Increased number of modules. PaDA* adds three new modules into the system, increasing the modules management complexity.

- *Increased bytecode.* Each aspect definition will result in a class after woven it into the system, which will increase the system bytecode.

- *Name dependence.* The aspects definition depends of the system classes, methods, attributes, and argument names, which decreases the aspects reuse. However, tools can mostly automate the aspects definition, increasing the aspects productivity and reuse.

- *Dynamic change of middleware.* At the moment, the AOP languages do not allow dynamic crosscutting, which does not allow changing the distribution protocol at execution time. This can be done by other design pattern, *DAP* [1], however, without achieving the separation of concerns we achieve with *PaDA*.

- *Allow using a same system through different middleware.* The idea of AOP is generate versions of a system including concerns. The feature of using a same system through different middleware can be achieved if several versions of the system were generated. However, this implies in having several instances of the system (server-side) executing, beside a single one, which may affect or invalidate concurrency control. On the other hand, *DAP* [1] can do this easily.

Implementation

The *PaDA* pattern implementation is composed of four major steps:

- Identify the components, server and client, to have the communication between them distributed.
- Write the server-side aspect. The server-side aspect is responsible to use specific distribution API code changing the server component, making it available to response remote calls. This aspect may also have to change others components used as parameters or return values of the server component, depending of the distribution API.
- Write the client-side aspect. The client-side aspects are responsible to intercept the original local calls made by the client component redirecting them to remote calls made to the remote component (server).
- Write the exception handler aspect. The exception handler aspect is responsible handle with new exceptions added by the aspects definition. These exceptions raised in the server-side are wrapped to an unchecked exception to throw them without changing the signature of the original system source code. Therefore, the exception handler aspect should also provide the necessary handling in the client-side classes.

Example

To exemplify the pattern we now consider a banking application and the RMI API to distribute the communication. Figure 6 presents a UML class diagram that models the banking example.

The `BankServlet` class is a servlet Java that provides a HTML and JavaScript user interface making requests to the `Bank` object. This is the communication to be distributed, therefore the `ServerSide` aspect should crosscuts the `Bank` class and the `ClientSide` aspect should crosscuts the `BankServlet` class. The `Bank` class is the system *Facade* [8] and has attributes like accounts and customers records

```
public class Bank {
  private AccountRecord accounts;
  public void deposit(String number, double value)
      throws AccountNotFoundException {
    accounts.deposit(number,value);
  } ...
}
```

and the operations to manipulate them.

In AspectJ the aspects can affect the dynamic structure of a program changing the way a program executes, by intercepting points of the program execution flow, called *join points*, and adding behavior *before*, *after*, or *around* (instead of) the *join point*. Examples of *join points* are method calls, method executions, instantiations, constructor

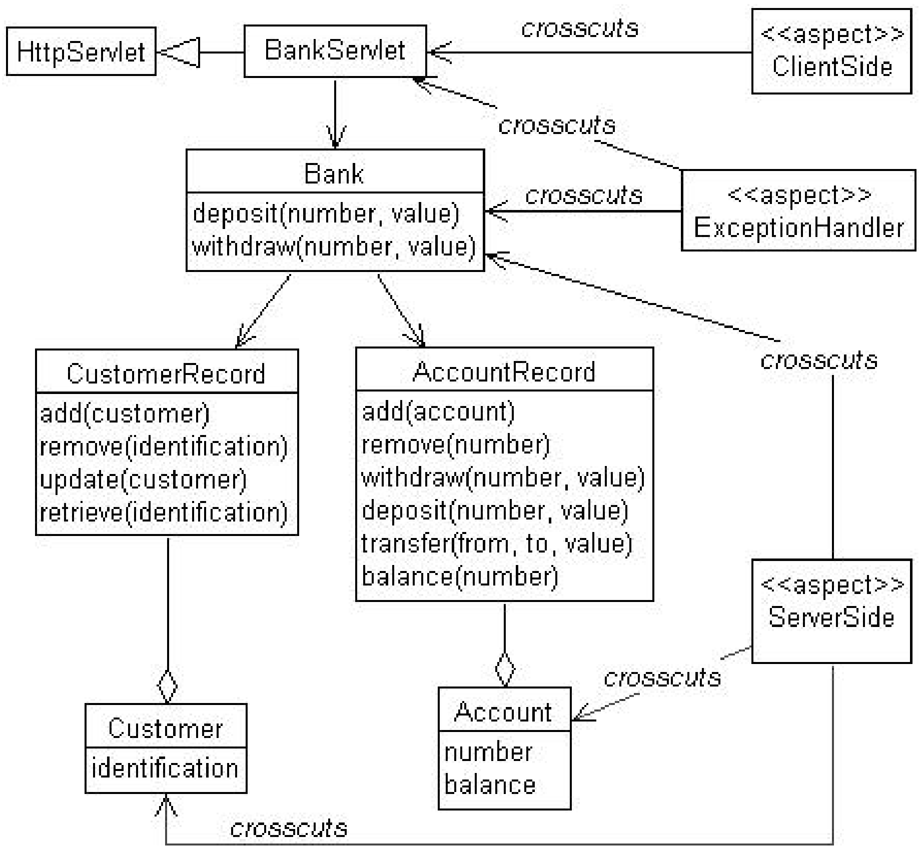

Figure 6: Class diagram of a banking application using *PaDA*.

executions, field references (get and set), exception handling, static initializations, others, and combinations of these by using the !, && or || operators. Usually, an aspect defines a *pointcut* that selects some *join points* and values at those *join points*. Then an *advice* defines the code that is executed when a *pointcut* is reached. The *advice* is who defines what code should execute *before*, *after*, or *around* the *pointcut*.

Server-side aspect

The `ServerSide` aspects should make the `Bank` instance available to remote calls. Besides being the system facade, the `Bank` class also implements the *Singleton* [8] design pattern. The server-side aspect should intercept the `Bank` initialization to make it available to be remotely accessed. The first step is defining a pointcut to identify the `Bank` object initialization, which is shown in following piece of code

```
public aspect ServerSide {
  pointcut bankInit(Bank b): execution(Bank.new(..)) && this(b);
```

where the pointcut designator `execution` join points when any constructor of `Bank` is executed, and the `this` designator join points when the currently executing object is an instance of the type of `b` (`Bank`).

This pointcut is used by the following advice

```
1:  after(Bank b): bankInit(b) {
2:    try {
3:      UnicastRemoteObject.exportObject(b);
4:      java.rmi.Naming.rebind("/BankingSystem", b);
5:    } catch (Exception rmiEx) {  ... }
6:  }
```

that adds some code (lines 2 to 5) after the pointcut, i.e., after the execution of any `Bank` constructor. The added code is responsible to make the `Bank` instance available to be remotely accessed, through the name "BankingSystem".

The server-side aspect has to define a remote interface that has all facade methods signatures adding a specific RMI API exception (`java.rmi.RemoteException`).

```
public interface IRemoteBank extends java.rmi.Remote {
  void deposit(String number, double value)
      throws AccountNotFoundException, java.rmi.RemoteException;
  ...
}
```

The aspect also has to modify the classes whose objects will be remotely transmitted over the distributed communication channel. They just have to use the Java Object Serialization mechanism, by implementing the `java.io.Serializable` interface. We use the AspectJ's introduction mechanism that can modify the static structure of programs to do it, as in the following piece of code

```
declare parents: Bank implements IRemoteBank;
declare parents: Account || Client implements java.io.Serializable;
```

Client-side aspect

The client-side aspect should define a pointcut to identify all executions of the `Bank` methods (lines 3 and 4), and advices to redirect local calls to facade's remote instance, like the one in lines 6 to 13

```
 1:  public aspect ClientSide {
 2:    private IRemoteBank remoteBank;
 3:    pointcut facadeCalls(): within(HttpServlet+) &&
 4:                            call(* Bank.*(..));
 5:
 6:    Object around(double value) throws /* ... */:
 7:        facadeCalls() && call(void deposit(double)) && args(value) {
 8:      Object response = null;
 9:      try {
10:        response = remoteBank.deposit(value);
11:      } catch (RemoteException ex) { ...  }
12:      return resposta;
13:    } ...
14:  }
```

where `remoteBank` (lines `2` and `10`) references the facade remote instance whose local call will be redirected to. In this case the `around` advice executes its code instead the code identified by the pointcut `facadeCalls` and the additional join points (line `7`), that identify calls to the `deposit` methods that gets a `double` as argument, which should be used as argument to the remote call (line `10`).

Exception handling

The AspectJ police to handle with exceptions introduced by the aspects definition is encapsulating them in to an unchecked exception, called *soft* exception. To do it we use the `declare soft` declaration to wrap the `NewException` that gets thrown at any join point picked out by the pointcut `mightThrowNewException`

```
public aspect ExceptionHandler {
  declare soft: NewException: mightThrowNewException();
```

Therefore, this unchecked exception (`SoftException`) should be handled in the user interface class. Note that exception handling is a natural crosscutting concern, usually spread in the system units. To handle this exception we should define an `after throwing` advice that runs after the join points defined by the pointcut `facadeCalls` if it throws the `SoftException`

```
  after() throwing (SoftException ex): facadeCalls() {
    // exception handling, for example, messages to the user
  }
}
```

providing the convenient exception handling.

Variants

An extension of this pattern can define other aspects to provide additional non-functional requirements, such as fault-tolerance, caching, and object transmission on demand to increase both system robustness and efficiency. Aspects can also provide functional requirements.

Another extension can define the three aspects as a single one that crosscuts source and target components and other classes that are return types or arguments type of the target component methods.

Known Uses

This pattern was used in an experiment to implement distribution in a system that allows citizens to complain about health problems and to retrieve information about the public health system, such as the location or the specialties of a health unit. The client-side aspect was defined to the system servlets, and the server-side aspect was defined to the facade class. The system facade was not in the web server due to security and performance reasons.

Another use of *PaDA* in Web based information systems can define the client-side aspect to an applet, but we have not implemented or seen that.

Developers have been using patterns [17, 2] similar to *PaDA* to implement distribution. In particular, the pattern in the first work is similar to *PaDA*'s client-side aspect, and the pattern in the second work is similar to the *PaDA*'s server-side aspect.

In fact, we know several real software projects that implement distribution and could use this pattern. Some of these systems are the following:

- The real system for registering health system complaints.
- A system to manage clients of a telecommunication company. The system is able to register mobile telephones and manage client information and telephone services configuration. The system can be used over the Internet.
- A system for performing online exams. This system has been used to offer different kinds of exams, such as simulations based on previous university entry exams, helping students to evaluate their knowledge before the real exams.
- A complex supermarket system. A system that is responsible for the control of sales in a supermarket. This system will be used in several supermarkets and is already been used in other kinds of stores.

In addition, *PaDA* can be used as one of the basic patterns of the Progressive Implementation Method (Pim) [4]. Pim is a method for the systematic implementation of complex object-oriented applications in Java. In particular, this method supports a progressive approach for object-oriented implementation, where persistence, distribution and concurrency control are not initially considered in the implementation activities, but are gradually introduced, preserving the application's functional requirements. The *PaDA* design pattern can be applied for dealing with distribution.

See Also

- *DAP — Distributed Adapters Pattern [1].* This pattern and *PaDA* has the same objectives, however, *DAP* uses plain object-oriented programming techniques and others design patterns, which do not provide full separation of concerns. Another difference is that *DAP* does not separate the exception handling as a crosscutting concern like *PaDA* does.
- *Reflection [5].* This pattern is related to aspect-oriented programming. It provides a mechanism for changing structure and behavior of software systems dynamically. This pattern splits the application into two levels. A base level that implements the functional requirements, and a meta level that can modify the base level behavior. Comparing it with *PaDA* the base level is analog to the functional requirements, for example, implemented in Java, and the meta level is analog to the aspects, for example, implements in AspectJ.
- *Distributed Proxy Pattern [14].* This pattern and *PaDA* have similar objectives, like making the incorporation of distribution transparent. However, as the previous one, this pattern does not provide full separation of concerns.

- *Wrapper-Facade [13].* Like *PaDA* this pattern has the goal of minimizing platform-specific variation in application code. However, Wrapper-Facade encapsulates existing lower-level non-object-oriented APIs (such as operating systems mutex, sockets, and threads), whereas *PaDA* encapsulates object-oriented distribution APIs, such as RMI and CORBA. Again, this pattern does not provide full separation of concerns.

- *Broker and Trader [5].* These architectural patterns focus mostly on providing fundamental distribution issues, such as marshalling and message protocols. Therefore, they are mostly tailored to the implementation of distributed platforms, such as CORBA. *PaDA* provides a higher level of abstraction: distribution API transparency to both clients and servers.

- *Chain of Responsibility [8].* Similar to *PaDA* this patterns decouples the sender of a request from its receiver. However, it does not perform isolation of the distribution platform's API.

- *Model-View-Controller (MVC) [5]* is used in the context of interactive applications with a flexible human-computer interface. Its goal is to make changes to user interface easy and even possible at run time. *PaDA* is used in the context of distributed applications and aims at making changes to the distribution platform a simple task, not impacting in other parts of the system.

Acknowledgments

We would like to give special thanks to Jorge L. Ortega Arjona, our shepherd, for his important comments, helping us to improve our pattern. We also thanks Rossana Andrade, Jonivan Lisbôa, Marcos Quináia, and Rubens Ferreira for the suggestions made at the conference.

References

[1] Vander Alves and Paulo Borba. Distributed Adapters Pattern: A Design Pattern for Object-Oriented Distributed Applications. In *First Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.

[2] Dan Becker. Design Networked Applications in RMI Using the Adapter Design Pattern. *Java World*, May 1999.

[3] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[4] Paulo Borba, Saulo Araújo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive Implementation of Distributed Java Applications. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 40–47, Los Angeles, EUA, 17th–18th May 1999.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

[6] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path–specific customization in operating system code. *FSE*, 2001.

[7] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect–Oriented Programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] R. Hirschfeld. AspectS: AOP with Squeak. In *OOPSLA'01 Workshop on Advanced Separation of Concerns*, Tampa FL, 2001.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[11] Karl Lieberherr and Doug Orleans. Preventive program maintenance in Demeter/Java. In *International Conference on Software Engineering*, pages 604–605, Boston, MA, 1997.

[12] Harold Ossher and Peri Tarr. Hyper/J: multi–dimensional separation of concerns for Java. In *22nd International Conference on Software Engineering*, pages 734–737. ACM, 2000.

[13] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern–Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*. Wiley & Sons, 2000.

[14] Antonio Rito Silva, Francisco Rosa, and Teresa Goncalves. Distributed proxy: A design pattern for distributed object communication. In *PLoP'97*, Monticello, USA, September 1997. http://jerry.cs.uiuc.edu/~plop/plop97/Proceedings/ritosilva.pdf.

[15] Sérgio Soares and Paulo Borba. Progressive implementation with aspect–oriented programming. In Springer Verlag, editor, *The 12th Workshop for PhD Students in Object–Oriented Systems, ECOOP02*, Malaga, Spain, June 2002.

[16] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA'02, Object Oriented Programming Systems Languages and Applications*. ACM Press, November 2002. To appear.

[17] Gregg Sporar. Retrofit Existing Applications with RMI. *Java World*, January 2001.