Aspect Testing Framework

Daniel Hughes and Philip Greenwood

Computing Department, Lancaster University,
Lancaster, LA1 4YR. UK
d.r.hughes@lancaster.ac.uk | p.greenwood@lancaster.ac.uk

Abstract. Testing is a vital stage in the development cycle of any application but is often neglected due to the difficulty and expense to perform the stage successfully. This is especially so for distributed applications due to the co-ordination required to successfully test several distributed components simultaneously. This position paper proposes a framework implemented using Aspect-Oriented Programming and Reflection, which aims to ease the testing of such systems and other varieties of systems which encounter similar problems. Two case-studies are examined to illustrate how the framework will simplify the testing stage.

1. Background

Software testing is an important part of the development process; however, it is both difficult and time consuming and therefore often goes neglected. Distributed software engineering is one of the most high pressure areas of software engineering. This is due to the increased complexity of developing systems that run in distributed (and often unpredictable) environments, coupled with software development cycles which are increasingly restricted by the need to get products to market quickly. This often leads to sloppy testing procedures and low quality products. Distributed software engineering poses some specific problems:

- Monitoring of many distributed components running simultaneously.
- Insertion and removal of custom monitoring code.
- Reuse of testing code in other applications

An Aspect Oriented Testing Framework will be proposed in this document, which aims to address these problems. This system should support the creation and maintenance of a distributed testing environment; allowing the behaviour of processes, which may potentially be running on remote machines, to be monitored from a central interface.

In order to ensure high quality distributed software, testing in a genuinely distributed environment is often necessary. This is because it is not usually possible to simulate enough nodes on a single machine due to the high CPU usage and network intensive nature of many distributed applications. This problem will only increase as the

number of users participating in distributed communities grows. To thoroughly test the effectiveness of distributed systems for such communities, testing with ever greater numbers of nodes will be required.

Manual creation and maintenance of such tests is an extremely time consuming activity, especially where nodes are required to change their behaviour dynamically. This often requires the hand-coding of system monitoring programs to record the behaviour of nodes from a central point. Furthermore, communications code is often spread throughout a distributed application making the addition of monitoring code extremely time consuming and error-prone. To facilitate the efficient insertion and removal of monitoring to accomplish this, we expect to use a combination of Reflection [7] and Aspect Oriented Programming (AOP) [2].

2. Aspect Oriented Programming

AOP is an emerging programming paradigm which extends Object-Oriented Programming (OOP) and claims to improve certain areas where OOP fails. The purpose of OOP is to allow the programmer to cleanly capture a single piece of functionality or concern in an encapsulated object, only exposing features via an interface. Suppose however that a concern can not be cleanly captured in a single object, this would normally result in the concern being spread out over several objects with sections of code implementing the concern contained in each of these objects. This leads to several problems: the code is less maintainable, the readability of the code is diminished and the encapsulation provided by OOP is lost.

AOP aims to solve these problems by allowing these *crosscutting* concerns to be cleanly captured in one self-contained unit of code. Concerns are implemented in AOP by using units of code called *aspects*. Aspects contain pieces of code called *advice* which are used to implement the crosscutting concern and the places where the advice should be applied to the OOP base-code are defined using *joinpoints*. A *weaver* is used to combine the AOP code with base-code so the appropriate links can be inserted at the places within the base-code specified by the joinpoints to reference the appropriate aspect-code. Typical examples of such crosscutting concerns which could be implemented using AOP are: security, synchronisation and tracing. This paper will concentrate on implementing a tracing concern used for system monitoring.

When tracing a piece of code a programmer normally inserts a number of print statements to trace the flow of the program and they can output them to screen, save them to a file or send them to a central server (if it is a distributed application). However this can be both time consuming and unreliable since some statements could be missed from a vital section of code. Furthermore once the tracing has been completed the print statements need to be removed which again is time consuming and potentially dangerous as deconstructive changes are being made to the code. Implementing a tracing concern such as this, using AOP, will allow the trace statements to be easily added and then later removed due to the ability to easily weave and un-weave the aspect code without needing to modify the base-code.

AOP Languages

There are several AOP languages available which are compatible with Java (our preferred development language) such as AspectJ [5] [6], JAC [10] and Hyper/J [9]. AspectJ is a relatively simple language extension to Java which uses static weaving; when the aspects are weaved with the base-code at compile time. JAC is similar to AspectJ in that it uses similar concepts but does not implement a language extension and uses dynamic weaving which means the aspects can be applied while the base-code is being executed; however in this case it is an unnecessary overhead. Hyper/J is very different to the previous two languages in that it requires the structure of the program to be carefully constructed and the inheritance/interfaces of the objects to be defined thoroughly. These constraints imposed by Hyper/J make it unsuitable for our needs. AspectJ will be our chosen aspect language due to its simplistic nature, good compatibility with Java and the ease with which it allows aspects to be defined.

The joinpoint model in AspectJ allows advice to be attached to such places as method calls, method execution, method reception, field gets, field sets and exception handlers. The advice can also be specified to be executed before, after or even around these joinpoints. AspectJ introduces the concept of *pointcuts* which are collections of joinpoints. Advice defined in AspectJ is similar to Java method constructs and can be attached to pointcuts so that they are executed at the appropriate places.

One of the main problems when using AOP is identifying which joinpoints are present in a particular application. This is especially troublesome for 3rd party objects or when the source code for an application is no longer available. This problem will hamper the reuse of aspects as the joinpoints specified in the aspect may only suit one particular application and destructive changes may have to be applied to either the aspect-code or the base-code in order to allow an aspect to be compatible with other applications.

What is required is some kind of framework which is able to examine the object for which we are interested in weaving an aspect with and then to use the information gathered to customise the aspect to suit the object.

Reflection

One way which this can be achieved is by using reflection. Reflection is defined as "The capability of a system to reason about and act upon itself. A reflective system contains a representation of its own behaviour, amenable to examination and change, and which is causally connected to the behaviour it describes. Causally connected means that: changes made to the system's self representation are immediately reflected in its actual state and behaviour and vice versa" [1]. In this instance reflection will only be used to examine and extract information regarding the structure of the objects.

Java implements its own Reflection API which makes this process much easier. The Reflection API represents the classes, objects, interfaces currently loaded in the executing JVM. Several operations are possible using the Reflection API such as: determining the class of an object, extract information about a class's methods, fields constructors, inheritance and information regarding interfaces.

As can be seen, the Reflection API is a very powerful tool and fits our need for being able to extract information about a class. What is now needed is some kind of framework which will allow this information to customise aspects in order to change the joinpoints and advice of the aspects to suit a wider range of applications.

What we propose to use, in conjunction with Reflection, is a template structure which will incorporate the AspectJ code and our own custom tags which will identify the parts of the aspects which are 'customisable'.

Tags and Templates

We anticipate that in the first instance the framework will load the object chosen by the programmer to be examined and the reflection API will be used to extract certain information about the object. This information will consist of: class types, fields, method signatures, constructors, return values and parameter lists; elements which can be used in the joinpoints of AspectJ. The list of elements found will be presented to the user where they will be able to construct the joinpoints and select which advice/template should be used to construct the joinpoints.

The benefit of using the template and tags will allow the programmer to create a normal aspect using the language extensions which AspectJ provides and then simply substitute the parts of the aspect that need to be customised to suit different applications with the appropriate tag. Additionally, the advice in the aspect can also make use of these tags to access elements of the object such as fields or parameters. For example the following creates a piece of advice which should be executed before the method call on the method foo within the class c:

```
before() : call(void c.foo()){

}
```

Suppose that the same piece of advice is going to be used in a number of applications all of which have the class c but foo has a different implementation and to reflect this is called foo2 instead. Normally the programmer would either have to maintain two copies of this aspect or continually have to make changes to switch between the versions of foo. Instead using our framework the user could simply create a template using the tag <METHODNAME> to generalise the method name and then use the framework to switch between the two.

```
before() : call(void c.<METHODNAME>){

}
```

Or to take it a stage further the classname which the method belongs to could be generalised:

```
before() : call(void <CLASSNAME>.<METHODNAME>){
}
```

The framework will use the reflection API to examine the classes and present to the user all the potential joinpoints. The programmer can then select the class, methods, fields etc. to substitute the tags with.

Work needs to be done to expand the tags used as the code samples given here are merely examples of how the tags could be used. The framework will also need to be integrated with the Java compiler in order to ease the effort of customising the aspects to suit the base-code.

The proposed system would have the following advantages over manual testing of distributed systems:

1. Automatic insertion and removal of necessary testing code.
2. Support for automated monitoring of the state of many distributed components.
3. More rapid deployment of test scenarios.
4. Easy customisation of test cases
5. Reuse of test code
6. Ensures the correctness of test code

This is not the first time that Aspects have been used for a testing/tracing purpose. The Atlas project [4] found that using print-line statements or a normal debugger are not effective when debugging a servlet, but instead found that AOP was an ideal solution. In this work the tracing aspects were created manually. In contrast, the work described in this paper extends this manual approach and aims to use Aspects in the same way but to have them created automatically.

3. Example Scenario – Testing "AGnuS: The Altruistic Gnutella Server"

Consider the following testing scenario for evaluating "AGnuS: The Altruistic Gnutella Server" [3] AGnuS is a specialised Gnutella node which layers content based routing, load balancing, caching and file filtering on top of the core Gnutella Protocol. Gnutella is a decentralised file sharing protocol. Gnutella nodes perform all functions on the network; downloading and serving files and routing all messages, unlike traditional file-sharing services such as Napster [8] which rely upon central servers to process requests. Thorough manual testing of AGnuS is difficult for the following reasons:

Difficulty of Scalability Testing
One of the key problems with decentralised peer-to-peer networks is scalability. Any small change in the Gnutella algorithm has the potential to dramatically affect the scalability of the network. For example, not implementing the time to live (TTL) value which segments the network would destroy its scalability. Without the TTL value and the resultant hop-limit, a simply 80 byte query e.g. "Grateful Dead Live" broadcast on a network of users similar in size to that which Napster supported would require more than 160MB of data to be sent over the network. In order to reveal such potentially catastrophic performance issues, it is essential to be able to simulate networks of an appropriate size. Simultaneous monitoring of the required number of nodes is not possible manually.

Understanding the Effects of Compound Node Interactions
Another significant problem in the evaluation of complex distributed systems such as AGnuS is that the emergent structure and behaviour of the network are not always predictable based on the algorithms used to generate them. User behaviour and node failure can have unpredictable effects on the network as a whole. To thoroughly test the behaviour of the network, monitoring of individual nodes or groups of nodes throughout the network may be required, this is not possible manually.

Testing the Effect of Node Failures
As nodes in a peer-to-peer network such as this one are running on normal general-purpose workstations, which are potentially insecure and fallible, node failure must be anticipated and its effects on the network analysed. As with testing compound node interactions, this is difficult as it requires the monitoring of many distributed components simultaneously.

One example where the Aspect Testing Framework could be used, to increase the efficiency of testing for AGnuS, is in testing the accuracy and performance of the Content Based Routing System. AGnuS' Content Based Routing system parses incoming messages, categorises them according to their type and then routes them to the most appropriate area of the network: First the method **`getQueryType(String query)`** is called. This method returns a type constant. Based on the constant returned, one of the following methods is called:

```
routeAudioQuery(String query)

routeVideoQuery(String query)

routeImageQuery(String query)

routeTextQuery(String query)

routeSoftwareQuery(String query)
```

These methods route an incoming message to the area of the network that is richest in that data type. For Example: Running getQueryType on the incoming query "Elvis MP3" would return its type as 'AUDIO'. Based on this classification, the query will be forwarded using the routeAudioQuery method, which directs it to peers known to be rich in this file type. It is not possible to manually monitor enough simultaneously executing nodes, therefore monitoring code must be inserted throughout the content based routing system for evaluation.

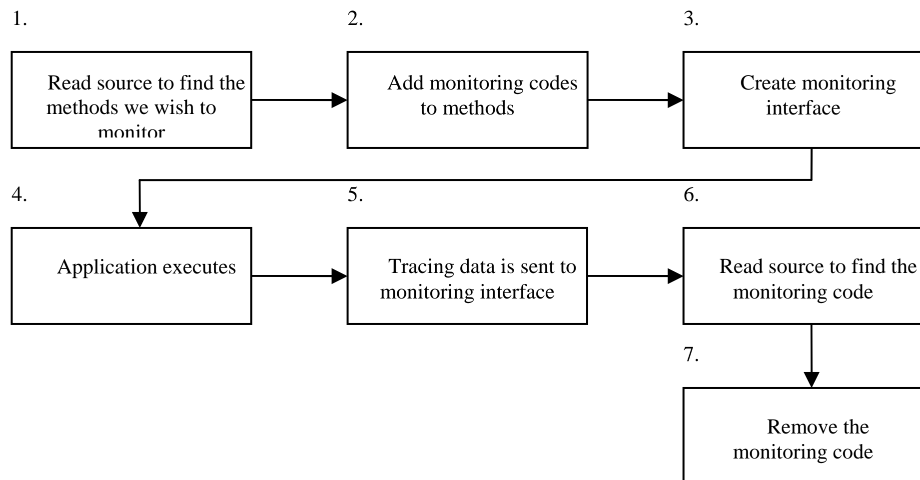

Figure 1 Hand generation and removal of monitoring code

The proposed Aspect Testing Framework simplifies the problem of testing complex distributed systems such as AGnuS by automating the insertion/removal of monitoring code. Furthermore, the ability to easily add, modify and remove communications code makes it easy to tailor communications to fit any monitoring interface, thus facilitating the re-use of interface code.

Insertion and removal of monitoring code, especially where it needs to be spread throughout the program is a time-consuming task. The automation of this process and the re-use of monitoring code and the central monitoring interface should significantly reduce the time required to thoroughly test complex distributed systems.

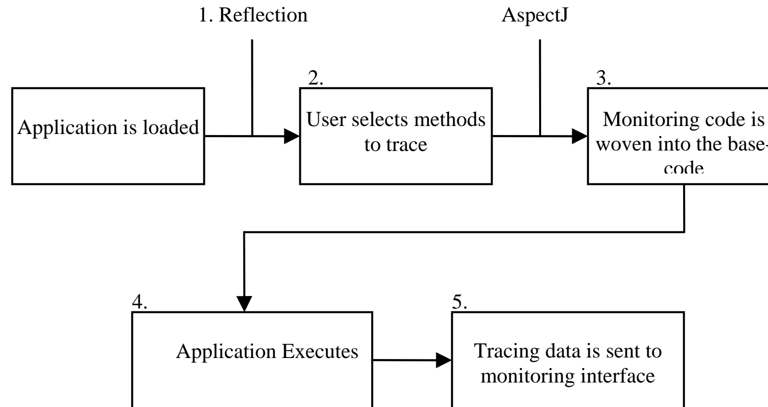

Figure 2 Automated insertion and removal of monitoring code

1. Java Reflection
Use of the Java reflection API allows the user to inspect the structure of AGnuS' content based routing System.

2. User Selects Methods to Trace
Presented with a list of the methods contained within the program, the user is able to select those methods which they wish to monitor.

3. Monitoring Code is Compiled into the Program
AspectJ uses the templates discussed in section 2 to compile monitoring code into the application.

4. Program Executes
As the program executes the tracing code woven into the application sends monitoring information back to the Monitoring Interface.

5. The System Monitoring Interface
It is now possible to monitor a large number of AGnuS nodes running simultaneously, potentially on a distributed environment from a single interface.

Performance Measuring
Performance measuring is a variation of testing which is suited to be implemented using the proposed framework. Performance measuring of applications is often implemented by inserting code into an application to count the number of times a particular event occurs or to time how long a certain action takes to execute. As in the distributed application example described earlier one of the major drawbacks of performing this task is adding and removing the code to perform the measuring. Another difficulty encountered in implementing this task is customising the

performance measuring code to count and time the desired events. This is especially difficult as each different application will have different events that need measuring.

The AOP framework proposed can potentially solve both of these problems. As in the distributed application example the code required to perform the measuring can be easily added or removed due to the weaving process used by AOP. Additionally using the framework to implement this should promote the reuse of aspects which perform the measuring, as they can be easily applied to other applications by simply selecting the events via the GUI presented to the programmer which have been gathered by using Reflection. This relies on the premise that the events fit into the joinpoint model of AspectJ.

The selected elements are interpreted into joinpoints and inserted at the correct places in the aspect template. The joinpoints are attached to advice which will carry out the performance measuring such as counting or timing the selected events. The completed aspect is then woven with the base-code so that the performance measuring can take place when the application is executed and the joinpoints (which represent the selected events) are reached.

4. Summary

The Aspect Testing Framework seeks to facilitate the testing of systems by providing support for automated monitoring of the state of many (potentially distributed) components. The Aspect Testing Framework will accomplish this by automating the process of inserting and removing testing code. We present two scenarios; one derived from testing 'AGnuS: The Altruistic Gnutella Server' and a Performance Monitoring example which clearly demonstrate the potential improvements the framework provides over traditional testing methods for distributed and non-distributed software.

The Aspect Testing Framework encourages re-use of test code: the same test code and central interface can be used to monitor many different kinds of applications.

The facility to rapidly insert and remove test code throughout any application should enable more rapid creation of test scenarios; potentially alleviating the problems caused by the spiralling levels of software complexity and shrinking development schedules.

Furthermore, as test-code is compartmentalised and separate from the system which it is to monitor, it will be easier to ensure its correctness. This also aids the removal of the code from the system after testing which ensures that the correctness of the final system is not compromised by forgotten test-code.

References

[1]Coulson, G., "What is Reflection?", http://dsonline.computer.org/middleware/, 2003.

[2] Elrad, T. et al, “Discussing aspects of AOP”, Communications of the ACM Vol. 44 No. 10 pp 33-38, 2001.
[3] Hughes, D. et al, “AGnuS: The Altruistic Gnutella Server”, proceedings of the Third international conference on peer-to-peer Computing pp202 – 203, Linköping Sweden, 2003.
[4] Kersten M., G. C. Murphy, Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-Oriented Programming. 1999.
[5] Kiczales, G. et al, “Getting Started with AspectJ”, Communications of the ACM Vol. 44 No. 10 pp 59-65, 2001.
[6] Kiczales, G. et al, “An Overview of AspectJ”, Proceedings of ECOOP pp 327-353, 2001.
[7] McCluskey G., “Using Java Reflection”, Java Developer Connection, 1998.
[8] Napster, “Napster Home Page”, http://www.napster.com, 2003.
[9] Ossher, H., Tarr, P., “Multi-Dimensional Separation of Concerns using Hyperspaces”, IBM Research Report 21452, 1999.
[10] Pawlak, R. et al, “JAC: A Flexible Solution for Aspect-Oriented Programming in Java”, Reflection 2001, 2001.