Using AOP to Bring a Project Back in Shape: The OurGrid Case

Ayla Dantas[†] **Walfredo Cirne**[†] **Katia Saikoski**[⋆]

[†] Laboratório de Sistemas Distribuídos
Departamento de Sistemas e Computação
Universidade Federal de Campina Grande, 58109-970, Campina Grande, PB, Brazil
ayla, walfredo@dsc.ufcg.edu.br

[⋆] Computing Lab - Research and Development
HP Brazil
katia.saikoski@hp.com

Abstract

The design and development of distributed software is a complex task. This was not different in OurGrid, a project whose objective was to develop a free-to-join grid. After two years of development, it was necessary to redesign OurGrid in order to cope with the integration problems that emerged. This paper reports our experience in using Aspect-Oriented Programming (AOP) in the process of redesigning the OurGrid middleware. The essential direction of our approach was to get the project (and the software) back in shape. We discuss how the lack of separation of concerns created difficulties in the project design and development and how AOP has been introduced to overcome these problems. In particular, we present the event-based pattern designed to better isolate the middleware concerns and the threads. Besides, we also present the aspects designed for managing the threads and for aiding the testing of multithreaded code. We also highlight the lessons learned in the process of regaining control of the software.

Keywords: Separation of Concerns, AspectJ, Grid Computing, Software Reengineering, Software Architecture, Tests

1. Introduction

OurGrid is an open, free-to-join, cooperative grid in which institutions donate their idle computational resources in exchange for accessing someone else's idle resources when needed [8, 35]. OurGrid leverages the fact that people do not use their computers all the time. OurGrid started by providing a simple and complete way for users to run applications over all resources they could log in [9]. This solution has grown in several ways and has become widely used. Several research results (e.g., fault management [25], resource sharing [2, 3], application scheduling [12], etc) have been incorporated into the original prototype. The architecture of the system had to be modified to accommodate such new features. If on the one hand the inclusion of new capabilities was a good evolution for the original idea; on the other hand, it introduced chaos to the project. Most of the new features have been developed by independent teams. In order to help the integration process, automatic tests needed to be provided by each integrator, and all the tests needed to be executed before the integration was completed. However, providing high-quality automatic tests for a grid solution is a complex task. Besides the usual difficulties in developing and testing multi-threaded and distributed software, grids add new challenges due to their wide-dispersion, loose coupling, and presence of multiple administrative domains. In fact, both grid infrastructure and applications are currently very brittle [27]. After the integration process, OurGrid 2.0 was released and used. At this point, the bugs begun to appear and we started to identify improvements that needed to be performed. Nevertheless, changing the code in a secure way had become a challenge and a very error-prone task. The several application concerns were not well isolated, and the application tests were not reliable since they could fail either due to an insufficient time to wait before an assertion or due to a bug. At that point, we have decided to experiment with AOP techniques to identify the problems associated with the software. The first step was to use AspectJ to debug OurGrid and get a better idea and control of its threads, especially during tests. By doing so, we could get a working

version of OurGrid with a reduced number of bugs (OurGrid 2.1.3). Another concern was the software evolution. This working version was very hard to evolve. Then we started an effort to isolate concerns or aspects of the application, redesigning it using an architecture where threads could be well managed and where the tests could be more deterministic and easier to implement.

In this paper we report our experience in using AOP during the OurGrid development, especially in a critical phase of the project, and we present the redesigned OurGrid architecture. The main contribution was a better separation of concerns, which makes easier the integration of new features to the software as well as improves OurGrid's usability. Another contribution is the development of a general package for thread management using AspectJ. This package aids in testing and debugging of multithreaded code.

This work is organized as follows. Section 2 briefly presents AOP and the AspectJ language. Section 3 presents the problems we faced during the OurGrid development, in special, for not dealing correctly with distinct *aspects* of the software. Section 4 presents how we used AspectJ to identify the OurGrid problems and to support the testing process. Section 5 presents the redesigned OurGrid aiming an isolation of concerns and threads management using an event-based architecture. Section 6 is the evaluation section, where we analyze the OurGrid redesign and we present the lessons learned during the process. Section 7 discusses some related work. Finally, Section 8 presents our conclusions and suggests directions for future research.

2. AOP Overview

Separation of concerns is an important matter for software development. In its most general form, separation of concerns refers to the ability to identify, encapsulate, and manipulate the parts of a software that are relevant to a particular concept, goal, task, or purpose [33]. By analyzing the application concerns, we can organize and decompose software into smaller, more manageable and comprehensible parts that address one or more concerns.

Programming paradigms address this issue in distinct forms (e.g., procedures and functions, modules, objects). However, not all *aspects* that need to be addressed in a program can be encapsulated in traditional programming ways to separate concerns (e.g. message logging or error handling). This is because these aspects are usually scattered across the code, which makes maintenance and evolution complicated. Aspect-Oriented Programming (AOP) [21] is a programming paradigm that aims to clearly address the *aspects* that *crosscut* traditional modules. AOP proposes that those aspects (e.g., transaction, message logging, error handling, failure handling, and output formatting) should be written separately from the functional code.

An example of an aspect language is AspectJ [20], which is a general-purpose aspect-oriented extension for Java. It supports the concept of *join points*, which are well-defined points in the execution flow of a program [34]. It also has a way of identifying particular join points (named *pointcuts*) and a mechanism to change the application behavior at join points (named *advice*).

Pointcut designators identify particular join points by picking out a subset of all the join points in the program flow [20] and the corresponding values of objects at those points. A pointcut example is shown in the following:

```
pointcut startingApplication():
  execution (public static void
      main(String [])) ;
```

This pointcut captures the execution of any public and static method called `main` that has a `String[]` parameter and has `void` as its return type. This is just one example of the several kinds of pointcuts provided by AspectJ.

Advice declarations are used to define code that runs when a pointcut is reached. For example, we can define code to run before a pointcut as shown in the following example:

```
before(): startingApplication(){
  System.out.println(
    "The system will start");
}
```

With this advice, a message is displayed on the standard output before the execution of any `main` method identified by the `startingApplication` pointcut. Besides the `before` advice, AspectJ also provides `after` and `around` advice. The former runs after the computation under the join point finishes, while the latter runs when the join point is reached, and has explicit control over whether the computation under the join point is allowed to run at all [34].

AspectJ also has a way of statically affecting a program. With *inter-type* declarations, the static structure of a program can be changed. For example, we can change the members of a class and the relationship between classes [34].

Finally, AspectJ has the concept of an *aspect*, which is a modular unit of crosscutting implementation. An aspect is defined similar to a class definition, and it can have methods, fields, constructors, initializers, named pointcuts, advice and inter-type declarations. In short, aspects group pointcuts, advice, and inter-type declarations. The aspect code is combined with the primary program code by an aspect *weaver* [40].

3. Getting into Trouble during the Development of OurGrid

In this section we give an overview of OurGrid and grid computing. Then, we discuss the problems we had

in the process used to build this grid middleware. Finally, we describe the project status before the redesign.

3.1. OurGrid overview

OurGrid is an open, free-to-join, cooperative grid in which participants donate their idle computational resources in exchange for accessing other participants' idle resources when needed [8]. It aims to enhance the computing capabilities of research labs around the world, by allowing them to trade resources that would otherwise be wasted. OurGrid was designed to be scalable, both in the sense that it supports thousands of labs, and that joining the system is straightforward. In fact, anyone can just download the OurGrid software and join the grid. There is no need for paperwork or human negotiation, as it is the case for other grids [8].

The current design of OurGrid assumes applications to be Bag-of-Tasks (BoT). A Bag-of-Tasks application is a parallel application composed of independent tasks that can be executed in any order. A typical example of a BoT application is a set of tasks composing a parameter-sweep simulation. BoT applications are both relevant and suited for execution in grids. OurGrid is open-source and it is available for downloading at `http://www.ourgrid.org`. The current status of the OurGrid community is available at `http://status.ourgrid.org`. The OurGrid basic architecture shown in Figure 1 comprises the following elements: the **Grid Machines** (GuMs), the **Grid Machine Providers** (GuMPs) and the **Scheduler**.

We may summarize the communication between these elements as follows. The user submits jobs to the **Scheduler**. The **Scheduler** then requests machines (GuMs) to the providers (GuMPs) and allocates the received GuMs to execute the tasks of the submitted jobs. A **GuMP** controls machines in a given administrative domain. GuMPs can trade machines among themselves using an incentive-compatible peer-to-peer protocol [2, 3].

In the 1.0 version of OurGrid[1], users could submit their jobs directly through the `Scheduler` remote object or using Linux shell scripts. The machines available for the jobs in the initial versions were just those previously configured in a local grid machine provider.

OurGrid 2.0 brought a host of new features compared with the 1.0 version, such as new scheduling heuristics [12, 30] and a new way to obtain GuMs using a peer-to-peer community [3]. However, the process of incorporating these functionalities faced some problems. As with other grid systems [27], the result was a brittle code. Besides the classic difficulties in developing and testing multi-threaded and distributed software, grids add new challenges due to their wide-dispersion, loose coupling, and presence of multiple administrative domains. In particular, we had very serious problems in testing the software.

[1]Version 1.0 of OurGrid was originally called MyGrid and it was released in the beginning of 2003

3.2. Problems in our development process

The process used to build OurGrid was XP-based [5]. However, during the development of the initial OurGrid versions, practices such as "Continuous Integration", "Pair Programming", "Collective Ownership" and "Small Releases" were not used.

In contrast with the weakness of omitting some practices, there was a strong point in the process by the use of "automatic tests". However, even the tests had problems (low coverage and non determinism) and the test-first approach (implement the tests before the functionality being tested) was not used due to the system complexity and the strict deadlines.

Once the initial version of OurGrid was developed, several research efforts evolved in parallel. The evolution of the OurGrid code and the external features were managed separately (e.g., different branches in a repository, bug fixing, etc.). Then, after one year, each branch needed to be integrated into the repository main branch in order to release version 2.0 with several new features.

The integration of each new feature (each branch in the repository tree) was a hard task. First, the automatic tests took a long time to be performed (more than 3 hours). Second, sometimes the tests failed because of timing problems, that is, the time the test would wait before assertion was not enough because the functionality to be tested had not finished yet. This happened because the test thread usually created other threads that would actually make the assertions correct. Therefore, sometimes it was necessary to include `sleep` calls on the tests to make them wait before these assertions. In several cases the code was committed with bugs because application developers interpreted failures as an insufficient "sleep" time in the thread control of the test. Finally, some of the failures were really bugs, especially those related to the order of execution of threads that were not foreseen. Such bugs made the tests pass sometimes and fail in others.

The result of this integration approach was a repository with non-deterministic tests whose failures could not be well diagnosed. This was due to the increase in the number of threads and synchronized blocks in the application (for example, there were 174 synchronized blocks at the end of this set of integrations). The worst scenario was that each thread could freely "walk" throughout a poorly modularized code, making deadlocks easy to be created and difficult to be detected. To make things worse, because tests would take a long time, developers would frequently interrupt a test and execute it later without noticing that the delay could be the result of a deadlock that only happened in a certain threads configuration. Besides the testing problems, the code was not well understood by the whole development team and evolution (including bug fixing) had become a hard problem, especially because separation of crosscutting concerns and even of the concerns that were not so crosscutting has not been considered during the integration.

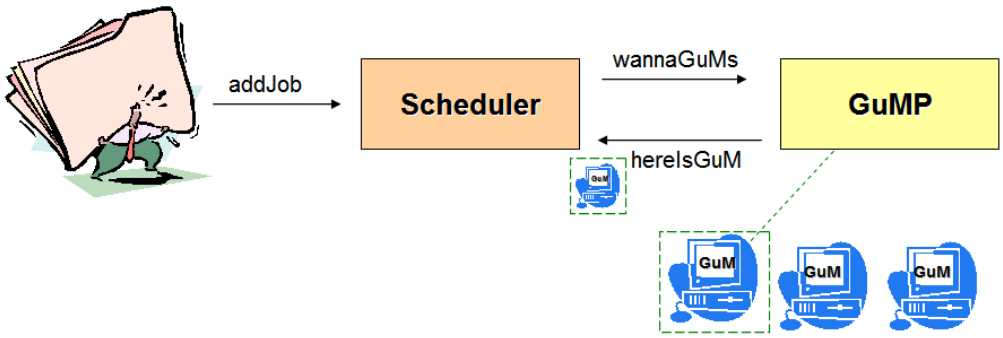

Figure 1. OurGrid Architecture Basics

4. Using Aspects to Diagnose the Problems and Test OurGrid

Because the code of OurGrid was handled by several people (around 20) without using *pair programming* and not focusing on the *collective code ownership* practice [5], it was difficult to understand strange behaviors of the complete application. An even more difficult task was to correct the bugs. Bug fixing begun to take much more time than expected after the release of versions 2.0 and 2.1. As a consequence, we started an effort to better understand the code in order to make its evolution possible. At this phase, *aspects* [21] were introduced.

4.1. Management of application threads during debugging and testing

The main problem was to understand the behavior of the application threads. In order to do that using pure object-oriented programming, we would need to insert code in several parts of the application (e.g., in each thread creation, `sleep` and `wait` call, etc). So, we decided to use AOP to help in this process. In the end, AOP was not only used to debug the code, but also to improve the testing process by controlling the application threads before performing assertions.

However, there were problems regarding the use of AOP by the development team. First, only one person was familiar with AOP and second, the team was very heterogeneous and changed frequently. The solution was to devise a transparent way to introduce AOP. This would allow anyone to use it, even without AOP knowledge. The programmers simply needed to compile their programs with a different command and to know the methods of a class that provided services that made the tests threads wait for the other application threads before proceeding. For the debugging process, we did not focus on much transparency because our intention was only to understand the problems in application threads. More specifically, we wanted to identify the following points: when threads started and finished to run, when the threads waited on a monitor or when these waiting threads were notified. This debugging code led to a general package for testing called `org.ourgrid.threadservices`, which can be reused in other projects.

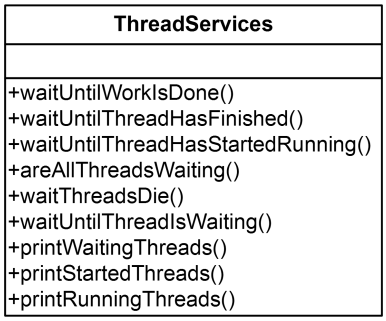

Figure 2. Thread Services class operations

The OurGrid tests use the JUnit framework [24], which is based on assertions. Each test invokes the application and then it makes an assertion to verify if a certain state has been reached. Before these assertions, we usually used `sleep` calls with a certain amount of time to wait until a certain condition to be tested was achieved. This happened because the test thread usually created other threads that actually changed the application state to the one expected by the test assertion. Nevertheless, the sleep time was not a good solution because it is dependent on the machine being used in the test and on the load it faces.

The `org.ourgrid.threadservices` package solves this problem. It was accessed through the `ThreadServices` class, which is a common class with a set of static operations (illustrated in Figure 2). From the methods available, the most used for tests was the `waitUntilWorkIsDone` method. This method makes the caller thread wait until all threads started by the test have finished or were waiting on a monitor. The other methods used to make the test thread to wait (methods `wait*`) were also widely used and they replaced most of the `sleep` calls from the tests, making them faster and more deterministic. For instance, if we wanted to test the Scheduler, the test would submit a job to it and then verify if the job had been successfully executed asserting that the job state was finished. Before asserting if the execution had successfully finished, we would call

the `ThreadServices.waitUntilWorkIsDone` method and then, the test thread would wait until the scheduler thread and the other threads started by the test had finished running or were waiting on a monitor.

In order to use the services shown in Figure 2, besides invoking the `ThreadServices` class on the tests, it was necessary to replace the command used to compile the application before the tests. In more practical terms, application developers need to invoke the *ant aspects* command. This did not cause any impact in the development process since developers already used Ant [18] for testing and compiling.

In order to implement the `ThreadServices` methods without directly changing any part of the code, we have implemented the `RunningThreadsMonitor` aspect. The `RunningThreadsMonitor` aspect manages the application threads with the help of a Java class called `ThreadLists`, which manages thread collections. Some of the `RunningThreadsMonitor` pointcut definitions are the following.

```
pointcut threadStartCalls(Thread t):
  call(public void start())&& target(t);
pointcut waitCalls(Object o):
  call(public void wait())&& target(o);
pointcut runnableRunExecutions():
  execution(public void Runnable+.run());
```

The `threadStartCalls` pointcut illustrated above collects every call to the `start` method on a `Thread` object. The `waitCalls` pointcut collects every call to the `wait` method on any `0bject`. This will indicate that one of the threads will be waiting on a given object. The `runnableRunExecutions` captures the moment a thread is actually running. This corresponds to the execution of every `run` method from a `Runnable` object or from a class that implements this interface, such as the `Thread` class. These execution points were important to capture, because in most of the tests the threads are only started (i.e., the `start` method is called). However, when the `start` method returns, it does not mean that the `run` method has started.

In the following, we show a number of *advice*s that present the code to be run immediately before the pointcuts described above are reached.

```
before(Thread t): threadStartCalls(t){
  tLists.includeInStartedThreads(t);
}

before(): runnableRunExecutions(){
  tLists.includeInRunningThreads(
    Thread.currentThread());
}

before(Object o): waitCalls(o) &&
  !within(ThreadLists){
    tLists.addWaiting(o);
}
```

The code inside each advice will include the threads captured by the pointcuts in different collections of threads according to their state (started, running or waiting). As the `ThreadLists` class is responsible for these collections of threads, the inclusion is done through an instance of this class, named `tLists`. Note that in the last advice we take the caution of excluding the `wait` calls that were inside the `ThreadLists` class itself.

We have also included an `after` advice to remove a thread from the collection of running threads after the `run` method execution has finished:

```
after(): runnableRunExecutions(){
  tLists.removeRunnableThread();
}
```

Another two `before` advices have also been implemented, as illustrated below. They are invoked when `notify` and `notifyAll` methods are called during the execution, but not within `ThreadLists` class. They are responsible for notifying the `tLists` object about threads that may stop to wait, and that could, therefore, have changed their state.

```
before(Object o):(
  call(public void notifyAll()))
    && target(o) &&!within(ThreadLists){
      tLists.notifyAllWaitingThreads(o);
}

before(Object o):(
  call(public void notify()))
    && target(o) &&!within(ThreadLists){
      tLists.notifyOneWaitingThread(o);
}
```

As could be seen, `ThreadLists` is responsible for managing the state of application threads. It implements the functionality that waits until a given configuration of threads is achieved or prints threads in a given state, which is provided by the `ThreadServices` class (see Figure 2). In order to do this, the `RunningThreadsMonitor` aspect replaces the `ThreadServices` static methods implementation using the AspectJ `around` advice. Developers would use static methods from `ThreadServices` class, but would be in fact using a real instance of `ThreadLists` instantiated by the `RunningThreadsMonitor` aspect. This little "trick" was essential to make the use of aspects transparent to most programmers.

One of these advices is shown below. The others follow the same idea.

```
void around(): execution(public static void
  org.ourgrid.threadServices.ThreadServices.
    waitUntilWorkIsDone()){
      tLists.waitUntilWorkIsDoneNotifying();
}
```

This advice defines that instead of executing what is defined in the body of the

`ThreadServices.waitUntilWorkIsDone` method, the `waitUntilWorkIsDoneNotifying` method is called on the `ThreadLists` instance owned by the aspect.

4.2. Finding deadlocks through existing tests

After we have solved the problems of non-deterministic tests that failed because of timing problems, we still had the challenge of discovering application deadlocks.

We randomly called the `sleep` method (with a random time at a given interval) on running threads so that application threads would run in different orders. Aspects aided in this task and provided a solution more appropriated than those based on pure object-oriented approach where several parts of the code needed to be changed. Testing would then be run over and over seeking for a deadlock.

A single advice in a new aspect (the `ThreadSleeperAspect`) was necessary to perform this task:

```
before(): call (* *(..))
  && withincode (* *..*.run()) {
    makeThreadSleepIfItIsHerTurn();
}
```

This advice invokes the `makeThreadSleepIfItIsHerTurn` method before the execution of every call to any method inside a `run` method execution. The method invoked inside the advice verifies if the sleep should be called or not, according to a random choice, and then invokes `sleep` on the currently executing thread choosing a random interval.

Although this aspect was really useful for us in order to find deadlocks, random `sleep` calls considerably increased the execution time of tests. Therefore, the tests should not be executed every time with these `sleep` calls. As we were using AspectJ, in order to perform this, we just exclude the `ThreadSleeperAspect` from the weaving process. To do that, the developers simply needed to use a different Ant task to compile.

In order to find a deadlock through a test that sometimes gets blocked and sometimes passes, the developers must include the aspect and execute the test many times. The more executions, the more likely a problem is found. If there is no deadlock suspicion, developers can normally execute their tests without this aspect. However, automatic tools, such as Linux `crontab` command, must be used to invoke the execution of the tests for several times using this aspect to assure all tests are passing and that they are not getting blocked.

The use of aspects for threads management during debugging and testing aided tremendously in the project critic phase. Besides that, they were not very difficult to be developed since there was someone already familiar with AOP in the team. Using the developed aspects was kept easy as it was introduced in a transparent way through a different call to a tool already know by the development team (Apache Ant tool).

5. The Redesign of OurGrid

According to the analysis based on the aspects included in the code, we discovered that a reengineering process was necessary or we would lose the control of the code completely. The main goal of this process was to well isolate internal concerns (aspects) of the grid middleware and to better control the application threads. Some classes could be reused, but others had to be completely rewritten or created.

In this reengineering process, we have tried to identify patterns in the concerns implementations that would make evolution easier. Besides that, identified patterns could also be used for future concerns to be included on the middleware. For simplification, we will just consider the OurGrid broker (called MyGrid) redesign to illustrate our experience and share some ideas from this process.

The first concern to isolate was the user interface. In the early versions of OurGrid, users had to directly access remote objects through RMI [17] or Linux scripts, and they were always forced to change their grid applications when any interface (or script) changed or when a new interface was added. Besides, if the communication infrastructure changed, users had to make several changes in their application if directly accessing the code (which was the most common use). Moreover, it was not clear for the users which methods exposed by the remote interfaces should be used. Therefore, we have defined the `org.ourgrid.mygrid.ui.UIServices` interface to offer all MyGrid services. This interface is illustrated in Figure 3 and it can be accessed via Java or using script wrappers from the OurGrid distribution. MyGrid also offers a graphical user interface that accesses the `UIServices` services. More details about the UI services can be found in the OurGrid manual [35]. Besides isolating the user interface, we needed to isolate internal concerns of MyGrid. In this isolation, we needed to organize the application threads and minimize the probability of introducing deadlocks in the evolution of the software. In order to do that, we have modularized the solution and employed an event-based architecture for communication [7]. The salient feature of one event-based design is that a thread in a module never wanders into another module. The first step of this process was to identify the concerns that needed to be modularized in the broker. Then, we implemented each concern with a better control of its threads and made communication between them use solely events. The following concerns had to be isolated in MyGrid: scheduling, local grid machines provisioning (the local GuMP), and the execution of replicas.

In our implementation of the event-based architecture, we have noticed a pattern that was repeated in many appli-

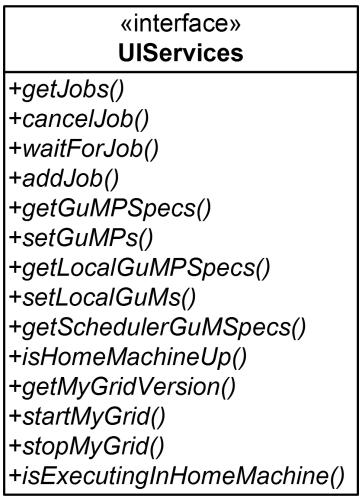

Figure 3. User Interface services offered via MyGrid

cation modules and that can probably be applied to other projects. It can be summarized as follows: the services provided by a module are offered through a Façade [14] class. This façade can be accessed by remote objects, by other façades, or by any element of the façade module. Each façade operation is converted into an event to a module, which is abstracted by the façade to its user. There is a contract that allows only one event in a module to be processed at a time, making the application threads more controllable. These events are processed by `EventProcessor`s classes and they use specific managers from each module, which are classes invoked by the events when they are processed. The event processing is performed using the Command design pattern [14] to make the `EventProcessor` a general entity. The basic elements of our pattern are therefore: the façade, the event processor, the events and the managers. Its dynamics is explained next with a concrete example that is illustrated by Figure 4.

In MyGrid, three modules were defined; all of them implement the pattern described above:

- Scheduler
- Local Grid Machine Provider
- Replica Executor

By well isolating these modules, we have separated three different aspects of the middleware: scheduling, local gums provisioning, and management of replicas' execution.

The *Scheduler* module is responsible for receiving the requests for the execution of jobs and allocating replicas of the tasks defined in these jobs for machines. The machines can be provided by the local GuMP (the *Local Grid Machine Provider*). After this allocation, the scheduler invokes the *Replica Executor* module to manage the execution of a chosen replica at a given machine. In order to receive machines from the local GuMP, users must define their grid machines and the way to access them. This is performed by invoking the *Local Grid Machine Provider* module. We will consider this operation to demonstrate the dynamics of the event-based pattern used in the MyGrid implementation that has proved to ease the evolution of the software. To do this, we will present the dynamics of a `setLocalGuMs` call, which is a method from the `UIServices` interface used to configure the local grid machines of the user. In order to do that, the implementer of this interface contacts a remote object, a `GuMManager`. Figure 4 illustrates this interaction. Every event to be processed by an `EventProcessor` implements the `ActionEvent` interface and therefore presents a `process()` method. In the creation of each event, such as the `SetGuMsRequestEvent`, illustrated by Figure 4, there must be an argument passed to the object that will actually perform the action represented by the event. In this figure, this object is the `RequestManager`.

With the event-based architecture divided in modules and following a pattern that was repeated in many places, maintenance had become easier and threads management too. There was a thread in each `EventProcessor` of a module and when other internal threads from each module were needed, they only changed the internal state of the module via the module façade (via an event). Besides isolating concerns, we had therefore isolated the threads and decreased the number of synchronized blocks (from 174 in version 2.1.3 to 110 in version 2.2). Another important observation is that many elements of the pattern implementation can be automatically generated, such as the `EventProcessor` and the basic structure of each `Event` used on a module.

Although we have focused on the MyGrid part of the OurGrid solution, the separation of concerns principle was also applied in other parts of OurGrid, making the system evolution possible and less stressing than in the past. We could have avoided redesign and made the isolation mostly using AspectJ, for example, with the code we had in the past. However, with this solution, we would be avoiding refactoring, which is necessary in several moments and cannot be replaced by AOP but aided by it.

Besides the redesign, another aspect that also made the software evolution better was the stronger focus on important XP practices that were not being followed during our development process, as we have discussed in Section 3.2.

6. Evaluation

In this section we evaluate the result of applying AOP in the redesign of OurGrid. We first present a comparison between the original and the redesigned versions of OurGrid. Then we present the lessons learned from the redesign process, evaluating the benefits it brought to the

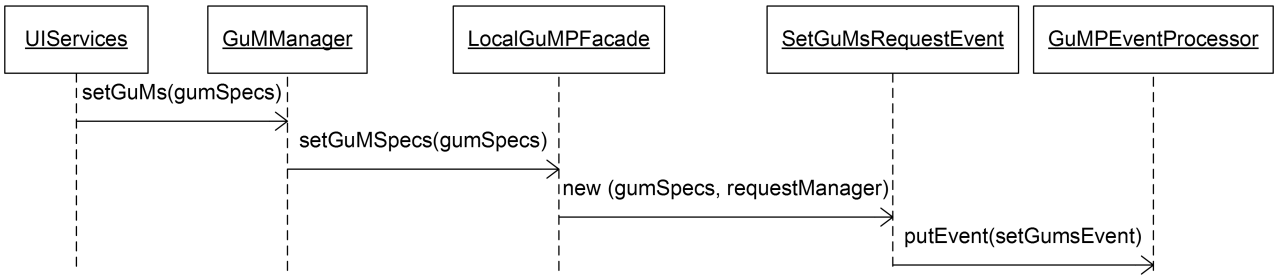

Figure 4. OurGrid event-based pattern dynamics

software and the team.

6.1. Comparative analysis

The first analysis we present is a comparison between the structure of the OurGrid code before the redesign and after separating the *crosscutting concerns*.

The aspect that triggered the rework was the thread management problem. Because the tasks of provisioning machines, scheduling and management of executions were scattered across the code, it was difficult to control threads related to each task. The implication was the complexity in finding deadlocks and correcting them since there were many synchronization blocks and different objects as locks for these blocks.

Before redesign, we can summarize the architecture using Figure 5. As can be seen, Remote Objects (objects that implement the `java.rmi.Remote` interface) received remote method invocations possibly from different threads. The threads that came from method calls to these objects could freely walk along the application packages. For each thread execution, several synchronized blocks were visited. In order to illustrate the complexity in extending and debugging OurGrid, Figure 6 shows a sequence diagram representing some internal method calls that were invoked when the user wanted to add a new grid machine to his personal grid. In this sequence of calls, many synchronization actions happened and different objects were used as locks. Initially, there is a synchronization action on the processors list managed by the `LocalGMProviderImpl` class. Then, there are calls to `getInstance` and `isAlive` methods from `GuMStateOracle`, which are synchronized. Synchronized methods are also called on the Processor class (`getGridMachine` and `active`). Besides that, there is also a `newProcessor` call, from the `RequestResponder` class, which is also synchronized.

In order to understand the problem, we made several drawings using different colors to identify threads behavior. In Figure 7, we illustrate one of the drawings that helped us understand all synchronized blocks (including methods) of OurGrid and the objects that were used as locks in these blocks. In order to isolate threads management in the redesigned version, developers had to follow an architecture based on the processing of events. Before redesign, it was really hard to know why a deadlock was happening. We have designed two aspects to help in identifying these problems. The `ThreadSleeper` aspect and the `RunningThreadsMonitor` aspect helped us improve the quality of our tests with multithreaded code. However, correcting an identified deadlock was really hard, because we had many synchronization blocks and different objects as locks for these blocks.

As we can observe, a reengineering process was necessary in order to better isolate OurGrid aspects. In AOP methodology, the crosscutting concerns are modularized by identifying a clear role for each one in the system, implementing each role in its own module, and loosely coupling each module to only a limited number of other modules [22].

After redesign, we created a structure to be followed by developers, clearly defining the application modules and the way these modules should interact. In order to better isolate thread management, which was spread throughout the code using synchronized blocks, we have reduced these blocks and we have used an event-based architecture. For each module, every invocation performed to a remote object by a different thread was redirected to the correspondent module façade. The façade then created an event, depending on the method called on it, which would be processed later by the `EventProcessor` of that module. The necessary classes from the module would be invoked in a secure way since only one event per module would be invoked at a time, changing the module state in a consistent way. Figure 8 summarizes the architecture after redesign considering threads execution in a general way. An instantiation of a thread execution in the redesigned software was shown in Figure 4. Although we could obtain a better separation of concerns and a better management of application threads, we had some impacts on some metrics, which were calculated using the Eclipse Metrics plugin [1]. They are summarized in Table 1.

One of the measures considered was the number of classes. It has grown from 277 to 427, representing an increase of 54.15%. This happened because we needed a class for each event to be processed in a module instead of a single method used in the previous versions. However,

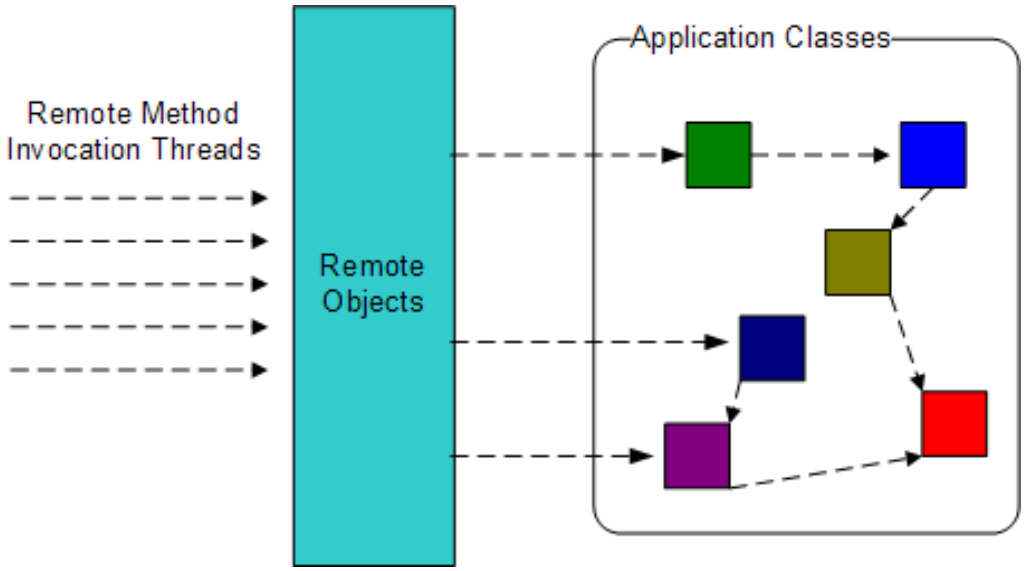

Figure 5. OurGrid before redesign

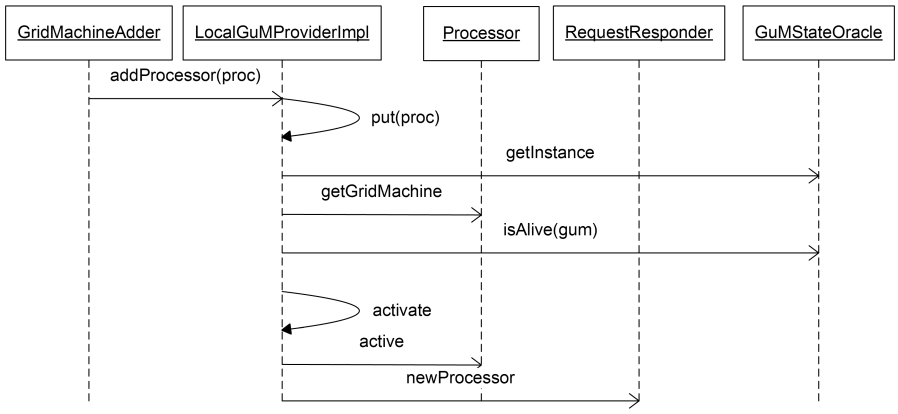

Figure 6. Adding a grid machine before redesign

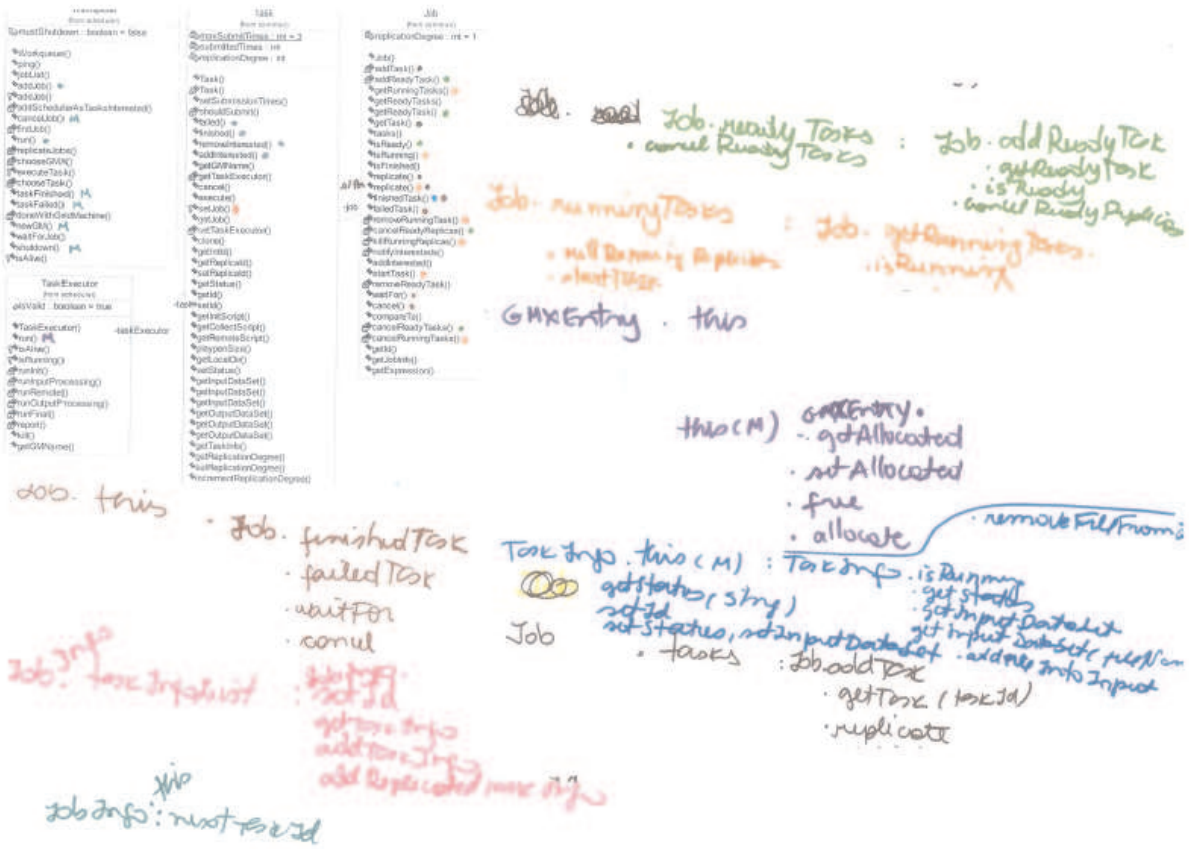

Figure 7. Attempt to understand OurGrid threads before redesign

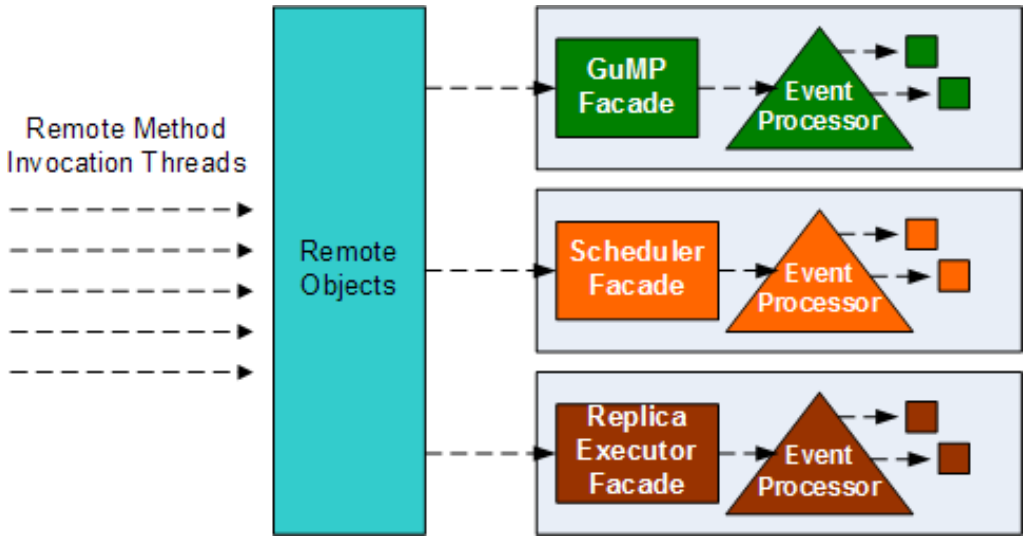

Figure 8. OurGrid after redesign

	OG 2.1.3	OG 2.2
Total Lines of Code	33156	34734
Number of classes	277	427
Afferent Coupling	21.577	20.721
Efferent coupling	9	8.581
Lack of cohesion of methods	0.339	0.249
Number of synchronized blocks	174	110

Table 1. OurGrid 2.1.3 and 2.2 comparison

part of the code regarding events creation and processing can be automatically generated[2]. Regarding the total number of lines of code, they had only increased 4.75%.

The Afferent Coupling average, which is the number of classes outside a package that depend on classes inside the package, had decreased 4% after redesign. The Efferent Coupling average, which indicates the number of classes inside a package that depend on classes outside the package, had decreased 4.6%. These metrics are interesting, but do not completely express the gains obtained regarding less coupling between classes, as these metrics are based on packages. The Lack of Cohesion of Methods(LCOM*) is a better metric as it considers methods and attributes. It is a measure for the Cohesiveness of a class calculated with the Henderson-Sellers method [19]. If m(A) is the number of methods accessing an attribute A, this metric calculates the average of m(A) for all attributes, subtract the number of methods m and divide the result by (1-m). A low value indicates a cohesive class. We obtained that the redesigned version LCOM* value is 26.54% lower than the previous one.

Besides these metrics, we have considered that the reduction of synchronized blocks was the better result for us, which had made programming more secure regarding the danger of including deadlocks. The number of synchronized blocks was 36.78% smaller after redesign.

6.2. Lessons learned

We have learned some lessons while applying AOP to a project that was in trouble due to the complexity in threads management. Next we analyze them in a generic form so that they can be applied to other projects in similar situation.

Use AOP to get a better control of multithreaded code: We have lost the control of the multithreaded code. Our first step was to find the easiest way to identify the problems regarding threads execution. Our approach was to provide a set of tools for testing that would make it easy to identify bugs regarding certain threads execution order and without directly changing the code. With such approach we could deliver a better version of OurGrid to final users, without many of the problems that existed. After that, we have made the redesign to better isolate the application concerns and specially its threads management, through an event-based architecture.

[2]Code generation for event handling is expected for OurGrid 4.0

Avoid developers' resistance of introducing a new technology, such as AOP, during a critical phase of a project by introducing it in a transparent way: If developers have a little time to finish up something and a new technology is introduced, there might be some resistance. We have avoided that resistance by introducing Aspect-Oriented Programming use as a new call to the Ant tool that would replace `ant compile` call and by providing a class with `Threads` utilities that looked like a normal Java class. Behind it, there was an aspect responsible for performing this class functionality due to its power of knowing the application threads states. With such approach, developers used AOP without compromising too much time in learning a new technology in a critical phase of the project.

Automatic tests are vital to get a software project back in shape and AOP provides a good support in the development of automatic tests: Developing tests for grid computing solutions is hard, especially because sometimes they impact in the implementation, since the code needs to be changed. By using AOP, it is possible to have more testable applications without directly changing them, improving test quality. For example, the `waitUntilWorkIsDone` method that we have provided helped us to test multithreaded code.

Really follow the development process you have adopted: We had serious problems in integration. In 2003, we had 6 different groups implementing new features for OurGrid, each one with a different version of the system. These versions started to be integrated at the end of the year in a serial manner. Each integration process was hard and the existing tests were of bad quality and they took too much time to execute. The code being integrated was not known by all the development team, making maintenance very difficult. Sometimes, only one person knew part of the code because pair programming was not used during development. In fact, we were using an XP-based process, but we were not following most of its practices. From the XP practices [5], the ones that caused more damage for not being followed were:

- **Continuous integration:** New code is integrated with the current system after no more than a few hours. When integrating, the system is built from scratch and all tests must pass [5].
- **Testing:** Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished.
- **Collective ownership:** Anyone can change any code anywhere in the system at any time.

- **Pair programming:** All production code is written with two programmers at one machine.
- **Refactoring:** Changes in the code to improve its design and that do not change its functionality.

As we gave a greater attention to these practices, we could get control about our code again and improve developers' confidence with it. Besides that, due to these practices, the redesign process was faster than expected.

7. Related Work

Several aspects of AOP have been subject of research. However, we are particularly interested in comparing our work with those projects that have applied AOP as the solution for reorganizing existing code.

Although general aspects of refactoring and AOP have been addressed in several works (e.g., [6, 38]), we needed some real examples to be able to compare the results of our efforts. Scenarios where real systems had to be (re)organized vary from interface implementations [37], reduction of middleware complexity [41], experiment with AOP in large scale middleware [11], comparison of the use or not of AOP in a component-based web crawling system [26] and others ([4, 28, 32]). None of them, however, deal with the reorganization of a system in a critical phase of a project.

An interesting result was presented in Coady & Kiczales [10] where parts of the FreeBSD operating system were refactored and the result was a software better organized and easier to evolve and maintain, similar to the results we found.

Although there are methodologies for finding crosscutting aspects, we did not explore them since we had a very specific initial need. In this area, proposals such as concern graphs [29] and aspect browser [16] could help us in the process of finding more crosscutting aspects in addition to the specific issues we have selected based on practical experiences.

Even though our selection of aspects was ad-hoc, we addressed a very interesting issue in our AOP experience with OurGrid. This same issue was identified in the literature. Schwanninger el al [31] have pointed out that software often present one or more crosscutting concerns, including optimization of resource management, e.g. memory management or thread management. Gibbs & Coady [15] present a case study where AOP has been applied in a memory management system to help flexibility in terms of evolution and adaptation. Walker et al [39] present an experiment with the objective of identifying if AOP could be used to help bug finding and fixing in multithreaded code. More specifically, a methodology for testing multithreaded programs was proposed in [13]. The method proposes reruns of existing tests in order to detect synchronization faults. In our work, besides rerunning the existing tests, we have introduced sleep calls in several points of the execution that varied in time, increasing the probability of different threads configurations for each test run. The interesting point was to use aspects, which made this implementation easier, modularized and did not introduce complexities to the normal application code.

Another work [23] proposes the use of aspects in the testing process. The focus is on using aspects to avoid changing the code just to make the implementation of a test possible, especially considering the use of a technique called Mock Objects for isolating application units during tests. We have also explored aspects for tests in our work, focusing in testing multithreaded code. We have also created a general package for thread management that can be used for testing, debugging and even in the normal execution of the application if such management is necessary.

Other works, such as [36], also present their cases of grid middleware development. In our work, besides presenting our history, and how we got into trouble during the development, we also provided a useful technique for separation of concerns in such systems that focused on isolation of the user interface and on the use of an event-based pattern in the implementation of each concern module. Besides that, we have provided reusable aspects and classes to aid testing and debugging.

Event-based communication is not new for large-scale distributed applications [7]. As this style brings some complexity to the code, we have tried to simplify the programming model by providing Façades that have hidden the use of events from the users of the modules.

8. Conclusions

We have concluded that aspect-oriented programming has been a useful mechanism in the maintenance and reengineering process of OurGrid. Besides helping in the debugging process and improving the quality of tests for multithreaded code, the focus on separation of concerns, especially crosscutting ones, was very important for better designing our software.

Modularized implementation of concerns results in easier-to-understand and easier-to-maintain systems [22]. When new concerns emerge, refactorings may be done to guarantee we keep the code simple and understandable. Besides that, changes occur all the time in a research project, requiring good automatic tests. Aspects helped in improving the testing of parallel and distribute applications. By using AspectJ, we have provided services for controlling the state of application threads and for making existing deadlocks of the code more prone to happen during the execution of tests.

We have also observed that the isolation of the user interface is really important in order to provide a software evolution that does not harm so much the developer. Besides that, we also believe that the pattern we have used in each OurGrid module can be easily applied to other

concerns that can come in the future for the project and even for other projects based on multithreaded code. As a future work, we want to apply the pattern for other parts of the middleware being developed and formally describe this pattern. We also plan to have automated support to it.

Another future work is to explore AOP in other aspects of grid computing. We believe AOP can improve the implementation of some crosscutting concerns such as grid monitoring, failure detection, accounting and adaptability. The implementation of these concerns is difficult to modularize using pure object-oriented techniques today and if not well modularized, will lead to maintainability problems when incorporated into the code. As there are open issues in grid architecture, we believe AOP can give an important contribution in this field.

9. Acknowledgments

We would like to thank the whole OurGrid team, in particular to Erica Gallindo and Lauro Costa, for important comments and suggestions to this paper. We also thank the anonymous reviewers for the insightful comments and questions. This work has been developed in collaboration with HP Brazil R&D.

References

[1] Eclipse Metrics plugin. At http://metrics.sourceforge.net/.

[2] Nazareno Andrade, Francisco Brasileiro, Walfredo Cirne, and Miranda Mowbray. Discouraging Free Riding in a Peer-to-Peer CPU Sharing Grid. In *Proceedings of the 13th High Performance Distributed Computing Symposium (HPDC'2004)*, 2004.

[3] Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro, and Paulo Roisenberg. OurGrid: An approach to easily assemble grids with equitable resource sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[4] Elisa L. A. Baniassad, Gail C. Murphy, Christa Schwanninger, and Michael Kircher. Managing crosscutting concerns during software evolution tasks: an inquisitive study. In *Proceedings of the 1st international conference on Aspect-oriented software development (AOSD'02)*, pages 120–126, April 2002.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[6] Paulo Borba and Sergio Soares. Refactoring and code generation tools for AspectJ. In *Proceedings of the Workshop on Tools for Aspect-Oriented Software Development (with OOPSLA)*, Seattle, Washington, USA, November 2002.

[7] Antonio Carzaniga, Elisabetta Di Nitto, David S. Rosenblum, and Alexander L. Wolf. Issues in Supporting Event-based Architectural Styles. In *Proceedings of the Third International Software Architecture Workshop (ISAW-3)*, November 1998.

[8] Walfredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro Costa, Alisson Andrade, Reynaldo Novaes, and Miranda Mowbray. Labs of the World, Unite!!! Accepted for publication in Journal of Grid Computing <http://www.springerlink.com/link.asp?id=111140>. Springer. 2006.

[9] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques Sauvé, Fabrício Alves Barbosa da Silva, Carla Osthoff Barros, and Cirano Silveira. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. In *Proceedings of the International Conference on Parallel Processing (ICCP'2003)*, October 2003.

[10] Yvonne Coady and Gregor Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD'03)*, pages 50–59, March 2003.

[11] Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD'04)*, pages 56–65, March 2004.

[12] Daniel Paranhos da Silva, Walfredo Cirne, and Francisco Vilar Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Proceedings of the Euro-Par 2003: International Conference on Parallel and Distributed Computing*, pages 169–180, 2003.

[13] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for Testing Multi-threaded Java Programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

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

[15] Celina Gibbs and Yvonne Coady. Aspects of Memory Management. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 9*, Big Island, Hawaii, 2005.

[16] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 265–274, 2001.

[17] William Grosso. *Java RMI*. O'Reilly, 2001.

[18] Erik Hatcher and Steve Loughran. *Java Development with Ant*. Manning Publications Co., 2004.

[19] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1995.

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

[21] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect–Oriented Programming. In *European Conference on Object–Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, Finland, June 1997. Springer–Verlag.

[22] Ramnivas Laddad. *AspectJ in Action*. Manning Publications Co., 2003.

[23] Nicholas Lesiecki. Test flexibly with AspectJ and mock objects. At ftp://www6.software.ibm.com/-software/developer/library/j-aspectj2.pdf.

[24] Vincent Massol. *JUnit In Action*. Manning Publications Co., 2004.

[25] Raissa Medeiros, Walfredo Cirne, Francisco Brasileiro, and Jacques Sauvé. Faults in Grids: Why are they so bad and What can be done about it? In *Proceedings of the 4th International Workshop on Grid Computing (Grid 2003)*, November 2003.

[26] Odysseas Papapetrou and George A. Papadopoulos. Aspect Oriented Programming for a component based real life application: A case study. In *Proceedings of the ACM Symposium on Applied Computing*, Nicosia, Cyprus, March 2004.

[27] Manish Parashar and Craig A. Lee. Scanning the Issue: Special Isssue on Grid-Computing. *Proceedings of the IEEE*, 93(3):479–484, March 2005.

[28] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD'03)*, pages 120–129, March 2003.

[29] Martin P. Robillard and Gail C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE´02)*, Orlando, Florida, USA, 2002.

[30] Elizeu Santos-Neto, Walfredo Cirne, Francisco Brasileiro, and Aliandro Lima. Exploiting Replication and Data Reuse to Efficiently Schedule Data-intensive Applications on Grids. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2004.

[31] Christa Schwanninger, Egon Wuchner, and Michael Kircher. Encapsulating Crosscutting Concerns in System Software. In *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Lancaster, UK, March 2004.

[32] Sergio Soares, Eduardo Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)*, pages 174–190, Seattle, Washington, USA, November 2002.

[33] Peri Tarr and Harold Ossher. Advanced Separation of Concerns in Software Engineering. In *Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001*, 2001. At http://www.research.ibm.com/ hyperspace/workshops/icse2001.

[34] AspectJ Team. The AspectJ Programming Guide. At http://www.eclipse.org/aspectj, 2003.

[35] OurGrid Team. Ourgrid 3.0 user manual. At http://www.ourgrid.org.

[36] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 2004.

[37] Paolo Tonella and Mariano Ceccato. Migrating Interface Implementation to Aspects. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, Chicago Illinois, USA, 2004.

[38] Arie van Deursen, Marius Marin, and Leon Moonen. Aspect mining and refactoring. In *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE)*, Waterloo, Canada, November 2003.

[39] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. Assessing Aspect-Oriented Programming and Design: Preliminary Results. In *Workshop on Aspect-Oriented Programming (In ECOOP'1998)*, Brussels, Belgium, July 1998.

[40] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An Initial Assessment of Aspect-Oriented

Programming. In *Proceedings of the 21st International Conference on Software Engineering*, pages 120–130. IEEE Computer Society Press, 1999.

[41] Charles Zhang and Hans-Arno Jacobsen. Refactoring Middleware With Aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11), 2003.