The Application Monitor Pattern

Roberta Coelho[1] Ayla Dantas[2] Uirá Kulesza[1]
Walfredo Cirne[2] Arndt von Staa[1] Carlos Lucena[1]

[1]Computer Science Department
Pontifical Catholic University of
Rio de Janeiro (PUC-Rio)
Rio de Janeiro
Brazil
{roberta, uira,arndt,lucena}@inf.puc-rio.br

[2]Computer Science Department
Universidade Federal de Campina Grande
Campina Grande, Paraíba
Brazil
{ayla,walfredo}@lsd.dsc.ufcg.edu.br

Application Monitor

Modern applications are typically complex, multithreaded, distributed, and often should provide real-time responses and small-footprint. Due to such characteristics, most often, it is hard to understand the behavior of such systems and consequently detect the root causes of performance or reliability problems. In order to collect information about system's runtime behavior - operations' performance, internal threads status - the system developer is required to instrument the target application (and sometimes also its execution platform). Such monitoring code which allows the developer to reason about the code execution is not localized in a single application module; it must be included in many modules. As a consequence, the monitoring concern tends to be scattered across multiple application/platform modules and tangled with other application concerns. The Application Monitor pattern supports the separate definition of monitoring-related functionalities concerns through the use of aspect-oriented programming. It decouples such concerns from the implementation of application-specific concerns, which in turn improves the system reusability and maintainability.

Keywords Dynamic Analysis, Monitoring, Aspect-Oriented Programming.

Example Consider a system for Book Trading, which is from herein referred to as Book Trading (BT) system. The BT system follows a service oriented architecture in which system functionalities are structured as a set of loosely coupled modules called services. The two main services that encompasses BT system are the following:

- Book Seller Service: deals with the book-registration requests. This service is accessed by users interested in selling books. It provides a book catalog in which book profiles can be managed. A book profile is a n-tuple which comprises a book title, a price and a book seller's identification.

- Book Buyer Service: receives the book-buying requests submitted to BT system. The Book Buyer Service receives a

book title from the user and interacts with the Book Seller Service in order to find out the cheapest book and then start the selling process.

As soon as a user logs in the BT System one thread of the Book Seller service and one thread of the Book Buyer service are created. As a consequence, each request to the Book Seller service or to the Book Buyer service is executed in separate thread – created per each user. Figure 1 depicts the sequence diagrams of book-registration and book-buying requests.

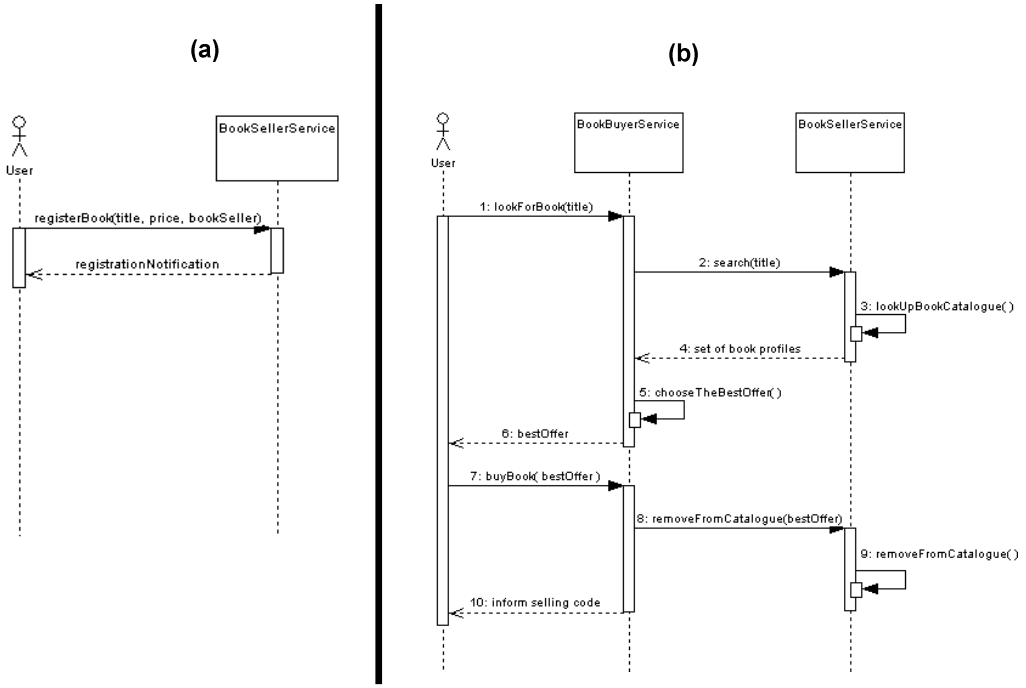

Figure 1. Sequence diagrams of book trading related transactions. Figure 1(a) illustrates the sequence diagram of a book-registration request, and Figure 1(b) depicts the sequence diagram of a book-buying request.

According to Figure 1(a), the BookBuyer service asks the BookSeller service about the available book offers for a specified book title (message 2). Than, BookSeller service searches its catalogue, looking for the requested book (message 3). If there are available books, the BookSeller returns a list of book profiles (message 4). Otherwise, the BookSeller will return a message informing the BookBuyer service that the book is not available - this exceptional message is not represented in Figure 1 since it only illustrates a successful scenario.

The BookBuyer service receives a set of book profiles from BookSeller service and chooses the one with the lowest price (the best book offer) (message 5). Then, it informs the best offer to the user, who sends a "purchase" request (message 7). When the BookBuyer service receives the "purchase" request, it asks the BookSeller service to update its catalogue – remove the book profile of the book that was just sold – (message 8) and notifies the user that the book sale was completed

(message 10).

Along the program execution a variety of information can be harvested, concerning the state of the program modules, the memory usage, and the main operations' performance – which can not be known until runtime. This information can be used to influence the program behavior at runtime, or in a subsequent offline analysis whose result can be used to improve the program on later runs.

During the development of the BT system some questions arose: *What's the cost of each operation? What is the status of each thread (ex: running, waiting or dead) at a specific moment during system execution?*

The answer of the first question enables the developer to discover the root cause of application poor performance. On the other hand, the answer of the second one enables to diagnose deadlocks and also to test a system based on multiple threads~~—~~. ~~as~~Since tests are based on assertions, in order to test a multi-threaded operation, the developer usually needs to know the application threads state before performing certain verifications.

In order to collect such information the developer will need to include monitoring code at specific places of application code. Figure 1 shows the main classes of BT system and some examples of where the monitoring code should be inserted in order to collect the information that allows the developer to answer the two questions stated above.

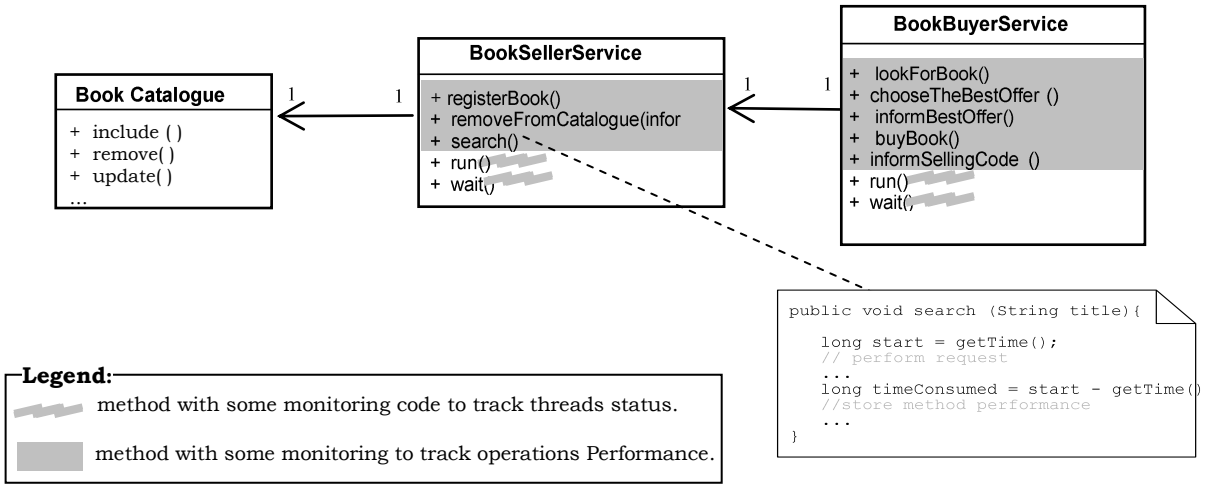

Figure 2. The Object-Oriented design of Book Trading system**.**

The object-oriented design and implementation of the monitoring concern are intrusive. Figure 2 illustrates how the monitoring-related code is tangled and scattered through the basic application classes. The methods affected by the performance-monitoring and thread-monitoring code are shadowed in the figure. The notes depicted in Figure 2 shows some code that should to be included in each method in order to keep track of methods performance and thread status, respectively.

Context When collecting dynamic information about program behavior in order to perform offline analysis or to use this information to make some decision during runtime.

Problem *How to monitor the behavior of complex (possibly legacy) applications?* The following forces are associated with this problem:

- Modern applications are typically complex, multithreaded, distributed, and often should provide real-time responses and small-footprint requirements. Due to such characteristics, most often, it is hard to understand the behavior of such systems and consequently detect the root causes of performance or reliability problems.
- Trying to understand or detect performance and reliability problems in such applications requires the developer to include additional instrumentation code. Such "monitoring" code is not localized in a single component; it is often spread over several application elements. As a consequence, the monitoring code becomes tangled with application code.
- The monitoring code should be defined in a way that facilitates its reusability and maintainability. The developer does not want to copy and paste the same code over several application components, since it will impairs the instrumentation code maintainability.
- It should be easy to evolve the application being monitored as well as the code responsible for monitoring the application.
- The monitoring code does not contribute with the application purpose, but consumes computational resources. Thus, the developer needs an easy way of removing the instrumentation code.
- Sometimes the components to be monitored may be part of third party library, or legacy application of which they do not have access to the source code. The solution should deal with this situation.

Solution Define an application monitor responsible for (i) gathering information about an application property (i.e performance, threads status) from a set of application components, and (ii) making such information available for a runtime use or a subsequent off-line analysis.

The application monitor comprises the monitoring code which, on the other hand, would be scattered across multiple platform modules and tangled with other application concerns. In order to make it possible, the application monitor is built upon the facilities of Aspect Oriented Software Development (AOSD) [7,10]. AOSD has been proposed as a paradigm for improving separation of concerns in software design and implementation. It proposes a new abstraction, called Aspect[1], with new composition mechanisms which support the modularization of crosscutting concerns. The aspect abstraction aims at encapsulating concerns that crosscut several system modules. Since the object monitor has a crosscutting nature, it is addressed in this pattern by an aspect.

Thus, the object monitor is able to crosscut specific classes execution

[1] Appendix A presents a brief overview of terminology related to aspect -oriented programming.

points – e.g. method invocations, constructors calls - and add an extra code in order to store monitoring information in a file or database for a subsequent off-line analysis, or in a data structure available in memory – which can be used to in program runtime decisions.

Structure

Figure 4 illustrates the structure of the Application Monitor pattern. In order to represent the new elements and composition mechanisms of AOP, we defined simple extensions to the standard UML class and sequence diagrams. In the former, an aspect is represented using the `<<aspect>>` stereotype in a class element, and the crosscutting relation between aspects and classes are represented by means of a dependency relationship with the `<<crosscuts>>` stereotype. In the latter, an aspect is represented by a diamond and its pointcuts are represented by circles in the intercepted objects' lifeline (see Figure 5) – this notation was inspired by the one proposed by [3].

The Application Monitor pattern has four participants:

- **Application Monitor**
  - It is one aspect that keeps track of specific events that occur along application execution. In order to do that it uses a Monitoring-Data repository in which it includes monitoring data related to such events. The events of interest are specified by means of pointcuts. Thus, the Application Monitor aspect should be a singleton [8].
- **Monitoring Target**
  - The element whose methods are under monitoring, which can be, for instance: a single class, a subsystem or a data structure.
- **Monitoring Data**
  - Specifies the data that is collected by the Application Monitor. The kind of such data varies according to the purpose of the monitoring task, which can be, for instance: (i) to check the operations performance; (ii) to detect bottle necks in the system; (iii) to track the memory usage[2]; and (iv) for testing purposes of multi-threaded applications.
- **Monitoring- Data Repository**
  - Stores a collection of Monitoring Data. It is only accessed by the monitor aspect in order to guarantee that the monitoring data will not be corrupted by another application component.

[2] It is possible to collect application memory usage data through programming language specific APIs such as System.freeMemory() and System.totalMemory() available in Java language.

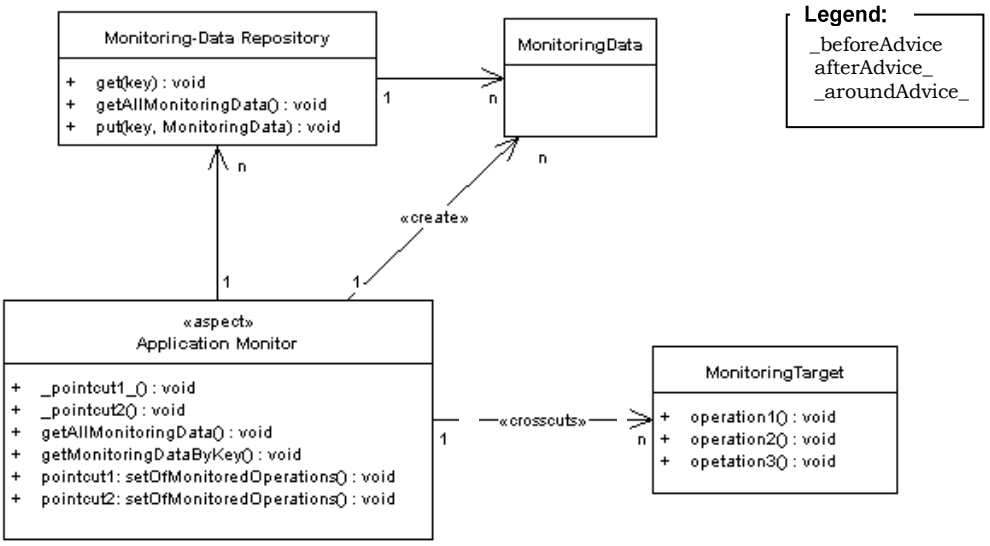

Figure 34. The Static View of the Application Monitor Aspect Pattern.

As depicted in Figure 4, the Application Monitor aspect has four parts: (i) one or more references to Monitoring Data Repositories; (ii) a set of pointcuts that specify the points of execution that should be monitored (`pointcut1` and `pointcut2`) ; (iii) a set of advices that contains the code that will be inserted in such points of execution in order to collect and store monitoring data (the around and the before advices - `pointcut1_()` and `_pointcut2()` respectively); and finally, (iv) a set of methods responsible for exposing the monitoring data, making it available to other system components (`getAllMonitoringData()` and `getMonitoringDataByKey()`).

Dynamics

The following scenarios depict the dynamic behavior of the Application Monitor pattern.

Scenario I – Collecting monitoring data, which is illustrated in Figure 5, presents the sequence of method calls performed when the Application Monitor aspect collects monitoring information at specific points in the execution flow of a Monitoring Target. When the operation to be monitored is called (step 1), the Application Monitor aspect intercepts its execution with a before, after or around advice (step 2). After that, it collects (or calculates) the data to be monitored (step 3) (i.e the operations performance). Then, the Application Monitor finds a key which uniquely identifies the Monitoring Target operation (i.e the signature of the method being monitored) or the Monitoring Target itself (step 4). Finally, it creates the monitoring data and stores it in the repository (steps 5 and 6).

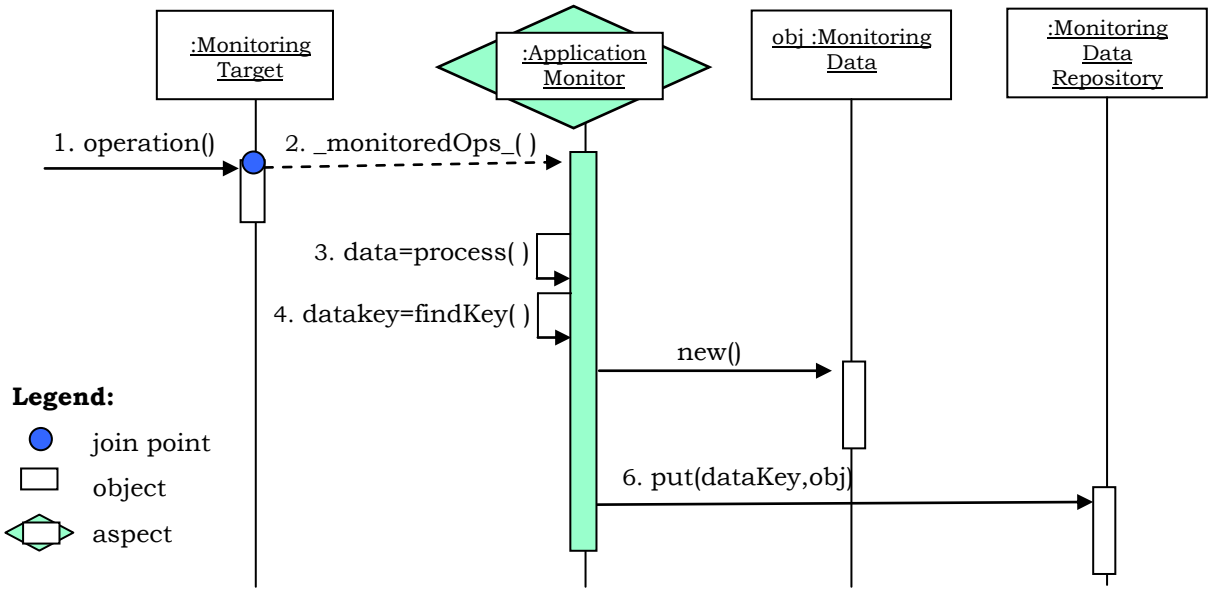

Figure 5. The monitoring task.

Scenario II – Using the monitoring data, which is illustrated in Figure 6, presents the sequence of method calls performed when a component gets any monitoring information. Since the Application Monitor is the only component that has direct access to the Monitoring Data Repositories (step 2), any access to the monitoring data must be mediated by the Application Monitor. The Application Monitor regulates the access to data repositories. By default the application components only have reading access to the monitoring data, through a set of accessor methods available in Application Monitor interface (step 1).

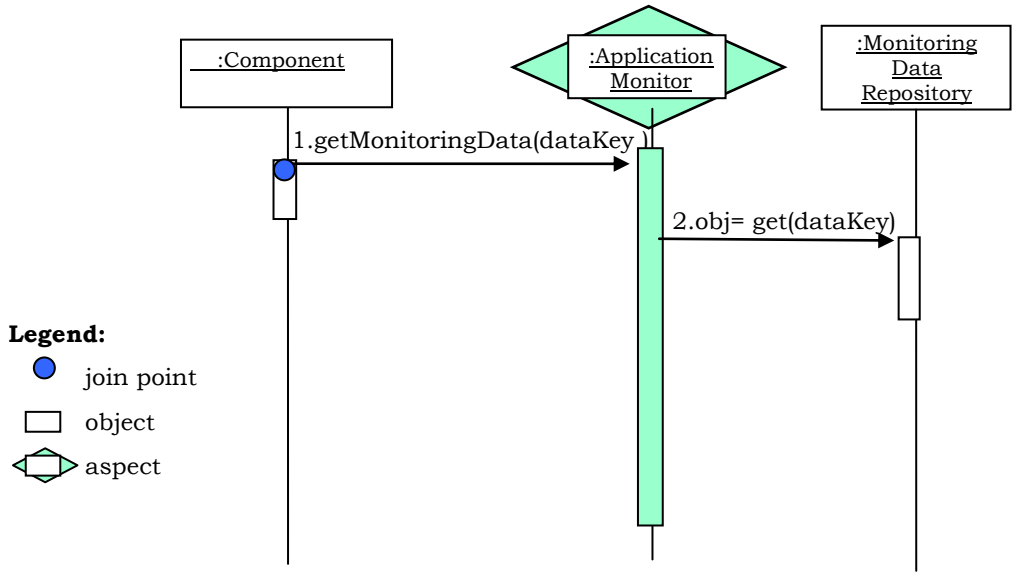

Figure 6. The monitoring data is made available by the Application Monitor and can be accessed by any application component.

Solved Example

Figure 7 illustrates the Application Monitor pattern instantiation in the context of the BT system. Note that, differently from Figure 1, monitoring concerns are modularized here in the Application Monitor aspect. For the sake of simplicity, Figure 7 omits some auxiliary classes which support book tnrading transactions (i.e Book).

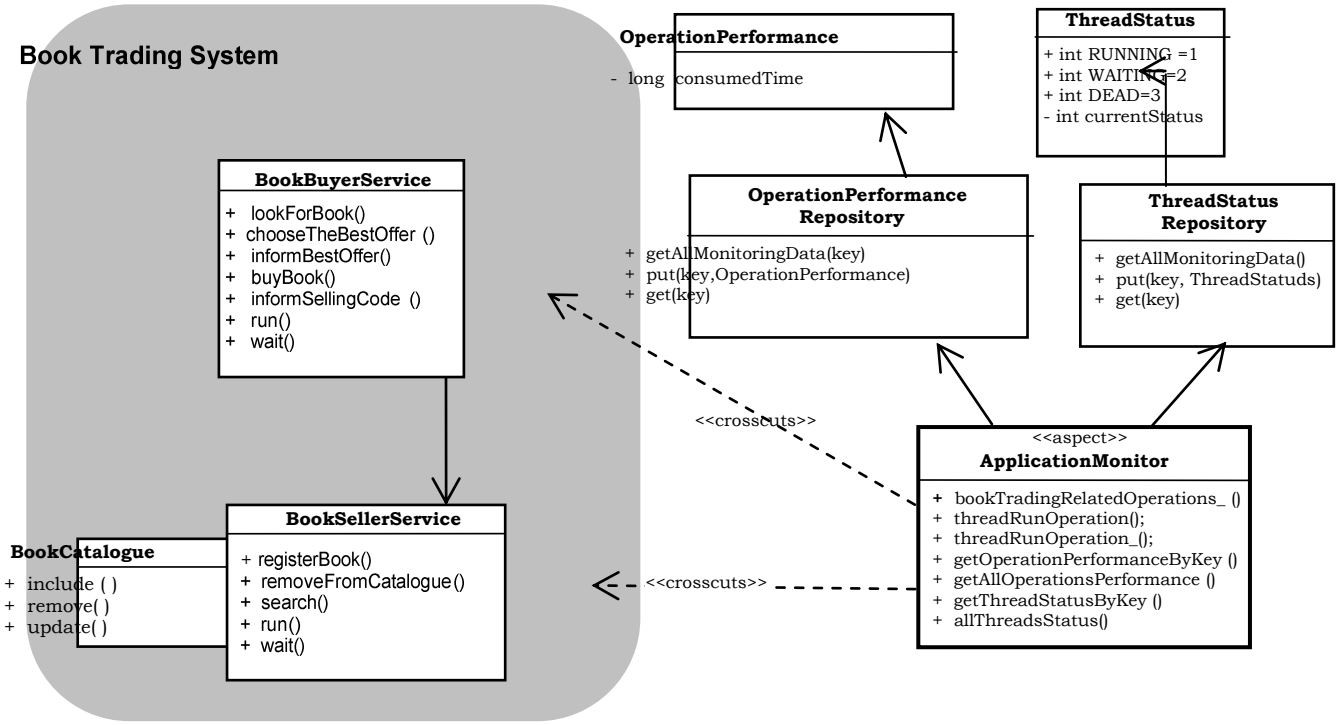

Figure 7. The Application Monitor Pattern for the EC's User Agent.

The Application Monitor aspect crosscuts two classes (BookBuyerService and BookSellerService) whose methods should be monitored for one of the following reasons: (i) the tracking of operations performance or (ii) the tracking of application thread status. The way the Application Monitor aspect affects such classes essentially follows two different patterns detailed in sequence diagrams illustrated in Figures 8 and 9.

Figure 8 illustrates a scenario in which the Application Monitor aspect intercepts a method from BookSellerService (steps 1 and 2) in order to calculate (step 3) and store the methods performance (steps 4, 5, 6).

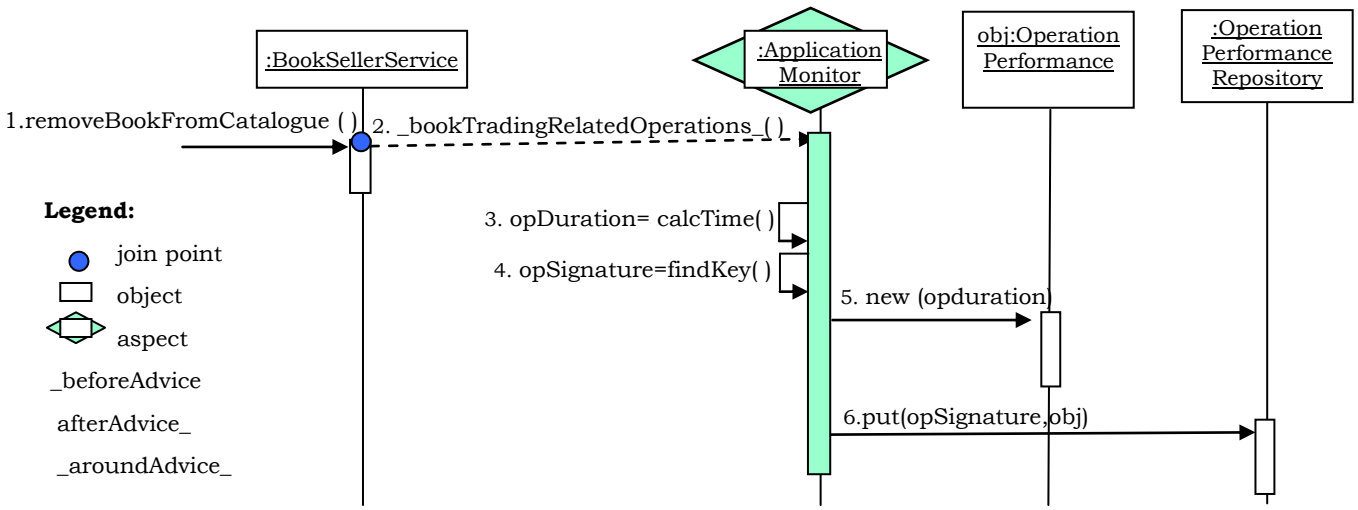

Figure 8. Application Monitor tracking the performance of an operation.

Figure 9 illustrates a scenario in which the Application Monitor aspect intercepts a method from BookBuyerService (steps 1 and 2) in order to collect and store information about the status of the Book Buyer thread.

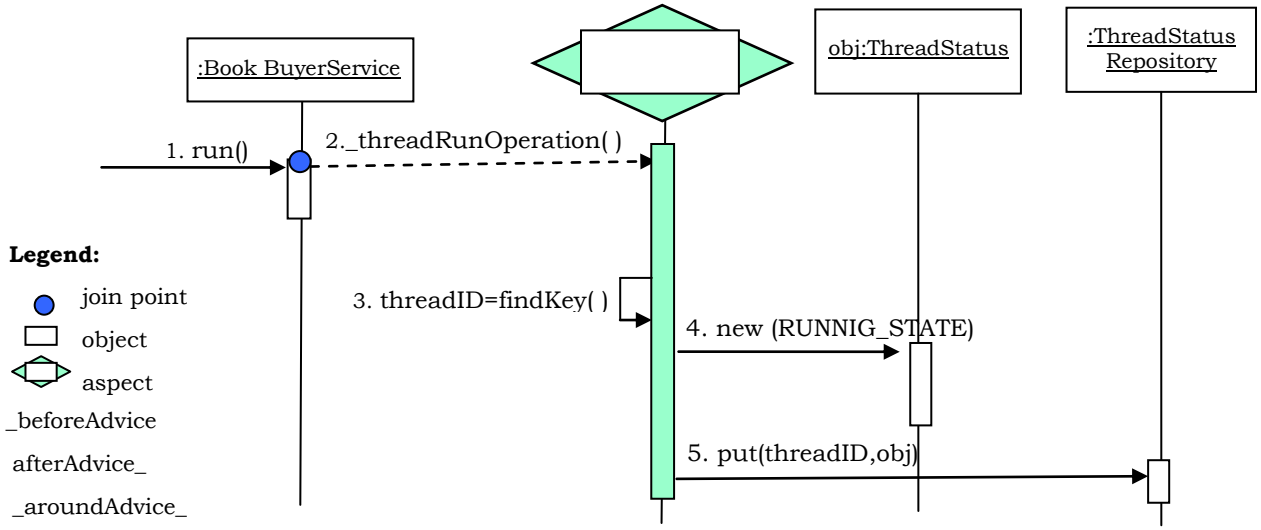

Figure 9. The Application Monitor tracking the thread status of a Book Buyer Service.

Consequences

+ The application classes do not need to be aware that they are monitored. It eases the evolution of the application and the monitoring codes.
+ Different application properties can be monitored without affecting the application code.
+ The monitoring-related code is entirely separated from the other concerns implemented in the application. It contributes to increase maintainability and reusability of the monitoring code.
+ This pattern ~~can~~may monitor components of third-party libraries or legacy systems whose source code is not available, since many aspect-oriented languages (such as AspectJ) support bytecode weaving. In this case, only the .class files are required to perform the weaving.

− This pattern is based on the use of aspects which is not a well known concept by many software developers.

− The proposed pattern may impose a burden on efficiency (due to characteristics of AOP languages) when compared to other non-modularized monitoring solutions. However, with the evolution of AOP language, this burden is becoming shorter.

Implementation

We present some guidelines for implementing the Application Monitor pattern. We provide AspectJ [1,9] code fragments to illustrate a possible codification of the pattern, describing implementation details in the context of BT system example. Although we use AspectJ, the Application Monitor pattern can be specified using other aspect-oriented programming languages following the guidelines presented below.

Step 1: Define the Application Monitor aspect.

The partial code of Application Monitor aspect presented below declares two Java `Hashtables`, the `operationPerfRep` (line 4) and the `threadStatusRep` (line 7), which represents the Monitoring Data Repositories in which the Application Monitor will store the Monitoring Data related to operations' performance and threads status, respectively. We could also implement these data repositories from scratch, creating two data structures that should implement the Monitoring Data Repository interface. However, we prefer to re-use Java Hashtables since it already implements the Monitoring Data Repository interface.

```
1. public aspect ApplicationMonitor {
2.
3.  //Monitoring Data Repository for performance data
4.  private Map operationPerfRep = new Hashtable();
5.
6.  // Monitoring Data Repositoryies for thread status data
7.  private Map threadStatusRep = new Hashtable();

 … ❑
}
```

The repositories are declared as private, which means that "they are private to the aspect". Only code in the aspect can access these data structures. This design decision avoids a monitoring-data corruption by other application component. The ApplicationMonitor aspect should be a singleton [8], since it needs to crosscut all system classes to collect monitoring data. In the current version of AspectJ, each aspect is singleton by default, which means the scope of each aspect is the whole system.

Step 2: Define the types of information to be monitored.

We need to specify what kind of data will be collected by the Application Monitor. In our example, we are interested in collecting two sorts of information: (i) the performance of operations; and (ii) the thread status of Book Sellers and Book Buyers - which are subclasses of Java `Thread` class. Bellow we show partial codes that show how these sorts of data are represented in this example.

```
/* Monitoring Data for tracking operations performance */
public class OperationPerf {

    private long time = 0;

    public OperationPerf (long pTime) {
        time = pTime;
    }
    public long getValue(){
        return time;
    }
    ...
}

/* Monitoring Data for tracking Threads status */
public class ThreadStatus {
    public static final int STARTED = 1;
    public static final int RUNNING = 2;
    public static final int WAITING = 3;
    public static final int DEAD = 4;
    int currentStatus = 0;

    public int getValue(){
          return currentStatus;
    }
}
```

Step 3: Exclude Application Monitor aspect from being monitored.

At this step we want to restrain the monitoring aspect from monitoring himself – this step was inspired in [6]. This restriction is useful, for instance, in scenarios where the developer defines a general pointcut expression to define the points of execution to be monitored:

```
pointcut methodExecutions(): execution(* *..*.*(..));
```

The point cut expression showed above intercepts all the methods executions of all system components. The pointcut expression below, named `withinMonitoring()`, restrains aspects from acting on themselves, in the following manner:

```
pointcut withinMonitoring(): within(booktrading.monitoring..*);
```

This pointcut relies on the fact that all the monitoring-related code is stored in specific packages. It intercepts all the components defined in the package `booktrading.monitoring` and its sub-packages. Using this pointcut expression, a new way of implementing the former pointcut would be:

```
pointcut methodExecutions():
                execution(* *..*.*(..)) && !withinMonitoring();
```

However, such restriction is not be desired in every circumstance. Sometimes it should be useful to measure the effect of the monitoring aspect itself in order to intentionally remove such effect from timing calculations (offline analysis). On the other hand, when the monitoring aspect is part of the system architecture and executes in production environment - constantly collecting performance data - we must consider the instrumentation time in our time calculations.

Step 4: Specify the points of application execution that should be monitored

From our point of view, classes should not be aware of the monitoring related tasks. The execution points that should be affected by monitoring-related code in BT system are shown in Figure 2. The partial code of `ApplicationMonitor` aspect listed bellow details how each one of these execution points should be specified in AspectJ language. `ApplicationMonitor` aspect contains a set of pointcuts for the methods that should be monitored for the purpose of collecting performance or thread lifecycle information.

```
1. public aspect ApplicationMonitor {
2.
     ...
8.   //Set of joinpoints under performance checking
9.   pointcut bookTradingRelatedOperations(Object operation):
10.         execution(* BookBuyer.*(..)) ||
11.         execution(* BookSeller.*(..)) &&
```

```
12.         this(operation);
13.
14.  pointcut allBTSystemOperations(Object obj):
15.         execution(* *..*.*(..)) && !withinMonitoring()&&
16.          this(obj);
17.
18.   //Set of joinpoints thread status monitoring
24.     pointcut threadIsRunnig (Object o):
25.        execution(* BookSeller.run()) ||
26.        execution( *  BookBuyer.run()) && target(o);
27.
```

- ❑

The pointcut `bookTradingRelatedOperations()` (lines 9-12) intercepts all methods which performance should be tracked – Figure 2 shows such methods inside a gray rectangle. The pointcut `allBTSystemOperations()`(lines 14-16) intercept the execution of all application methods, except the methods related to the monitoring concern. In line 24, one of the point cuts responsible for monitoring application's threads status was defined. This pointcut intercepts the execution of the `run()` method from BookSeller and BookBuyer components. The other thread-related monitoring point cuts that should track the other thread status should be similar to the one defined above.

Step 5: Store the monitoring data.

Once the Application Monitor has intercepted the specific points in the program that should be monitored, it is necessary to collect the monitoring information on such points and store them in one of the Monitoring Data Repositories defined in step 1, according to the monitoring purpose (performance or thread status tracking). The advice code associated to each pointcut will be responsible for this task.

The list bellow shows the advices associated with each one of the pointcuts defined in the step 4.

```
28.  // Advice that stores the performance data for each
29.  // monitored operation.
30.  void around() : bookTradingRelatedOperations() {
31.        OperationPerf opTime = null;
32.        long start = getTime();
33.        proceed();
34.        long execTime = getTime() - start;
35.        opTime = new  OperationPerf(execTime);
36.        String key;
37.     key=thisJoinPointStaticPart.getSignature().getName();
40.        Vector times = operationPerfRep.get(key);
41.           if(times == null){
42.               times = new Vector();
43.           }
44.           times.add(pTime);
45.           operationPerfRep.put(key,times);
46.  }
47.  //Advice that stores the status of each system thread
48.    before(Object o):threadIsRunnig(){
```

```
49.       ThreadStatus ts = null;
50.      ts = new ThreadStatus(ThreadStatus.RUNNING);
51.      threadStatusRep.put(getThreadID(o), ts);
52.     }
53.
54. //Advice that stores the status of each system thread
55. after(Object o):threadIsRunnig(){
56.     ThreadStatus ts = null;
57.      ts = new ThreadStatus (ThreadStatus.DEAD);
58.        threadStatusRep.put(getThreadID(o), ts);
59.  }
60.
61.  //Auxiliary method for getting the thread id
62.  private long getThreadID(Thread t){
63.            return new Long (t.hashCode());
64.            // In Java 5.0 would be:
65.            // return new Long (t.getID());
66.  }
```

The `threadStatusRep` hash table defined in step 1 maps a thread id to one instance of the `ThreadStatus` class (lines 51 and 58) which corresponds to the last collected thread status. Hence, when the status of a system thread changes (i.e when a thread dies), its entry in the hash table is overwritten with the new thread status.

However, in order to calculate the performance of an specific method we should not store only the last operation performance, but all the operation performances calculated along the system execution. As a consequence, the `performancePerOperationRep` hash table, defined in the step 1, maps a monitored operation signature to a vector of `OperationPerf` class instances (lines 40-45).

Step 6: Expose the Monitoring Data.

Once the Application Monitor has collected the performance and threads' status data along application execution, there is a variety of options of how to make this information available. The easiest way is to write the information in a log file. Another option is to store it in a database for off-line analysis. Moreover, the Application Monitor should provide direct access to the monitoring data. In order to provide such direct access, a set of accessor methods should be defined in the Application Monitor aspect as detailed bellow.

```
67.  //A set of methods for exposing monitoring data
68.   public IOperationPerformance
69.           getOperationPerformanceByKey (String key){
70.      return operationPerfRep.get(key)
71.   }
72.
73.   public Map getAllOperationsPerformance (){
74.      return generateACopy(operationPerfRep);
75.   }
76.
77.   public ThreadStatus
78.               getThreadStatusByKey (long threadID){
79.      return threadStatusRep.get(new Long(threadID))
80.   }
81.   public Map getAllThreadsStatus(){
82.         return generateACopy(threadStatusRep);
83.   }
```

The above code illustrates is a very simple way of reporting monitoring information during runtime. More sophisticated implementations could be defined, for example:

- The Application Monitor aspect should provide a notification (publish/subscribe) service. This would allow application components register interest in some particular condition (e.g. the number of running threads greater than 50).

- The Application Monitor aspect should enable the integration of the monitored application with a management tool. Management tools, such as the Java Management Extensions (JMX), provides graphical user interfaces to show the monitoring data, required that the application under monitoring implements a specific interface. The Application Monitor pattern should make application elements implement required interfaces through the use of intertype declarations.

Step 7: How to Use the Monitoring Data.

Once the monitoring information was collected and is accessible through the accessor methods defined in the Application Monitor interface (detailed previously), any other component can access the information according to the code showed bellow.

The Application Monitor aspect could be the one responsible for analyzing the monitoring data and generating reports. However, in our example we delegated this task to another component. We defined the PerformanceReporter class which generates a set of reports based on the performance information collected by the Application Monitor. We used the `aspectOf()` static method (line 5) available in all AspectJ aspects. The `aspectOf()` method returns the singleton instance of an aspect, which can be used by any class to call the public methods from an aspect.

```
1.public class PerformanceReporter {
2.
3. public void calcMaxPerformance (String key){
4.   Map perfs;
5.   perf=ApplicationMonitor.aspectOf().
                                getOperationPerformance(key);
6.   //analyze all operation performances
7.   // and generate a report.
        ...
   }
```

Related Patterns

- **Composite Design Pattern [8]:**

  The Composite Design Pattern may be used for tracking Monitoring-Data nested within another Monitoring-Data. The Composite pattern describes part-whole hierarchies where a composite object is comprised of numerous of pieces, all of which are treated as logically equivalent. In this pattern, each Monitoring-Data element should hold other Monitoring-Data elements (children) and the Application Monitor operations responsible for collecting and storing Monitoring-Data elements should be updated to address this new scenario.

- **Singleton [8]:**

  The Application Monitor element is based on the Singleton pattern, which provides a single instance to an object. This aspect is declared to be singleton in order to enable its scope to be the whole system, and as a result the aspect can crosscut all system classes.

Known Uses

Implementation of Application Monitor pattern can be seen in the following scenarios:

- A Unit Test Strategy for multi-agent systems detailed in [4], was developed based on the use of Application Monitor pattern. We have extended the JUnit framework to enable the testing of software agents implemented in the JADE platform [4]. The Application Monitor pattern was used in this system in order monitor the agents' life cycle. Such information was used during test time to enable the execution of unit tests of a multi-agent system. Details about the implementation can be found in [4].
- Bodkin [2] uses AspectJ in a flexible and modular approach to performance monitoring. The performance-monitoring solutions proposed in his work rely on the use of an aspect that contains a reference to a data structure in which the monitoring data is stored. Such solution can also be seen as an implementation of the Application Monitor pattern.
- Deters and Cytron [6] define aspects to harvests runtime information about the application's memory usage. This information is used in subsequent offline analysis. The purpose of this offline analysis was to optimize the program memory usage in latter runs. The Probe aspect defined in this work can be seen as a strict implementation of Application Monitor pattern.
- The Application Monitor pattern was also observed in the OurGrid project while applying a strategy to improve its tests based on AOP [5]. As the OurGrid middleware is multithreaded, it was difficult to provide deterministic tests for it, because a better control of the application threads was necessary for that. By using aspects, the tests could use methods to wait until certain thread configurations before performing assertions and that feature could be implemented in a modularized way. There were

aspects to monitor when each thread changed its state.

References

[1] AspectJ Team.The AspectJ Programming Guide. Available at: eclipse.org/aspectj

[2] Bodkin, R., Performance monitoring with AspectJ – Parts I & II, AOP@Work, Sep 2005. Available at: http://www-128.ibm.com/developerworks/java/libra ry/j-aopwor k12/index.html (Accessed 05/2006)

[3] Chavez, C. A Model-Driven Approach to Aspect-Oriented Design. PhD Thesis, Computer Science Department, PUC-Rio, April 2004, Rio de Janeiro, Brazil.

[4] Coelho, R., Kulesza, U., Staa, A., Lucena, C., Unit Testing in Multi-agent Systems using Mock Agents and Aspects, International Workshop on Software Engineering for Large-scale Multi-Agent Systems (SELMAS), 2006.

[5] Dantas, A., Cirne, W., Saikoski, K. Using AOP to Bring a Project Back in Shape : The OurGrid Case. Journal of the Brazilian Computer Society. 2006.

[6] Deters, M. and Cytron, R., Introduction of Program Instrumentation using Aspects, In: Proceedings of the OOPSLA 2001, Workshop on Advanced Separation of Concerns in Object-Oriented Systems, 2001.

[7] Filman, R., Elrad, T., Clarke, S., Aksit, M. Aspect-Oriented Software Development. Addison-Wesley, 2005.

[8] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[9] Kiczales, G. et al. An Overview of AspectJ. Proceedings of the European Conference on Object-Oriented Programming (ECOOP'01), Budapest, Hungary, 2001.

[10] Kiczales, G. et al. Aspect-Oriented Programming. Proc. of the European Conference on OO Programming - ECOOP'97, LNCS 1241, Springer, Finland, June 1997.

Appendix A – Aspect Terminology

This appendix contains a brief overview of the terminology associated with aspect-oriented software development. We have used the terminology described by Kiczales et al [7, 8] and adopted by aspect-oriented programming languages, such as AspectJ [1]. We present below the main terms that are usually considered as a conceptual framework for aspect-orientated design and programming.

Aspects. Aspects are modular units that aim to support improved separation of crosscutting concerns. An aspect can affect, or crosscut, one or more classes and/or objects in different ways. An aspect can change the static structure (static crosscutting) or the dynamics (dynamic crosscutting) of classes and objects. An aspect is composed of internal attributes and methods, pointcuts, advices, and inter-type declarations.

Join Points and Pointcuts. Join points are the elements that specify how classes and aspects are related. Join points are well-defined points in the dynamic execution of a system. Examples of join points are method calls, method executions, exception throwing and field sets and reads. Pointcuts are collections of join points and may have names.

Advices. Advice is a special method-like construct attached to pointcuts. Advices are dynamic crosscutting features since they affect the dynamic behavior of classes

or objects. There are different kinds of advices: (i) before advices - run whenever a join point is reached and before the actual computation proceeds; (ii) after advices - run after the computation "under the join point" finishes; (iii) around advices run whenever a join point is reached, and has explicit control whether the computation under the join point is allowed to run at all.

Inter-Type Declarations. Inter-type declarations either specify new members (attributes or methods) to the classes to which the aspect is attached, or change the inheritance relationship between classes. Inter-type declarations are static crosscutting features since they affect the static structure of components.

Weaving. Aspects are composed with classes by a process called weaving. Weaver is the mechanism responsible for composing the classes and aspects. Weaving can be performed either as a pre-processing step at compile-time or as a dynamic step at runtime.