ISTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

ANALYSIS of ASPECT ORIENTED PROGRAMMING BASED PROFILING of REAL TIME JAVA APPLICATIONS

M.Sc. Thesis by
Ilker AKKUS

Department : Computer Engineering

Programme : Computer Engineering

Thesis Supervisor: Prof. Dr. Nadia ERDOGAN

JUNE 2010

ISTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

ANALYSIS of ASPECT ORIENTED PROGRAMMING BASED PROFILING of REAL TIME JAVA APPLICATIONS

M.Sc. Thesis by
Ilker AKKUS
(504071740)

Date of submission : 07 May 2010
Date of defence examination: 14 June 2010

Supervisor (Chairman) : Prof. Dr. Nadia ERDOGAN (ITU)
Members of the Examining Committee : Assist. Prof. Feza BUZLUCA (ITU)
Assist. Prof. Yunus Emre SELCUK (YTU)

JUNE 2010

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

GERÇEK ZAMANLI JAVA UYGULAMA PERFORMANSININ İLGİYE YÖNELİK PROGRAMLAMA MODELİ İLE PERFORMANS ANALİZİ

YÜKSEK LİSANS TEZİ
İlker AKKUŞ
(504071540)

Tezin Enstitüye Verildiği Tarih : 07 Mayıs 2010
Tezin Savunulduğu Tarih : 14 Haziran 2010

Tez Danışmanı : Prof. Dr. Nadia ERDOĞAN (İTÜ)
Diğer Jüri Üyeleri : Yrd. Doc. Dr. Feza BUZLUCA (ITU)
Yrd. Doc. Dr. Yunus Emre SELCUK (YTU)

HAZİRAN 2010

FOREWORD

I would like to express my deep appreciation and thanks for my advisor. This work is supported by ITU Institute of Science and Technology.

June 2010

Ilker AKKUS

Computer Engineer

TABLE OF CONTENTS

Page

ABBREVIATIONS

AJC	**:** AspectJ Compiler
AOP	**:** Aspect Oriented Programming
API	**:** Application Programming Interface
CPU	**:** Central Processing Unit
IDE	**:** Integrated Development Environment
JAAOP	**:** Java Aspect Oriented Profiler
JRAT	**:** Java Run Time Analysis Toolkit
JVM	**:** Java Virtual Machine
JVMPI	**:** Java Virtual Machine Profiler Interface
JVMPTI	**:** Java Virtual Machine Tool Interface
OOP	**:** Object Oriented Programming
PARC	**:** Palo Alto Research Center
XML	**:** Extensible Markup Language

LIST OF TABLES

Page

LIST OF FIGURES

Page

ANALYSIS of ASPECT ORIENTED PROGRAMMING BASED PROFILING of REAL TIME JAVA APPLICATONS

SUMMARY

Performance is usually a major quality characteristic in software development. Although it can vary depending on the type of software, common performance indicators are memory allocation amount, method execution time and processor usage ratio. For better performing software, it is required to measure those indicators during development. This measuring process is called profiling and can easily reveal the hotspots and bottlenecks of a program. However, almost always profiling comes with the cost of an overhead to the program which can eventually reduce the accuracy of the profiling. Thus, an efficient profiler's overhead should be small enough to rely on.

Profiling the performance of software is a cross cutting concern, because a profiler application needs to cut across the whole program in order to produce performance data. This paper analyzes the efficiency of using aspect oriented programming (AOP) technique for profiling real time applications. AOP is a programming paradigm which complements object oriented programming (OOP). The main principle of AOP is separating the cross cutting concerns from software's business logic to increase readability, maintainability and modularity of code. For analyzing purposes, an AOP based java profiler (JAAOP) is introduced which uses aspectj for implementing AOP behavior. JAAOP can profile; method execution times, method execution counts, object initialization times, object counts and exceptions.

Profiler overhead results of JAAOP and a number of other known java profilers are presented for profiling Jake2, a java based 3D game engine. Jake2 is simply the java porting of well-known Quake2 game and it is a real time Java application. The conclusion whether profiling a real time Java application, using AOP and aspectj is efficient, reusable and maintainable is presented in end of the study according to profiling results.

GERÇEK ZAMANLI JAVA UYGULAMA PERFORMANSININ İLGİYE YÖNELİK PROGRAMLAMA MODELİ İLE PERFORMANS ANALİZİ

ÖZET

Yazılımlar için performans oldukça önemli bir kalite göstergesidir. Bir yazılımın performansı hakkında bilgi edinmek için göze alınması gereken göstergeler yazılımın amacına göre değişse de genel olarak; bellek tüketimi, çağrı süreleri ve işlemci kullanımıdır. Yazılım kalitesini arttırmak için geliştirme sürecinde performans analizi yapılmalıdır. Bu sayede performans tıkanıklığına neden olan ve çağrı sayılarının fazla olduğu metodlar belirlenebilir. Fakat bir yazılımın performans analizinin yapılmasını sağlayan araçlar ya da yazılımlar, ekstra yüke neden olurlar. Eğer yazılımı gözlemlemenin getirdiği yük çok fazla olursa, elde edilen sonuçlar yazilimin performansı hakkında yeterince doğru bilgi içermez. Bu yüzden yazilim performansını gözlemekte kullanılan araçların ve yazilimlarin minimum ek maliyet gereksinimi olmalıdır.

Performans gözleme işi, performansı gözlenen yazılımın metodları ile kesişen bir konudur. Bunun nedeni de gözleme işi için gerekli kod parçalarının, esas yazılımın yaptığı işten bağımsız olarak her metodda veye sınıfta bulunması gerekir. Bu çalışmada, ilgiye yönelik programlama tekniği kullanarak, Java tabanlı gerçek zamanlı uygulamaların performans gözleminin yapılması ve bu metodla yapılan gözleme işleminin yeterince etkili olup olmadığını incelemektedir. İlgiye yönelik programlamanın amacı birbiri ile kesişen yazilim isteklerini, kod okunabilirliği ve tekrar kullanılabilirliğini arttırabilmek için modüler hale getirebilmektir. Nesneye yönelik programlamayı birbiri ile kesişen konuların modüler hale getirilmesi konusunda tamamlayan bir programlama yöntemidir. Çalışma içerisinde yöntemin etkinliğini gözlemlemek için JAAOP adı verilen bir performans gözleme yazılımı geliştirilmiştir. Yazilimin geliştirilemesinde ilgiye yönelik programlama gerçeklemesi olan aspectj dili ve yazılım ortamı kullanılmıştır. Geliştirilen yazılım ile metodların yürütme zamanları, metodların toplam çağrılma süreleri, nesnelerin yaratılma süreleri, yazlımın yaşamı boyunca oluşturulan toplam nesne sayıları ve oluşan istisnai durumlar (ing. Exception) gözlemlenebilir.

Çalışmanın sonuç kısmında geliştirilen JAAOP yazilimi ile bilinen diğer Java program gözlemleyici yazılımların karşılaştırılması yapılmıştır. Karşılaştırmada test uygulaması olarak, gözlemlemek için, Jake2 adı verilen Java tabanlı 3 boyutlu oyun uygulaması kullanılmıştır. Jake2, tanınmış Quake2 oyun motorunun Java ile yeniden yazılmış halidir. Test uygulamasının Jake2 olarak seçilmesinin sebebi ise, Jake2 nin gerçek zamanlı bir yazılım olmasıdır. Bu çalışmada yapılan değerlendirmelerin sonucunda ilgiye yönelik programlama tekniği ile geliştirilmiş bir gözlemleme yazılımının, gerçek zamanlı Java programlarını gözlemekte kullanılabileceği sonucuna ulaşılmıştır.

1. INTRODUCTION

Performance of software is a very important quality measurement, especially when real time applications are considered. Real time term, in software, generally refers to a software system where there are hard deadlines between the operations. In those systems, software will fail or its behavior will become unpredictable if the operations are not completed within the deadline time interval. However, the real time term used in this study refers to a soft real time system where the missed deadline of an operation will only cause loss of performance in the software system. Examples for given type of applications might be; a high through put web server application, a video decoding program etc.

For a real time software application to be quality, it should be able to operate at a desired performance. Although performance indicators of a software application may vary in a wide range of application specific requirements, the most general performance indicators are as follows;

- Time spent during a method's execution.
- Number of each method invocation.
- Time spent during initializations of each object.
- Number of each objects instance creation.
- Thrown exceptions.

There are also other important performance indicators for software like central processing unit (CPU) allocation, memory context, threads and more. Within this study, indicators that are mentioned above are considered. Because adding more performance indicator data into the performance data set makes will make it harder to obtain accurate performance data.

The job of gathering performance data of a software application, while it is still running, is called profiling. Although software profiling needs to be done while software is still running, it is not necessarily mandatory to analyze the data at the same time. However, being able to profile software and see the results immediately would decrease the time spent for profiling, therefore it is a more preferred method for profiling.

Efficiency of a profiler can be determined by a few factors. First is the usability of the profiler application or application programming interface (API). Next is the set of performance indicators that a profiler can gather. Another factor is the ability of producing and presenting the results of performance analysis at the same time, while the software is running. Finally, the biggest factor determining the efficiency of a profiler is its overhead.

Overhead of a profiler is the extra amount of resources, such as CPU time and memory allocation, used by the profiler during the performance analysis process. The reason it is being a major efficiency factor is that it can directly affect the gathered performance results. Think of a profiler, which needs more resources than the application it profiles. In such a case, gathered performance data will be useless. For the performance indicating data to be precise enough, profiler's overhead must be as small as possible.

Profiling software for its performance at the early stages of software life cycle will ensure its end result performance at a lower cost. In case where poor performance is noticed at a later phase, such as during tests or after releasing, finding the bottlenecks and fixing them will cost higher. Therefore, it is a common approach to use profilers during development phase of software life cycle. There are a number of profilers for java applications to be used during development. The most common used technology among these profilers is java virtual machine profiler interface (JVMPI). JVMPI allows the use of java agents with a callback mechanism to provide event notifications such as, method calls, class loading etc. Since with version 1.5 of java JVMPI is replaced with java virtual machine tool interface (JVMTI), which provides an interface for following the java virtual machine (JVM). Another common approach for profiling is altering the executing java byte code to include additional set of profiling logic in the application to be profiled, instead of watching JVM with interfaces.

In addition, a number of profilers use the aspect oriented profiling (AOP) approach. AOP is a programming paradigm complementing object oriented approach. The main principle behind AOP is the idea of separating the unrelated code areas and modularizing them together, in cases where it is not possible using OOP. These are usually the cases where a programming logic needs to cut across the whole application logic, and thus it cannot be cleanly separated and modularized using regular OOP approach. A very common implementation of AOP in java programming language is aspectj. It is simply an AOP extension to java language.

2. BACKGROUND

2.1 Java

Java is a commonly used programming language released at 1995 by Sun Microsystems. The claim of Java is being a general-purpose class-based object-oriented programming language, designed especially to have fewest possible dependencies for implementation [1]. Once the Java code is developed, it can run on anywhere.

The platform that Java applications run on is java platform. It contains a number of programs that are part of it. Of those programs, main and important ones are java virtual machine (JVM), just in time compiler and java run time environment. Applications created with java language are converted to intermediate byte code after compilation using the compiler in java platform. After becoming byte code, program can run on JVM. JVM, in basic terms, is a virtual processor. It can run any type of byte code compiled from java or any other source. Other supported programming languages can also be compiled to byte code and run in JVM. Writing once and being able to run java code anywhere makes it platform independent. Same java application behaves the same way on any hardware, because the application is run by JVM and JVM is created specifically for each type of hardware, which makes it platform dependent.

For this study java is the target platform technology, whose performance profiling is going to be carried. In addition, the implemented profiler in this application uses java language along with aspectj. For information on aspectj, please see further sections.

2.2 Aspect Oriented Programming

AOP is a programming paradigm which complements object oriented programming. The main principle of aspect oriented programming is separating the unrelated code fragments from the actual business logic in a modular fashion [2].

Separating the unrelated areas from actual business logic increases the readability, maintainability and modularity of code. Unrelated code areas to business logic are often referred as cross cutting concerns in AOP terminology.

Cross cutting concerns are the areas that apply to many places in the business logic and because of the relatedness of those code areas, it is impossible to design these concerns by using regular OOP approach. A good example for a cross cutting concern might be the job of logging an application. The parts in the business logic, which requires to be logged has to add extra code. This extra code is not actually related to the business logic at all, but it can not be implemented in OOP approach, fully modularized.

An aspect is a programming logic, a sequence and it can be applied to desired parts in the business logic. It is a cross cutting concern, which is modular and can be applied based on pre defined rules. Definitions that complete AOP are as follows.

Joinpoint is an execution point in the program's business logic. A joinpoint can be; method execution, object initialization, field references etc... Second important element in AOP definition is pointcut. A pointcut is a set of rules combined of joinpoints, checking if the code has reached certain desired execution point with required parameters. Let a simple object initialization pointcut for class *Foo* be *ClassFooInitializePointcut*. This pointcut is reached twice by initialization of *Foo* class with the following example code part in business logic;

```
Foo bar1 = new Foo();

(new Foo()).doSomething();
```

Pointcut designator used within *ClassFooInitializePointcut* is object initialization of *Foo* class. It will only cross cut initialization of class *Foo* and nothing else.

Pointcuts are also used for method execution. Let *CatchDoSomethingPointcut* be a pointcut, which cuts *Foo.doSomething* method, in that case it is reached once by executing the above code snipped. The pointcut is met in second line. Similar to *ClassFooInitializePointcut*, *CatchDoSomethingPointcut* will also be reached for only execution of *Foo.doSomething* method.

The main logic in AOP is separating cross cutting concerns. The method in AOP for this separation is as follows. Define a pointcut with rules consisting of joinpoints. When the pointcut is matched, execute the separated code part within an advice block. An advice block in AOP is additional code that is executed when advised pointcut is reached. Actually, advice definition includes the time of applying the additional code relative to the pointcut. It can be before, after or around. When the around is applied for an advice it is the extra code's responsibility to do the actual method call. Therefore, it is even possible to skip the actual method calls using around advices. Neither the compiler, loader nor JVM will complain about that and the code will continue running without issuing the method call.

AOP has a number for implementations for Java programming language. This study focuses on performance evaluation of real time java applications, therefore it is wise to have a decision among those which will run faster. AOP implementations considered are as follows;

Aspectwerkz: It is an AOP implementation, which uses method of byte code modification, at run time, for including the aspects into the existing business logic.

Awproxy: This implementation uses dynamic proxy methodology for implementing AOP behavior. A dynamic proxy class implements runtime specified interfaces. This way a method invocation through those interfaces of the class can be encoded and dispatch for some other class via a uniform interface. Dynamic proxy class is used for creating type safe proxy objects with interfaces without the need of compile time generation. A method in dynamic proxy gathers all interface calls and determines the actual called method from an array of Objects, using java reflection methods. [3]. Awproxy uses Awbench with proxies.

Aspectj: Aspectj is simply an AOP extension of Java programming language. It uses aspectj compiler to generate java byte code from aspects, then aspect byte codes are woven into actual program's logic. As of current version of aspectj, weaving can be done during compile time and load time. Although using aspectj seems to require another set of compile and load process, these operations can be handled by many of the integrated development environments.

Spring AOP: It is a sub component of spring framework, which is an open source application framework for java.

Spring AOP uses dynamic proxies for implementing the AOP behavior. It is implemented in core java and does not require additional compiler or load time operation. Although it can be used separately, the implementation is focused on usage of AOP within spring framework instead of providing a fully capable AOP implementation [4].

JBoss AOP: JBoss is a well known open source Java application server. JBoss AOP is an AOP implementation provided by JBoss. It supports usage of AOP with or without the application server and it uses pure java and extensible markup language (XML) for supporting AOP behavior [5].

Dynaop: Dynaop is an AOP framework, which is based on proxies. The main idea behind Dynaop is providing a simple API for java developers to implement AOP [6].

Cglib: Code generation library (CGLIB) is an open source java library for code generation, as its name indicates. The use of CGLIB is by extending the java classes and implementing the interfaces at run time [7].

AOP Alliance: AOP alliance is a project, which aims to standardize the usage of AOP for java. It has its own implementation consisting of interfaces but the main claim of AOP alliance is creating standards for AOP implementations [8].

Table 2.1: presents the benchmarking results of work at [9], which includes a number of current AOP implementations test results for their operation times. The testing environment used for the benchmark is AWBench, an AOP performance test tool.

In the table, first two rows are the most important results. An AOP program that cuts across a java application, will most likely use a before or around advice in order to be able to execute additional code. Also access to run time information is needed in the advice part for interaction. Looking at the first two column results, aspectj is clearly the fastest AOP implementation.

Aspectj's properties as an AOP implementation are not limited to being the fastest one but it also complements java programming language very well, which is because aspectj uses a similar syntax to java. As a result it is very easy for a Java programmer to follow.

Table 2.1: AOP implementation's benchmarking results

AWBench ns/invoke	Aspectwerkz	Awproxy	**AspectJ**	Spring	JBoss	Dynaop	Cglib	AOP Alliance
before, args() target()	10	25	**10**	355	220	390	145	-
around x 2, args() target()	80	85	**50**	436	290	455	155	465
before	15	20	**15**	275	145	320	70	-
before, static info access	30	30	**25**	275	175	330	70	-
before, rtti info access	50	55	**50**	275	175	335	75	-
after returning	10	20	**10**	285	135	315	85	-
after throwing	3540	3870	**3009**	-	5032	6709	8127	-
before + after	20	30	**20**	445	160	345	80	-
before, args() primitives	10	20	**10**	350	195	375	145	-
before, args() objects	5	25	**10**	325	185	345	115	-
around	60	95	**10**	225	-	315	75	-
around, rtti info access	70	70	**50**	250	140	340	80	70
around, static info access	80	90	**25**	245	135	330	75	80

AspectJ is also one of the first implementations of AOP. Therefore, it is more stable and has better support. For all these reasons and most importantly for being the fastest implementation, aspectj is chosen in this study for profiling real time java applications.

2.3 Aspectj

Aspectj is the chosen AOP implementation for the purposes of this study. It basically is an aspect oriented extension to Java [10]. It actually is the outcome of the effort of producing a special purpose aspect oriented language before moving to a general purpose Java model, by Xerox Palo Alto Research Center (PARC).

An aspect in aspectj is a cross cutting concern which can be defined in a modular way. Aspects can be applied dynamically by a defined set of rules and they provide a mechanism to apply additional code for the aspect [11].

As the time of this study, aspectJ 5 supports the annotation based usage along with the regular code style of aspects. Since annotation based usage is more similar to regular Java programming language, it is chosen for this study. It should be noted that code style usage of aspectj is also similar to java language but annotation based usage is more readable.

A simple aspect declaration is as follows;

```
@Aspect
public class Foo {
    // Aspect code here
}
```

Similar to simple aspect declaration, an aspect instantiation model which applies only to elements within *abc* package can be declared as follows:

```
@Aspect("perthis(execution(* abc..*(..)))")
public class Foo {
     // Aspect code here
}
```

Joinpoints are the points in the application that could trigger an advice. In addition, the available joinpoints may different between implementations, in aspectj below are available to define points in code.

- Method call
- Method execution
- Constructor invoke
- Constructor execution
- Aspect advice execution
- Object initialization
- Static initialize execution
- Referencing a class field
- Assigning a class field
- Handler execution

Pointcuts are the mechanism in aspectj and in AOP, which enable declaring interest in a join point for executing an advice. The decision logic on executing the advice, when a joinpoint matched, is encapsulated with pointcuts. A sample poincut declaration is as follows.

```
@Pointcut("call(* org.foo.Bar.*(..))")
void listOperation() { }
```

Above code snippet for a pointcut will be matched by any method call in *org.foo.Bar* class with any return type and any number of parameters of any type.

Advice is the piece of code, which is executed when an aspect is invoked. It contains its set of rules on when to be included according to the join point that was triggered.

Pointcuts define the set of rules consisting of join points, that an advice is applied. In addition, when to execute the advice according to its pointcut has to be defined. Possible definitions of advice execution are as follows.

- Before
- After
- Around
- After Returning
- After Throwing

A simple advice example for applying the previous *listOperation* pointcut is as follows.

```
@Before("listOperation(thisJPSP )")
public void beforeListOperation(JoinPoint.StaticPart thisJPSP) {
     //Adviced code here
}
```

The same advice could be defined without specifically defining a point cut forehand. This way the point cut of an advice has to be declared within the advice. Following code snippet is the same as previous one, the only difference is that it does not require the *listOperation* pointcut.

```
@Before("call(* org.foo.Bar.*(..))")
public void beforeListOperation(JoinPoint.StaticPart thisJPSP) {
   //Adviced code here
}
```

The only difference with using an existing pointcut definition against defining it in the advice is that, in the case of defining the point cut expression in the advice, it can not be reused. Lets say a new advice is needed with the same *listOperation* pointcut, which happens to be applied after the method call. If the point cut is defined separately it would be easier to create the new advice, and duplicate coding will be prevented.

To use the aspectj implementation, creating the aspects is not enough alone. They need to be compiled into java byte code. To accomplish this task aspectj introduces the aspectj compiler. Aspectj compiler compiles the aspect codes into java byte code so that they can be executed by JVM. After creating the java byte codes from aspects next step is weaving these code with the actual java code. There are currently a few possible methods for weaving aspect code into the application.

First weaving method is compile time weaving. As the name indicates aspectj compiler works with java compiler during the process of compiling of both aspects and java code. Together aspectj compiler and java compiler weave the compiled aspectj code into compiled java code and create a final java byte code, which includes the aspect behavior within.

Second method for weaving the aspects into the java code is load time weaving. As the name indicates, in load time weaving aspects are compiled to java byte code and they are woven into existing java byte code at the time when java class loader is loading the java classes. For this method a special class loader provided by aspectj is used. This special class loader needs to know which java classes to load originally and additionally aspect classes to weave during loading operation needs to be passed to the class loader.

Final method for weaving aspects into a java application is run time weaving. Run time weaving is weaving aspect byte code into the java byte code, while it already loaded into JVM and running. Unfortunately, AspectJ does not support this option directly, but this type of weaving can be achieved by using ApsectJ's load time weaving along with the help of additional class recycling tools.

2.4 Profiling

Profiling an application is the job of measuring a program execution for certain parameters [12]. These parameters may change depending on the required performance indicators of the application. For instance, in an embedded system profiling may require the measurement results of memory usage, because it has very limited amount of memory. On the other hand, in a very powerful web server profiling might be more focused in response times of its methods instead of memory usage.

This study is interested in real time Java applications. Real time term, in software, usually refers to a software system where there are hard deadlines between the operations. In those systems, software will fail or its behavior will become unpredictable if the operations are not completed within the given time interval. However, the real time term in this study refers to soft real time systems where the missed deadline of an operation will only cause loss of performance in the software system. An example application might be a video decoding program. It is a real time application but when the performance of decoder drops down for some reason, it will not crush but will only drop some frames. Another example for this type of applications can be 3D games, where loss of performance drops down the generated frames per second but it will not fail the game.

For a real time software application to be quality, it should be able to operate at a desired performance. Performance indicators used in this work are as follows.

- Time spent during a method's execution.
- Number of each method invocation.
- Time spent during initializations of each object.
- Number of each objects instance creation.
- Thrown exceptions.

To assure the performance of software, the best method would be measuring it during the development, because any anomalies found in the program such as; low performance at certain conditions, bugs causing performance falls, etc. will be more expensive to fix in the later phases of software life cycle. Work at [13] indicates that fixing a bug in software during testing costs around 2 times more expensive than of developing. Same study also results that fixing the same bug will cost around 10 times more after testing. That is why this study considers the use of profiler at early stages of development.

There are currently some profilers for profiling java applications. Each profiler has a different technology for carrying the profiling job. Since a profiler is presented within the work in this study, it needs to be compared against some other profilers to be able to decide on the efficiency of it. For that reason below well known java profilers are considered for comparison.

JRAT: Java run time performance analysis toolkit (JRAT) is an open source profiler. Its claim is being a lightweight application for profiling the performance of Java applications [14]. JRAT works by adding hooks into the java code. In order to do that JRAT adds its agent into the code using JVMTI. In order to start profiling an application with JRAT its agent code must be passed as a parameter to the Java application while invoking the JVM. After that JRAT code will be added into the code that will be run with byte code manipulation tools. However this approach seems similar to AOP, JRAT does not use any AOP implementation. Once JRAT start profiling, it generates a log file. To be able to see the results of profiling, the running application must end. There is no way to see the results while the program is still running. Some of the data, collected during performance analysis by JRAT are; method execution time, overall time rate of method, exception information.

Netbeans Profiler: Netbeans is a well-known integrated development environment (IDE) for Java, created by Sun Microsystems. Netbeans profiler is an additional module included in Netbeans IDE. It is an integration of profiling technology, and it uses byte code manipulation at run time for adding the profiler logic into the running code [15]. Netbeans profiler is a complete program analyzer besides of being a profiler it has many capabilities. Not all, but an important portion of those are as follows.

- Memory Profiling
- CPU Profiling
- Thread Profiling
- Heap Walking
- Following Garbage Collection
- Run Time analysis of applications
- Profiling an already running application
- Profiling a remote application as well as a local one

JProfiler: JProfiler is a commercially available Java application profiler. It uses JVMTI technology for profiling the Java applications. Can be used as a stand alone product or as a plug-in to Eclipse IDE. Some of its features are as follows.

- Easy to use
- Displays results real time
- Heap walker
- CPU profiler
- Thread profiler
- Java enterprise edition (JEE) support, which allows to separate call tree of a JEE component.
    - Java Messaging Services (JMS)
    - Java Naming and Directory Interface (JNDI)
    - Java Data Base Connectivity (JDBC)
- Extensible, it can be used to create your own profiler.

Jprofiler provides the best set feature among these three profilers. However, Netbeans profiler is also a direct competitor of JProfiler with its similar set of features. Netbeans profiler lacks with only a few minor capabilities that JProfiler can do. Finally, JRAT has the lowest set of features as a profiler among these three profilers. The most obvious lacking feature is that, it does not allow run time analysis of the Java application. Application needs to finish running for analyzing with JRAT. But it might seem fair for a profiler whose claim is to be light weight as possible.

3. ANALYSIS

3.1 Definition

This study focuses on profiling real time Java applications with AOP technologies, namely Aspectj. Therefore, it includes a profiler implementation which uses AspectJ technology for cross cutting concerns. The problem for this study to analyze is the need of a software application for profiling real time java applications. This software will be used for determining whether AspectJ can be used as a technology for profiling real time Java applications, so it has to follow attributes such as having low overhead, must be able to profile method executions and so on.

3.2 Requirements

Requirements for the profiler that is created as part of this study are explained in the following sections

3.2.1 Requirement 1 :AOP technology

As the study discusses the usage of AOP for profiling real time Java applications, the profiler created should make good use of AOP technology. Particular AOP implementation to be used is AspectJ, because among the other implementations compared in Table 2.1: AspectJ is the fastest. Another reasons for using AspectJ is that it can be used with a syntax very similar to Java and it is a very common implementation.

3.2.2 Requirement 2 :Usability

The profiler is intended to be used during development phase of software life cycle. It should be easy to use in order to not bring additional time cost for developers.

3.2.3 Requirement 3 :Extensible

The profiler should be extensible such that it should be possible to create a new profiler based on this one.

Besides of being possible it should also be easy enough to create a new profiler from the one created in this study instead of creating one from scratch.

3.2.4 Requirement 4 :Run time analysis

The profiler should be able to do run time analysis of a real time Java application. This does not necessarily mean adding the profiler to an already running Java application, but it means that profiling results should be able to review while the profiling itself is still in progress. Also the results seen should be updated periodically.

3.2.5 Requirement 5 :Remote analysis

The profiler should be able to allow performance analysis of a running application, form a remote location. This way the viewing process overhead will not be added to the same machine that runs the application. Thinking that the running application will be a real time Java application it makes sense to reduce the overall overhead on the machine caused by profiling.

3.2.6 Requirement 6 :Exporting results

The profiler should be able to export the analyzed results at run time in a certain format at disk. So that the results that are stored on the disk can be compared later.

3.2.7 Requirement 7 :Offline analysis

The profiler should be able to store analysis results even when the profiling session is not reviewed at all. This way there is no need to explicitly export the results of a profiling session, instead each session will store the produced results on disk for viewing later.

3.2.8 Requirement 8 :Importing results

The profiler should be able to restore the results of an earlier profiler session, either from exported file or the automatically saved results. This way earlier profiling sessions execution reports can be analyzed at a later time.

3.2.9 Requirement 9 :Method execution time

The profiler should be able to gather the information of how long does a method execution take.

3.2.10 Requirement 10 :Method call count

The profiler should be able to gather the information of how many times a method is called.

3.2.11 Requirement 11 : Object count

The profiler should be able to gather the information of how many instances of an object is created in total execution time of the profiled application.

3.2.12 Requirement 12 :Object initialization time

The profiler should be able to gather the information of how long does an object initialization take.

3.2.13 Requirement 13 :Exceptions

The profiler should be able to get the information of thrown exceptions and their exact locations in the applications.

3.2.14 Requirement 14 :Partial profiling

The profiler should be able to allow profiling of selected certain methods and classes instead of profiling the whole application.

3.2.15 Requirement 15: Profiling overhead

The profiler should have the least possible overhead that can be achieved with the use of AspectJ.

3.3 Use Cases

The uses cases for the usage of the profiler introduced within this study are in the following sections.

3.3.1 Use Case 1: Run time profiling for method execution

Actors:

Application: The Java application which will be profiled.

Developer: The person developing a Java application and will profile it.

Profiler: The profiler application.

Preconditions:

Developer has the knowledge of which methods to profiler (these are most likely the ones being developed or updated).

Post conditions:

Developer can use profiler to gather the method execution time and method call count of each method defined.

Basic Flow:

1. Developer marks the methods which needs profiling.

2. Developer starts the profiler with the application

3. Profiler shows the method execution times and method call counts to the developer while the application is running.

4. The results shown by profiler are updated accordingly.

3.3.2 Use Case 2: Run time for object initialization

Actors:

Application: The Java application which will be profiled.

Developer: The person developing a Java application and will profile it.

Profiler: The profiler application.

Preconditions:

Developer has the knowledge of which classes to profiler (these are most likely the ones being developed or updated).

Post conditions:

Developer can use profiler to gather the object initialization time and total created object count of each class defined.

Basic Flow

1. Developer marks the classes which needs profiling.

2. Developer starts the profiler with the application

3. Profiler shows the object initialization times and object counts to the developer while the application is running.

4. The results shown by profiler are updated accordingly.

3.3.3 Use Case 3: Run time for exceptions

Actors:

Application: The Java application which will be profiled.

Developer: The person developing a Java application and will profile it.

Profiler: The profiler application.

Preconditions:

None

Post conditions:

Developer can use profiler to gather the information of thrown exceptions.

Basic Flow:

1. Developer starts the profiler with the application

2. Profiler shows the thrown exceptions to the developer with their source location information.

3. If any of the exceptions are thrown more than once from the same source location, their total number is shown by profiler.

4. The results shown by profiler are updated accordingly.

3.3.4 Use Case 4: Offline profiling a java application

Actors:

Application: The Java application which will be profiled.

Developer: The person developing a Java application and will profile it.

Profiler: The profiler application.

Preconditions:

Developer has started a profiling session.

Post conditions:

During profiling session, results are saved by profiler for later examination

Basic Flow:

1. Profiler creates performance data for the started profiling session.

2. Profiler saves the performance data of the current session periodically.

3. The developer finishes the profiling session.

4. The results shown by profiler are updated accordingly.

Extensions:

3a. Profiler process is killed

4a. Latest periodic save results remain in hard disk.

3.3.5 Use Case 5: Exporting profile results

Actors:

Application: The Java application which will be profiled.

Developer: The person developing a Java application and will profile it.

Profiler: The profiler application.

Preconditions:

Developer has started a profiling session.

Post conditions:

Developer can save the profiling results for examining later.

Basic Flow:

1. Profiler creates performance data for the started profiling session.

2. Developer chooses to export performance data.

3. Profiler saves the current results to a desired location of developer.

4. Profiling session continues.

3.3.6 Use Case 6: Importing profile results

Actors:

Application: The Java application which will be profiled.

Developer: The person developing a Java application and will profile it.

Profiler: The profiler application.

Preconditions:

Developer has started a profiling session and exported the results.

Alternatively, there exists a profiling session file automatically stored by profiler.

Post conditions:

Developer can examine the earlier saved or exported performance data

Basic Flow:

1. Developer imports the earlier profiling session export data from profiler.

2. Profiler shows the saved results

Extensions:

1a. Profiler had saved a session's performance data automatically. Developer imports this session.

3.3.7 Use Case 7: Remote profiling

Actors:

Application: The Java application which will be profiled.

Developer: The person developing a Java application and will profile it.

Profiler: The profiler application.

Preconditions:

Machine A has the application.

Machine B has the profiler.

Machine A and machine B can connect through network.

Post conditions:

An application running on machine A can be profiled remotely by profiler on machine B

Basic Flow:

1. Developer starts the application from machine A.

2. Developer stars a profiling session from machine B.

3. Developer can configure profiler to profile the running application on machine B.

4. Profiler can show the performance data while the application on machine A is running.

5. Profiler updates the performance data shown periodically.

5. Developer can use profiler on machine B to save the performance data of application running on machine A.

3.4 Solution description

In this section the solution produced for the given requirements and use cases will be detailed. First of all the solution needs to include at least two different parties of applications which will together be the profiler application. The reason is that in requirement 8 and in use case 7 remote profiling is mentioned. The task of profiling an application from a remote location, without having a separate part cannot be accomplished. The solution breaks the profiler into two separate parts as "*core*" and "*viewer*".

3.4.1 Core

This core part of the profiler is the part which does the job of gathering the performance data.

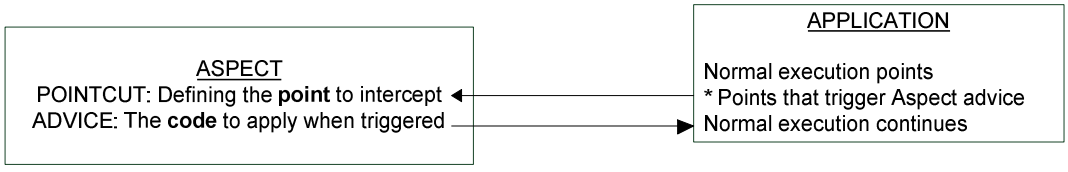

Figure 3.1 : Aspect cross cuts the application at certain point.

It uses the AspectJ to utilize the cross cutting profiling concern. Certain aspects are created for intercepting the required portions of the application. A sample aspect behavior will be as in above figure

The aspects that are used in the core part are as follows.

Method Aspect: The aspect which cross cuts the desired method execution points in the profiled application. The job of the advice code within this aspect will be to collect the method execution information of the intercepted method.

Class Aspect: The aspect which cuts across the object initialization of desired classes. The job of the advice in this aspect is to calculate and store the total object count.

Constructor Aspect: This aspect will cut across desired class's constructor calls and its advice code will collect the constructed object's initialization time information.

Exception Aspect: This aspect will intercept the points in the application which throws exceptions. Its advice code will get the exception information along with the source location of the exception.

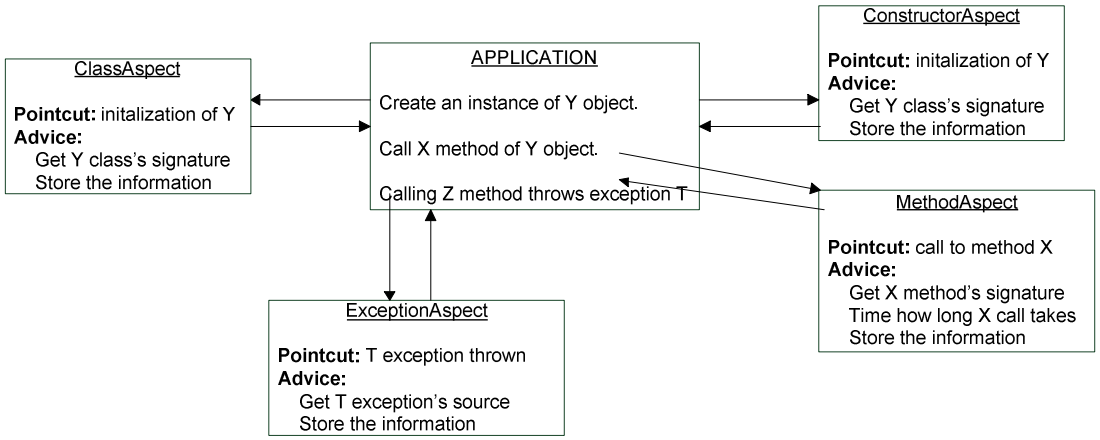

Figure 3.2 : Aspects in core of the profiler.

The aspects defined in Figure 3.2 :, cut across necessary method calls, object constructors and exception throwing statements in the profiled application. While these aspect's defined pointcuts are matched their advised code will be invoked. For being able to gather performance data, each aspect's advice code stored the information gathered about the execution by sending this information to *Controller* element.

Controller element has more than one task in this solution. Its major task is store the information sent from aspects into the *DataMap*. *DataMap* is basically the store mechanism, which stores all the profiling information gathered. *Controller* is the single element who can store data into *DataMap* and gather data from *DataMap*. By limiting the access to *DataMap* and allowing only the *Controller* element in charge, controlled access to the *DataMap* is achieved.

The storing of the performance information of the application is done by *Logger* element. *Logger* element gathers the stored data from *Controller* and stores that information to the disk. This operation is done periodically and by a seperate thread not to intercept profiling process.

Final element of the profiler is the *Server*. *Server* is the element which can be connected over the network to get gathered profiler information. It gets the profiler data from *Controller*, because *Controller* is the only element that can reach the profiling data.

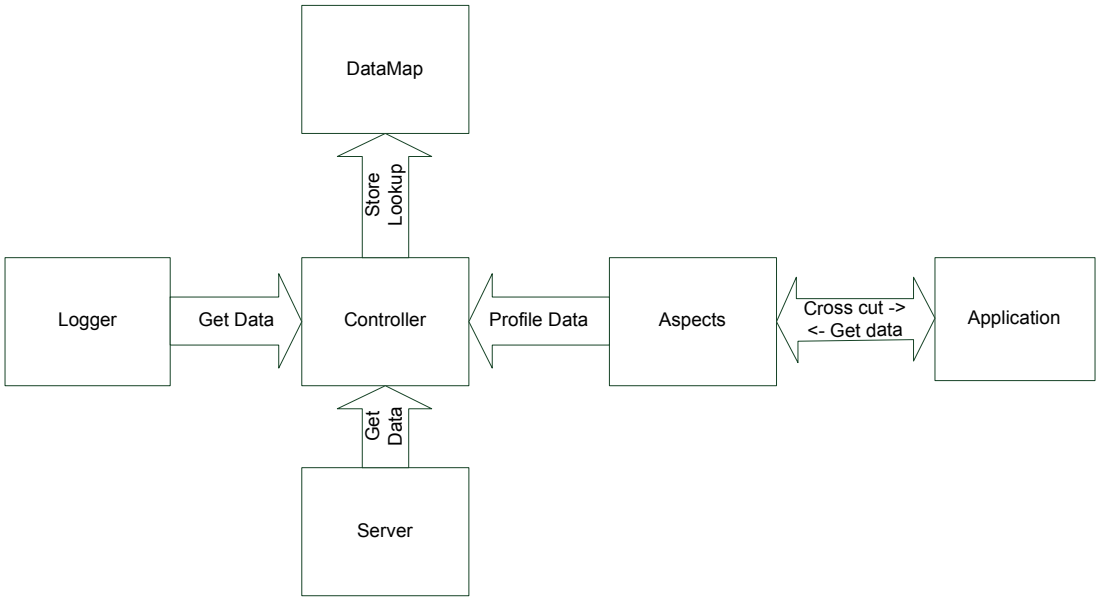

Figure 3.3 : Elements of the core of the profiler

3.4.2 Viewer

Viewer is the other part of the profiler. The main focus of the *viewer* is the representation of the profiled data. *Viewer* gathers data from the core part of the profiler either remotely or by accessing the data files saved.

These data files can be the files that are saved by profiler application's *Logger* element or it can be stored by the viewer application itself.

Viewer is a separate application than core part of the profiler. It runs as a separate process. It can run on the same machine that the profiling is being done or preferably it can be run on a remote machine. Running remotely will distrupt less the local resources used, while running the application to be profiled and core part of the profiler along with it.

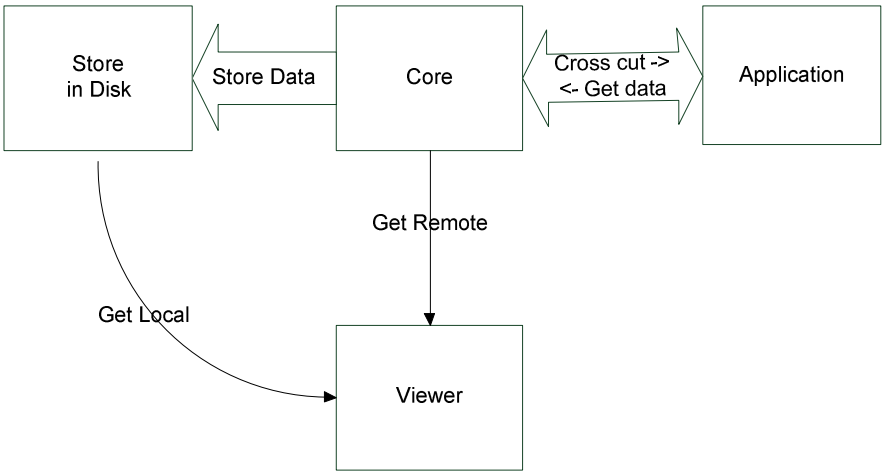

Figure 3.4 : Profiler Core and Viewer Interaction

The core part of the profiler stores the raw data. It does not do any sort of calculation on the stored data. That is in order to reduce the overhead of the profiler's core part. That is why all the sorting and the calculation on the raw information is done in the viewer. After all viewer is the part that will represent the data.

Controller in the viewer is the most essential part of the viewer, similar to the core part. It is the only element who can interact with *DataMap*. Viewer contains a *DataMap* similar to core. It contains the performance information of the running application, collected by *core*.

As said earlier in the section, *DataMap* is shaped within the *viewer*. The element, which carries this job, is *Parser* and is controlled by controller. The request of the *Controller* are sent to *Parser* and then *Parser* carries necessary operations on the performance data and makes it ready for viewing properly. This usually means sorting the information on different required criterions.

An example is sorting the all class count data so that the first element becomes the class who is initialized most during the execution. Notice that Parser can not reach the *DataMap* information, it should be invoked by *Controller* only.

Remote Interface is the element in the viewer, which interacts with the *core* of profiler in a remote location. To achieve this job it needs to be configured by *Controller* first. After configuration of the remote location is applied to *Remote Interface,* it starts getting the information periodically. When new data is gathered, it will invoke methods in *Controller* so that *Controller* updates the *DataMap*. Again this job is carried over *Controller*.

Of course, there are visual elements within the viewer. The main visual element is the *GUI* part of the viewer. It contains the screens, which enables user interaction. But when the user interacts with *GUI* and this interaction generates an action, this action is first passed to the middleware element *Screen Controls*. Then *Screen Controls* element properly executes these actions by making method calls to *Controller*. *Controller* applies the requests received *by Screen Controls* and replies with correct information. Then Screen Controller updates necessary parts of the visual element *GUI*.

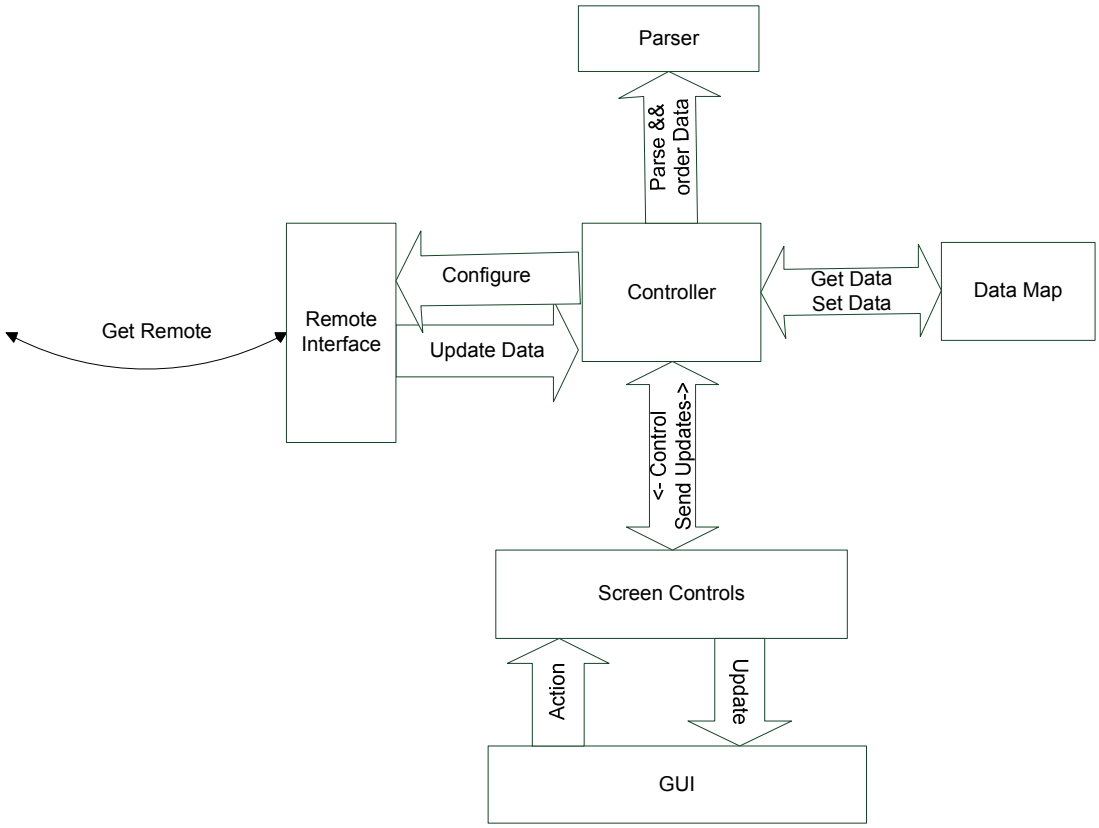

Figure 3.5 : Profiler Core and Viewer Interaction

4. DESIGN ALTERNATIVES

Before going into design details of the profiler, this section discusses some of the alternatives for the parts of the profiler introduced in this study.

4.1 Data Representation

Analysis section shows that created aspects intercept the necessary portions of the application and gather the performance indicating data. This data is represented in *DataMap* elements in both core and viewer part of the profiler. Storing type of this data is very important, especially for *core*. The reason for its importance is that data in it will be accessed constantly during profiling task by *Controller*.

The data needs to be stored will contain the following information.

- Signature of the method, class, constructor or exception.
- The number of times that this method/exception/class constructor is called or created.
- Execution time.

First thing to do is to create a data type for this kind of representation. Let's assume that we have a data type called Data which represents below information along with all the field getter and setters.

```
public class Data {
     private int count;
     private long execTime;
     private String signature;
     /* Getters and Setters here*/
}
```

Then an array of type Data can be created to store the profiler information. In this case the problem is that it is not certain how many elements will be in the array before the execution starts.

One way for passing this situation is getting all method's and class within the profiled application, before it starts running. Java's reflection API can be used to get all the methods and classes that are loaded, and with that on hand an array which is big enough to store all the method's and class could be created and used. Passing the size problem, another problem arises with the use of arrays.

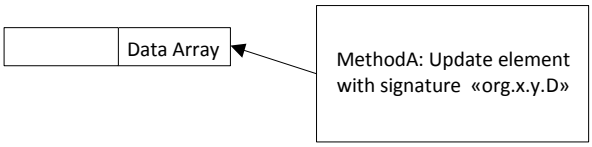

Figure 4.1 : MethodA being called lots of times from aspects

Figure 4.1 :shows an example method, which is being invoked within an aspect. Since this study focuses on real time applications, MethodA could be called many times. For instance, some of the methods in Jake2 applications are called over a million times within a few minutes. Therefore searching the whole array on each method call will bring a huge overhead, because on each method call except the first one, whole array will be searched for a Data type with a matching signature.

Another approach might be not storing the data in a meaningful way when the aspect has intercepted the method or class initialization. Instead of that, just write down that information to a list and let another process take care of the data representation. Since adding, an element to the list is a low cost operation this might be a better solution.

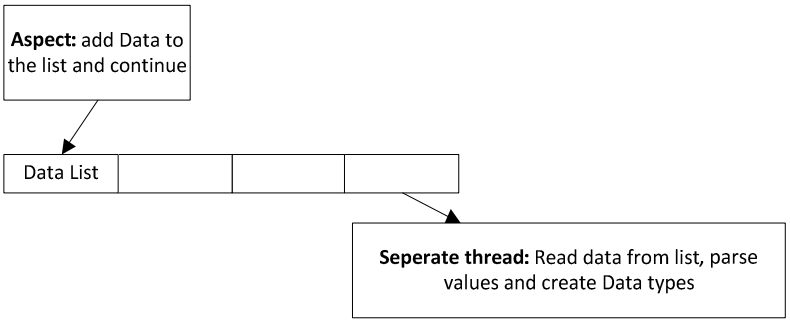

Figure 4.2 : Writing to a list

Although using a list seems like a better solution for data representation, the real time behavior is preventing the use of that approach.

It is true that an application will have a limited number of methods and classes, but with this method in each call, a new entry will be added into the list. Since the real time application has too many calls the list quickly fills up and application fails with an out of memory message.

The data produced by aspects are actually limited by the number of total methods in a program, but the data is being updated constantly on each call. Therefore, it is logical to use a key-value paired HashMap structure. This way the growing size problem will not happen. In addition, it is true that HashMap look up is faster than searching a certain object in an array. To determine the Data objects distinctively keys used are selected as Strings of their respective signatures, which is unique to each method. It should be noted that signature store in Data is long type, which means it includes arguments and returns types as well.

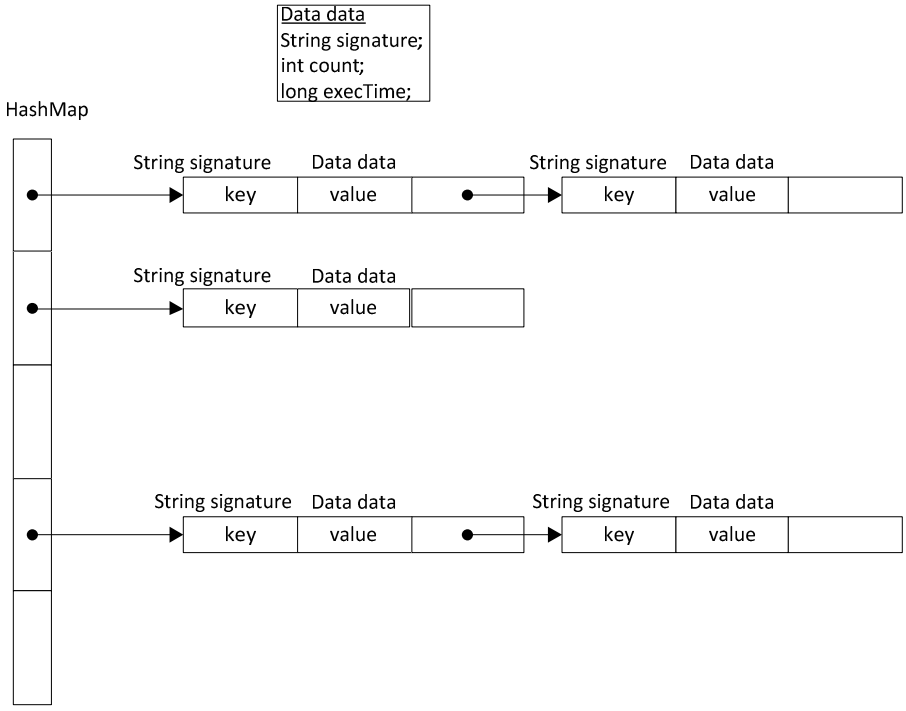

Figure 4.3 : Profiler performance data representation as hashmap.

4.2 Data Processing

After deciding that HashMap is the fastest way for lookup and update the profiler data, how to process it, during the execution of the profiled application, has to be determined.

The sample pseudo code for the critical area in the aspect advice, code area that updates the performance data, is as follows.

```
Original Method Call                        Aspect Code
String a = this.sampleMethod()
                    --> Cross cut-->
                                            startTimer()
                                            Call sampleMethod()
                                            stopTimer()
                                            getSignature()
                                            dataMap.update(time,
                                                      signature)
                                            return(result)
                    <-- Return from Aspect
Execution continues
```

When the execution of a regular application method is cross cut by an aspect, method call will not return until aspect code execution finishes. As it can be seen from the pseudo code snippet, updating the data is one of the time consuming areas in the aspect code. Therefore minimizing the time spent during update of the performance data will decrease the extra time spent for profiling operation.

First option is using a number of separate threads to do the update operation. This way aspect code will be able to return from its execution, but separate threads will complete the data update operation. This is a well-known producer consumer problem. In this case, there is a single producer, which is the aspect code and multiple consumers, the threads that update the data map.

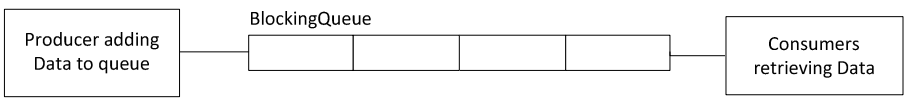

Figure 4.4 : Using threads to update data.

Figure 4.4 :shows the usage of a BlockingQueue, where aspects are the producer to push the Data information and continue their execution without worrying the update of the HashMap, on the other side of the queue separate threads are getting the Data objects and updating their values in the HashMap without disturbing the applications execution.

To examine the most effective number of threads and the size of the queue, a number of tests are done with the Jake2 application. Test are carried while limited number of methods in the application were being cross cut by aspects and each method call's results needed to be updated on the HashMap.

Table 4.1 : Using threads separately for updating the hashmap

Profiled Method Number	Started thread number	Maximum thread number	Queue size	Relative speed
5	0	0	0	1,00
5	5	10	100000	0,89
5	10	20	100000	0,91
5	50	100	1000	0,45
5	50	100	100000	0,81
25	0	0	0	1,00
25	5	5	MAX	0,71
25	10	20	100000	0,64
25	10	20	MAX	0,75
25	15	15	MAX	0,74
25	20	20	MAX	0,72

Result in Table 4.1 shows that the idea of using threads separately for the time consuming update operation is not a feasible approach. Although the numbers of profiled methods seem to be low, they are chosen as to be the most called 5 and most called 25 methods, during the execution of Jake2. This ensures that the results are acceptable for real time applications. What happens during threads and queue usage is that queue quickly fills up because of the huge number of total method calls in a short time; therefore, consumers cannot catch up with the producer and filled up queue makes the real application to wait for the finish of the consumers.

In the end this approach turns into a situation where the entire job is being by a single thread, but additionally the thread cost and an additional blocking queue cost is added. That explains why the single thread usage has the best results and why the increased thread number usually decreases the performance.

After this analysis the decision is to use HashMap directly in the aspects, because there are a huge number of method calls in a short time, using the alternative thread approach is not feasible.

4.3 Sampling

Sampling is selecting a random subset of individual results to obtain a general knowledge. In the case of profiling, sampling implementation would be profiling a certain portion of each operation to generate the performance results. For instance let's assume methodA is called 1000 times during the execution of a program. To gather performance data about methodA, one could only sample 100 of the method calls, calculate their execution times to generate the average execution time information. Since the total execution number of methodA is in interest profiler needs to count all executions. However, by skipping the starting and stopping the timer %90 of the time a performance increase could be gained.

Although sampling seems like a good idea for decreasing the overhead of the profiler, there are certain limitations to it. The major limitation is that starting and stopping a timer has a trivial overhead according to other operations. Gain from sampling would be very small. Another limitation is that sampling requires additional mechanism to decide on whether to time a certain execution. This mechanism will need a way to remember which execution number is the current one or which execution should be sampled. The method for sampling will most likely require a type of data representation and a data map look up eventually. As it is discussed in Section 4.2 data lookup is an expensive operations. The trial of using sampling while profiling Jake2 application showed that it does not decrease the overhead buy increase instead, which support the idea that gain of sampling is much smaller than its cost. Therefore, sampling method is not feasible to be used while profiling a real time application.

5. IMPLEMENTATION

This section includes the design details of the profiler. The design details consist of the Aspects introduced in the profiler along with the core and viewer part of the profiler.

5.1 Aspects

There are four aspects introduced within the design, these are;

Method Aspect: The aspect which cross cuts the desired method execution points in the profiled application. The job of the advice code within this aspect will be to collect the method execution information of the intercepted method.

Class Aspect: The aspect which cuts across the object initialization of desired classes. The job of the advice in this aspect is to calculate and store the total object count.

Constructor Aspect: This aspect will cut across desired class's constructor calls and its advice code will collect the constructed object's initialization time information.

Exception Aspect: This aspect will intercept the points in the application which throws exceptions. Its advice code will get the exception information along with the source location of the exception.

There is a need for a mechanism to provide information on which methods classes are needed to be considered by above aspects. To enable this annotation mechanism is chosen within this study. With the annotation mechanism each method, constructor or class can be marked for profiling operation. This way all the methods, constructors, classes are not going to be cut across by aspects and the overhead of the profiler will be kept at minimum.

There are three annotations represented for the profiler.

TraceCount: An annotation of type TraceCount can be only used for class

definitions. It indicates that the annotated class is going to be profiled for its number of initializations during the execution.

TraceConstructor: An annotation of type TraceConstructor can only be used for constructor definitions.

It indicates that the annotated constructor is going to be profiler for its number of calls and total number of executions.

TraceMethod: An annotation of type TraceMethod can only be used for method definitions. It indicates that the annotated method is going to be profiler for its number of calls and total number of executions.

Sample usage of each annotation in the profiled application would be as follows.

```
@TraceCount
public class render{
  init();
}

@TraceMethod
public void uScreen (xccmdt callback) {
   call.execute();
}

@TraceConstructor
public m_t {
   this.c_t c = new c_t();
}
```

AspectJ allows the usage of annotations as joinpoints; therefore it is possible to create pointcuts which will only be matched while the target code has the specific annotation.

The advised code in the aspect gathers the necessary information and puts that information into the maps. But the aspects need to interact with the *Controller* element in the core of the profiler in order to do so, because *Controller* is the only element which has access to the profiler data.

TraceCount, TraceConstrcutor and TraceMethod classes shown in the diagram are the representations of the respective annotations. They are used by pointcuts in the aspects in order to determine which points to cross cut. Pointcuts declared in each aspect are as follows;

```
@Aspect
public class myAspect{

   @Pointcut("call(@TraceMethod * *(..))")
   public static void myExamplePointCut(JoinPoint.StaticPart
                                                thisJPSP ) { }

   @Before("methodCallPointCut(thisJPSP )")
   public void beforeMethodCall(JoinPoint.StaticPart
                                                    thisJPSP){
      /*! Adviced code here !*/
   }
}
```

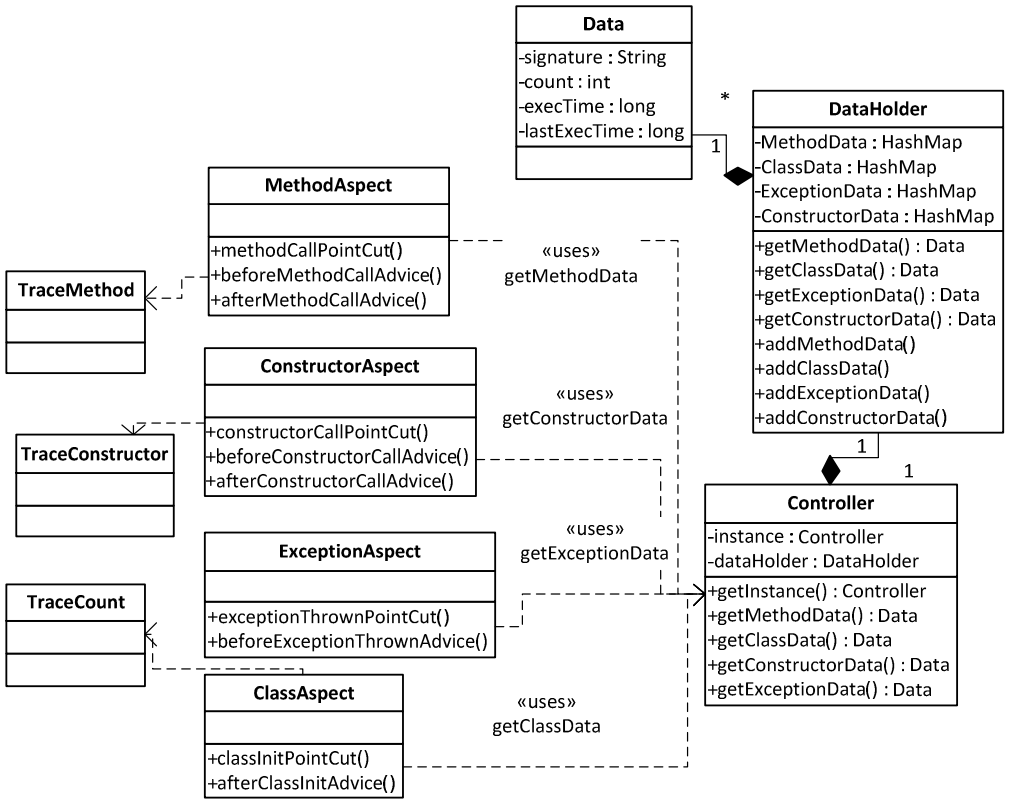

Figure 5.1 : UML diagram of aspects and controller

methodCallPointCut: This pointcut is owned by MethodAspect class and it cross cuts the method calls with TraceMethod annotation.

constructorCallPointCut: This pointcut is owned by ConstructorAspect class and it cross cuts the constructor calls with TraceConstructor annotation.

methodCallPointCut: This pointcut is owned by MethodAspect class and it cross cuts the method calls with TraceMethod annotation.

exceptionThrownPointCut: This pointcut is owned by ExceptionAspect class and it cross cuts the code areas that throws exceptions.

These point cuts determine where to cross cut in the target application, but the actual code is run in the advices which are triggered based on below point cuts. The advices in the aspects are as follows;

beforeMethodCallAdvice: This advice is triggered based on methodCallPointcut, before the execution of the method. Current time value is gathered and saved into the Data object gathered from Controller according to target method's signature.

afterMethıodCallAdvice: This advice is triggered based on methodCallPointCut after the execution of the method. Current time value is gathered and the previous time value saved earlier is subtracted from it to generate the total time spend. Method Data's count is updated with the current information.

beforeConstructorCallAdvice: This advice is triggered based on constructorCallPointcut, before the execution of the constructor. Current time value is gathered and saved into the Data object gathered from Controller according to target constructor's signature.

afterConstructorCallAdvice: This advice is triggered based on constructorCallPointCut after the execution of the constructor. Current time value is gathered and the previous time value saved earlier is subtracted from it to generate the total time spend. Constructor Data's count is updated with the current information.

afterClassInitAdvice: This advice is triggered by classInitPointCut, after the initialization of object. This advices gets the class data from Controller and updates the count information of the class.

beforeExceptionThrownAdvice: This advice is triggered by exceptionThrownPointCut, just before an exception is thrown. This advice gets the exception data from controller and updates the count information of the current exception.

5.2 Core

Core part of the profiler is the part which comes into play after aspects gather the required performance data. Core part consists of below elements.

Data class is a representation of data. It contains;

- String type *signature* to hold the signature value.
- Primitive int type *count* to hold the execution time.
- Primitive long type *execTime* is average execution time
- Primitve long type *lastExecTime* is the last long type date value used by before and after advice couples to determine the total execution time. This value is not used anywhere else

Data class elements are private and they can be reached using getters and setters.

DataHolder class contains four HashMap types;

- *MethodData* HashMap: Signature is the key and *Data* is the value, it holds method execution information.
- *ClassData* HashMap: Signature is the key and *Data* is the value, it holds class count information.
- *ConstructorData* HashMap: Signature is the key and *Data* is the value, it holds constructor execution information.
- ExceptionData HashMap: Signature is the key and *Data* is the value, it holds exception information.

DataHolder's getters accept a *String* parameter and will return the *Data* object from requested HashMap. If the requested *Data* value which matches the given String type of signature does not exists, getter method will create a *Data* type with the given signature using initial values for *Data* fields and add this to HashMap and return the object. This way each *Data* type will be created at the first request.

Controller is the main controlling part of the code as the name indicates. It contains below classes and controls them.

- *Controller* type instance, this is the self of the object. Used for singleton behavior.

- *DataHolder* type *dataHolder*. This is the data of the profiling. Only controller class can reach this data directly. If any other class needs profiler data, controller can provide it with wrapped getter methods for *DataHolder* type in it.
- *RMIServer* type *rmiServer*. This is the external communication protocol of the core. It is configured and started by Controller at first initialization. This class sends the gathered profiler data information to a remote client over RMI protocol.
- *Logger* type *logger*. This is the *logger* class which logs the gathered Data to a file on disk periodically. Controller is the class which configures and starts the Logger

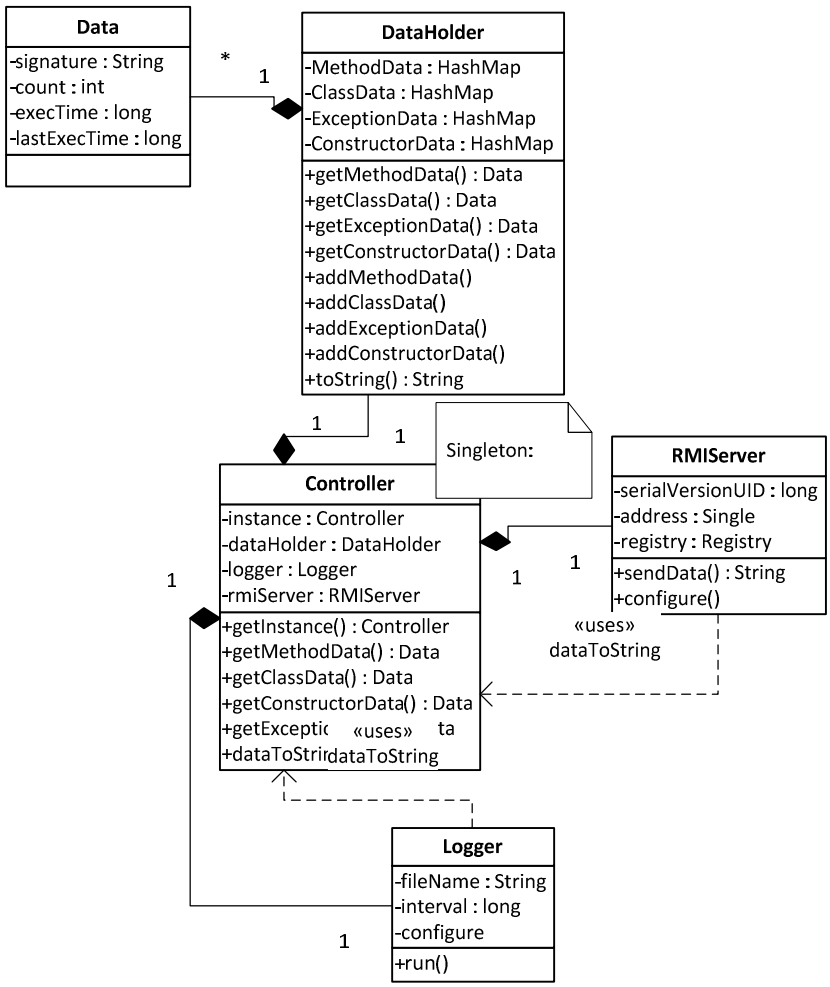

Figure 5.2 : UML diagram of profiler core, showing important classes and their important methods/attributes.

5.3 Viewer

Viewer is the part where the representation of the profiled data is done. The classes and their responsibilities are as follows;

Data class is a representation of data. It contains;

- String type *signature* to hold the signature value.
- Primitive int type *count* to hold the execution time.
- Primitive long type *execTime* is average execution time

Data class elements are private and they can be reached using getters and setters.

DataHolder class contains four HashMap types;

- *MethodData* HashMap: Signature is the key and *Data* is the value, it holds method execution information.
- *ClassData* HashMap: Signature is the key and *Data* is the value, it holds class count information.
- *ConstructorData* HashMap: Signature is the key and *Data* is the value, it holds constructor execution information.
- ExceptionData HashMap: Signature is the key and *Data* is the value, it holds exception information.

DataHolder's getters accept a *String* parameter and will return the *Data* object from requested HashMap. If the requested *Data* value which matches the given String type of signature does not exists, getter method will create a *Data* type with the given signature using initial values for *Data* fields and add this to HashMap and return the object. This way each *Data* type will be created at the first request.

Controller is the main controlling part of the code as the name indicates. It contains below classes and controls them.

- *Controller* type instance, this is the self of the object. Used for singleton behaviour.
- *DataHolder* type *dataHolder*. This is the data of the profiling. Only controller class can reach this data directly.

If any other class needs profiler data, controller can provide it with wrapped getter methods for *DataHolder* type in it.

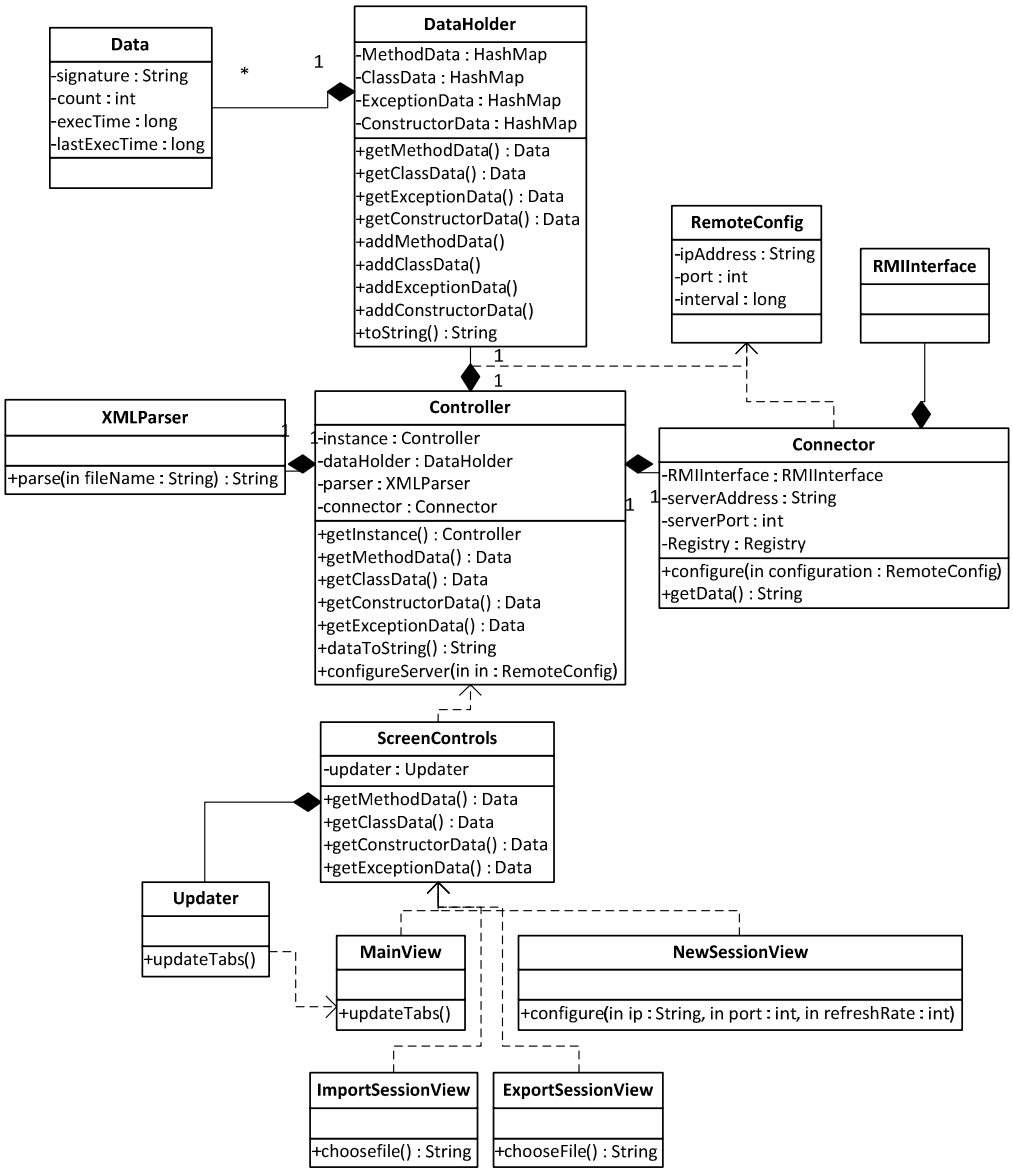

Figure 5.3 : UML diagram of profiler viewer, showing important classes and their important methods/attributes.

- *XMLParser* type *xmlParser*. This is a utility class maintained within *Controller* class. The job of *xmlParser* is to parse the profiler data in the given file and return it to *Controller* as *String*.
- *Connector* type *connector*. This is the RMI client class within the *viewer* which is able to connect to the RMI Server part of the *core* and gather

profiled data information. Enable to connect it needs to be configured with appropriate *RemoteConfig* class.

- *RemoteConfig* class is a simple data object including connection information for *Connector* to use. Passing an instance of *RemoteConfig* into the configure method of *Connector* will enable it to connect to RMI server.
- There are four screen views, which all communicate with the inner parts using *ScreenControls* class. *ScreenControls* class contains wrapper methods to gather profiler data over *Controller*. It also has an *Updater* type element *updater* which is assigned to update the views in the GUI.

The views in the viewer and their attributes are as follows;

MainView: This is the main screen of the viewer application. It has got four tab views.

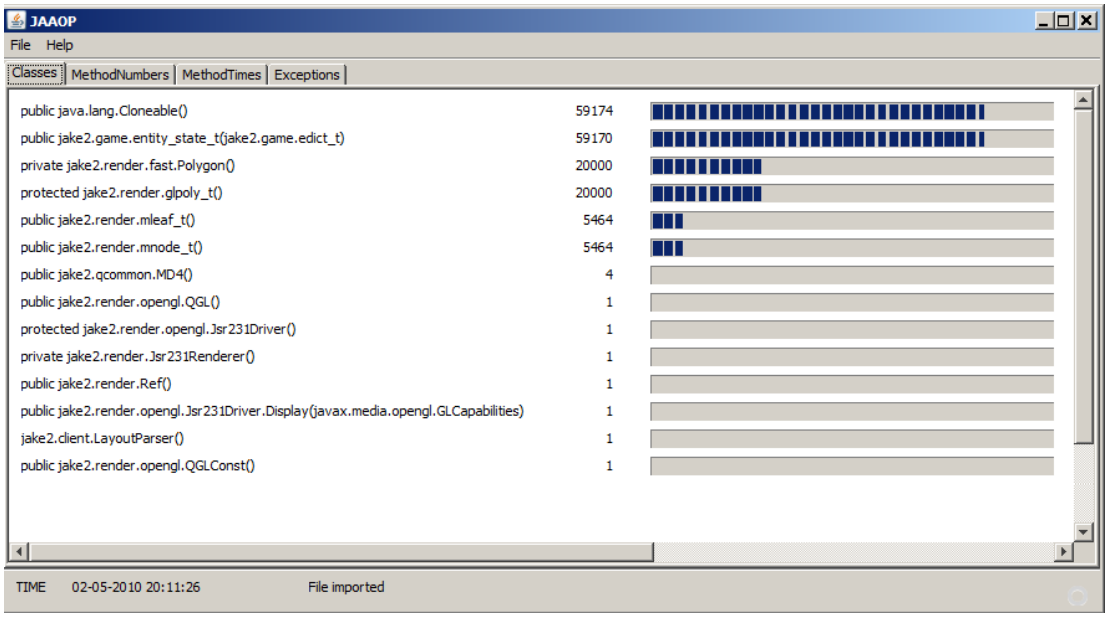

Figure 5.4 : Classes tab shows the total object initialization of classes ordered from higher to lower with a bar view to ease the visual comparison.

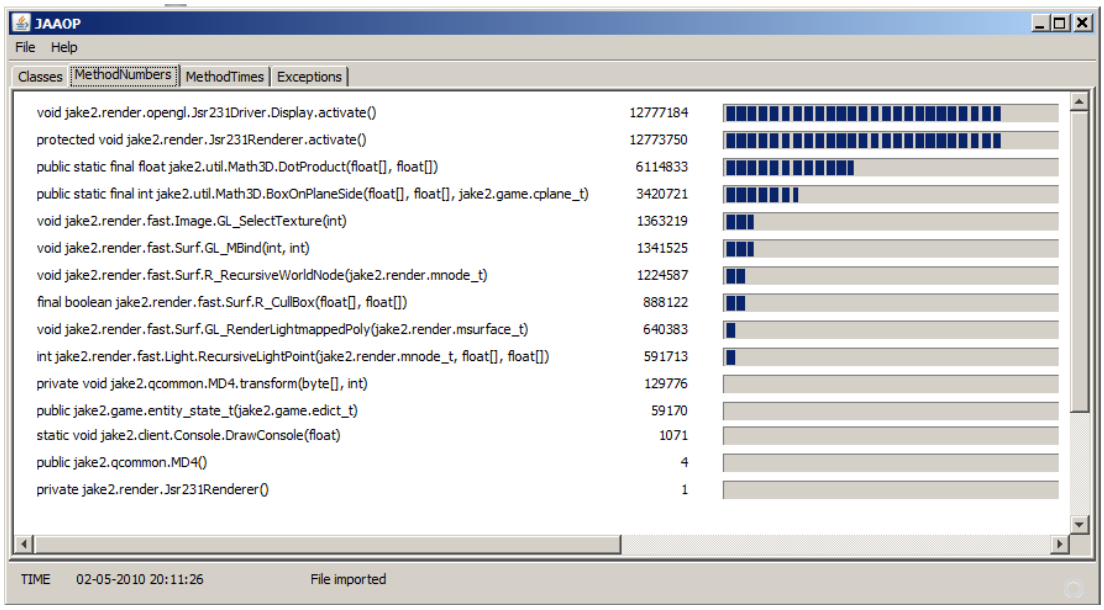

Figure 5.5 : Method Numbers tab shows the total method calls ordered from higher to lower with a bar view to ease the visual comparison.

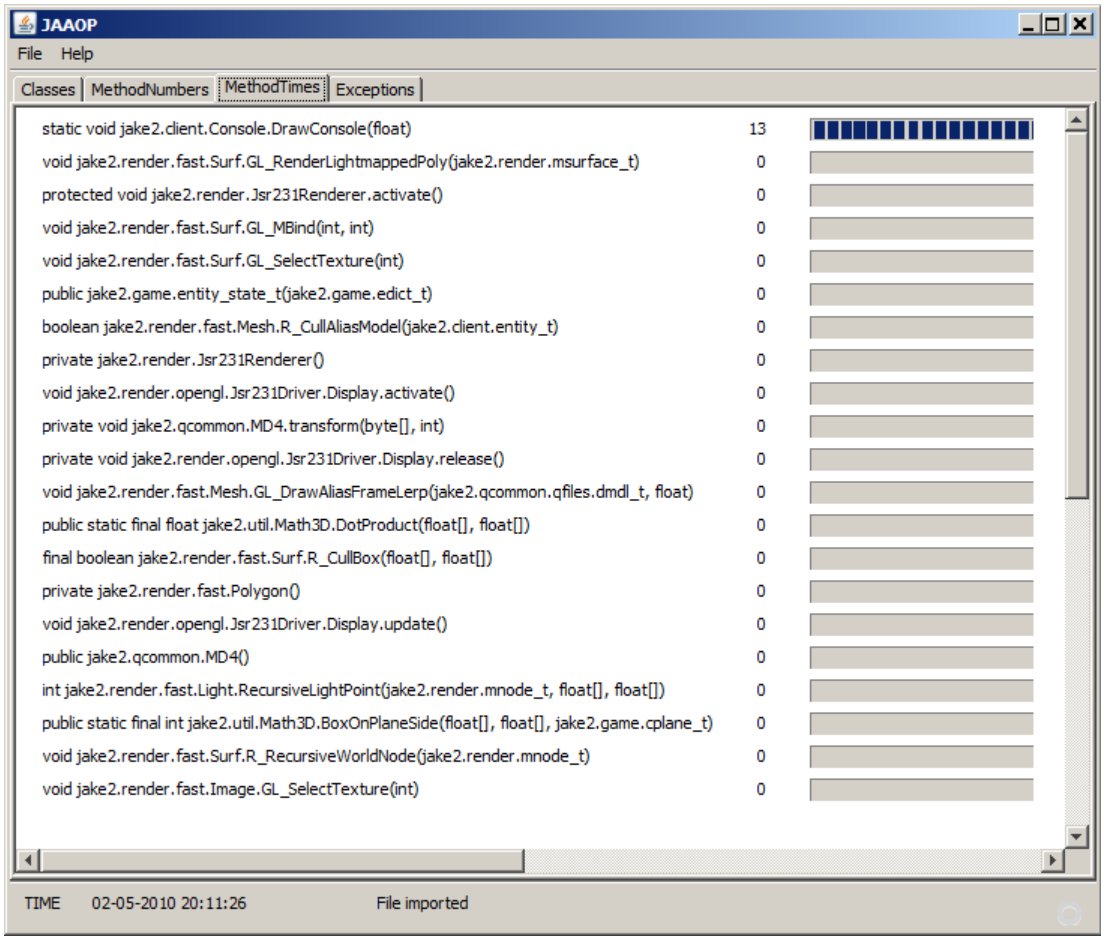

Figure 5.6 : Method Times tab shows the average method call duration of each method profiled in milliseconds.

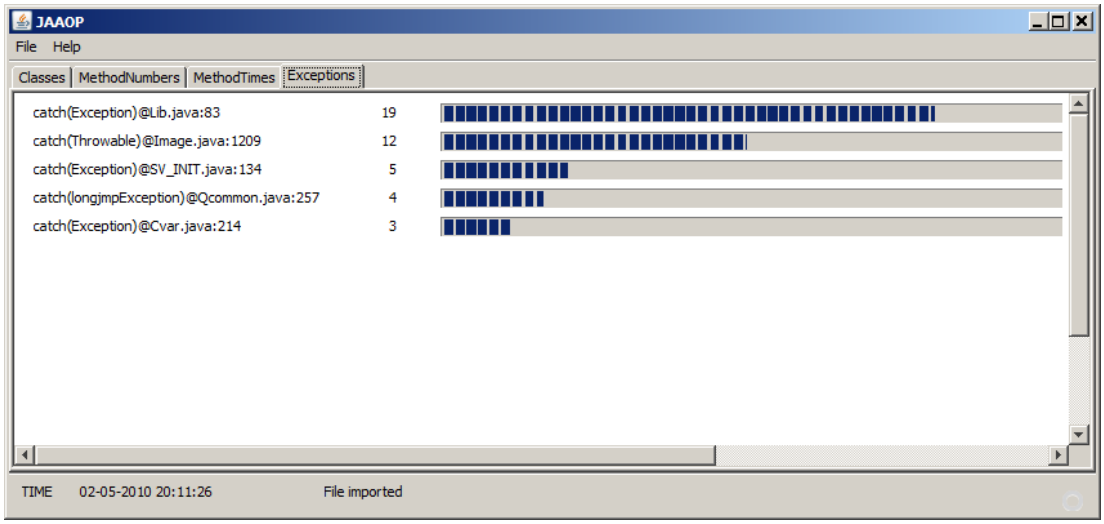

Figure 5.7 : Exceptions tab shows the thrown exceptions and how many times they were thrown.

New Session View: This is the view, which can be opened from main view by selecting File->New Session. When selected new session view will open and ask the user for below information.

- Host IP : IP address of the host machine which runs the profiling task.
- Port: Port information of the remote host machine's RMI interface.
- Refresh Rate: The period which the data will be updated from remote source and screen will be updated accordingly. Value is in seconds.

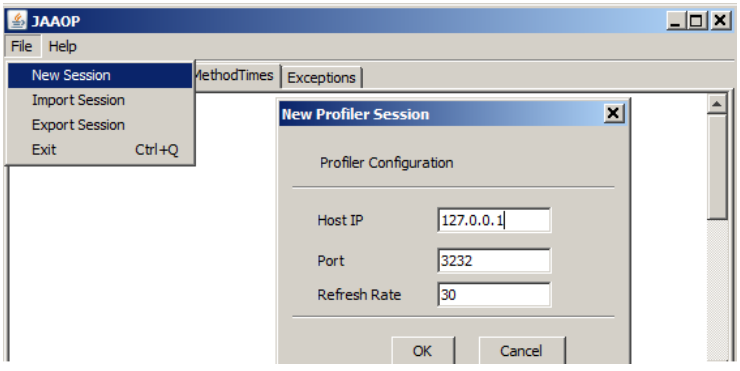

Figure 5.8 : New Profiler Session will start a remote profiling session with the given information.

When the OK button is clicked viewer will start connection to the remote profiling core via RMI and gather the so far collected data continuously. The gathered data will be ordered and presented in the four tabs of the main view.

Import View: This view is activated by selecting “Import Session” from File. When selecting a file selection box will open and ask for the xml file to import. When the file is selected and important, the data saved in that file will be shown.

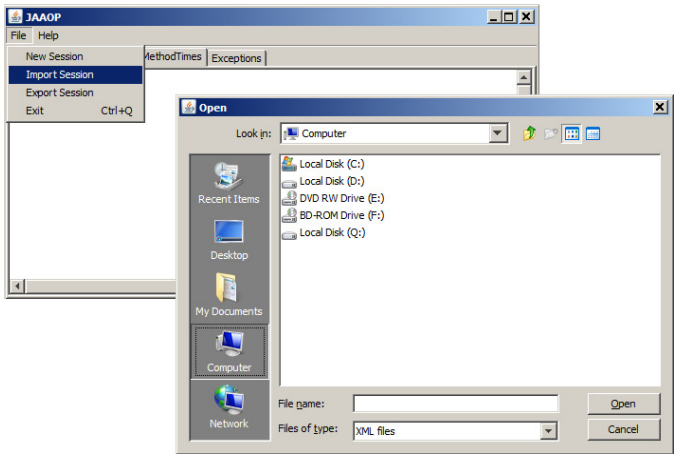

Figure 5.9 : Import option will ask for a valid profiler session data, saved earlier. When selected data in the file will be shown in viewer.

Export View: This view is activated by selecting “Export Session” from File option. This view should be selected when there is a remote profiling session going on in the viewer. When selected, export view will save the current data of the profiler to the selected file in the disk. This file can later be imported and viewed.

5.4 Important Sequence Diagrams

5.4.1 Initialization sequence

Below figure shows the sequence diagram of the initialization of the Controller element of core part of the profiler application. Within this sequence Logger is started and RMI Server is configured to serve for providing the performance data.

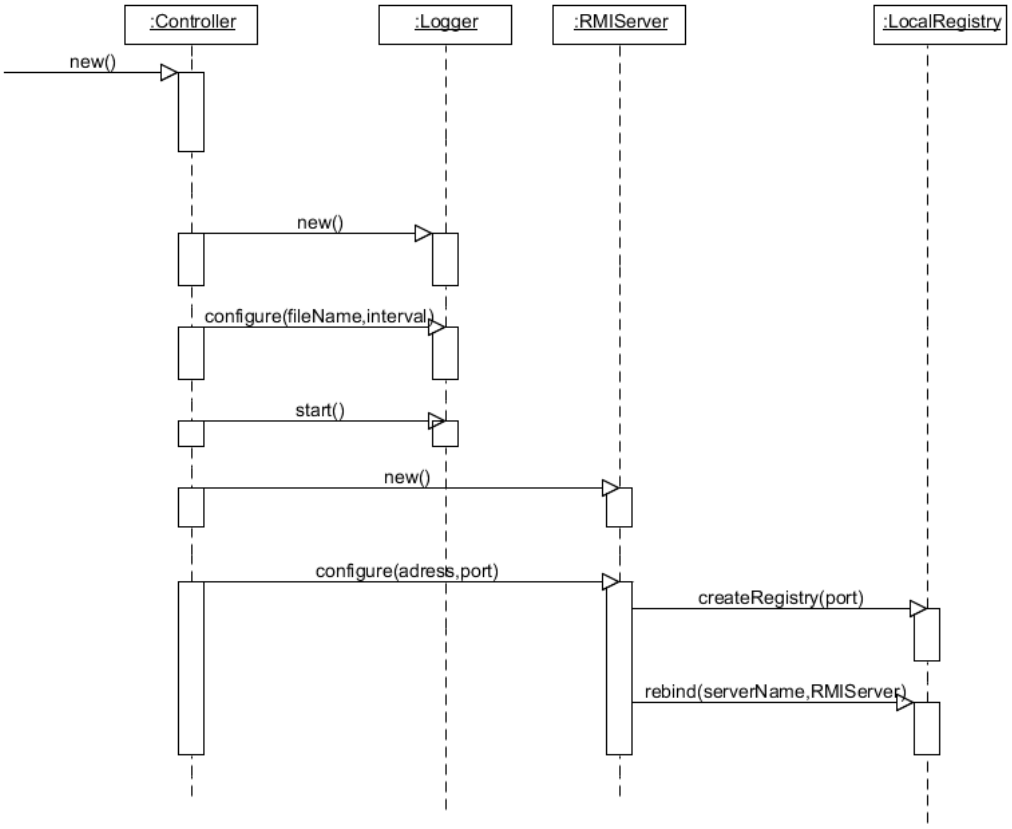

Figure 5.10 : Sequence diagram: Initialization of the Controller in core part of the profiler application.

5.4.2 Method call sequence

Below figure shows the sequence diagram of an example method call which is cross cut by MethodAspect. When an object (Object in the diagram), which includes a method marked for profiling (call in the diagram) what happens actually is nıt as straightforward as in the diagram. But to make it clearer some of the details are omitted which happen between Object and MethodAspect. The omitted part includes the matching of the pointcut in MethodAspect by the call originated from Object class. Next step is that aspect code will invoke the necessary advices. These are ;

- “beforeMethodCallAdvice” is advised by AspectJ to activate before the method call. The method call is matched by the pointcut, for methods including @TraceMethod annotation.

- "beforeMethodCallAdvice" is advised by AspectJ to activate after the method call. The method call is matched by the pointcut, for methods including @TraceMethod annotation.

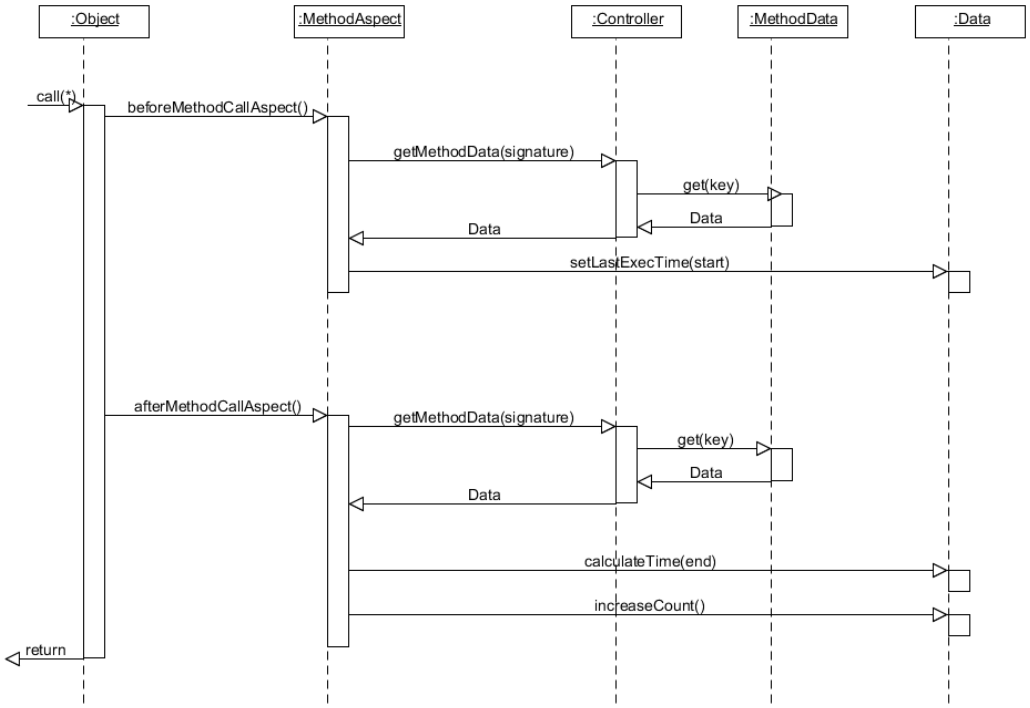

Figure 5.11 : Sequence diagram: Method execution cross cutting.

Constructor call sequence, object count sequence and exception sequence execute in a similar manner to method call sequence. The main flow is same for all except a few minor differences, like exceptions thrown and object counts are not timed, and the difference in the aspects, pointcuts and advices used in.

5.4.3 Logging sequence

Logging job is carried by a separate thread. Logger thread is first initialized and started by Controller as seen in initialization diagram. When started logger constantly iterates to gather formatted data and write to log file on disk.

Formatting of the output is done by DataHolder. Since Logger does not have direct access to DataHolder, Controller has a dataToString method, which calls DataHolder's toString method

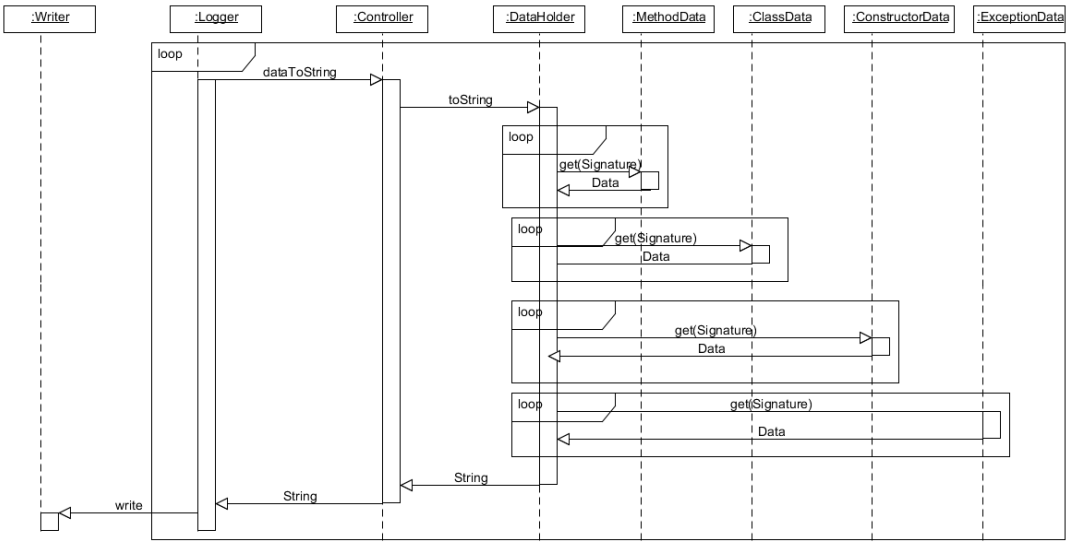

Figure 5.12 : Sequence diagram: Logger.

DataHolder's overridden toString method format's the contents of method data, class data, constructor data and exception data into a String and returns it.

5.4.4 RMI data send sequence

RMI Server is the class which provides remote access to the profiler data in a fixed xml format. Below sequence shows RMI server's internal working.

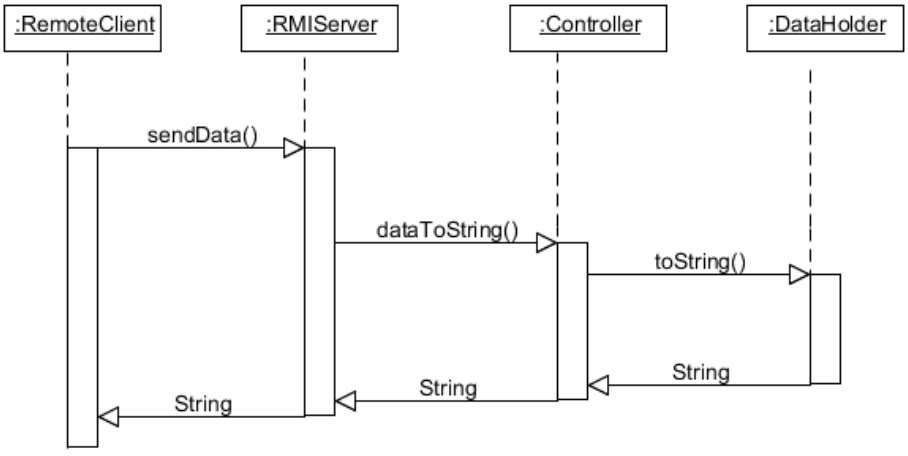

Figure 5.13 : Sequence diagram: Logger.

Although RemoteClient seems like a local class, actually it is a remote class who has the RMI interface provided by RMIServer class. RMI interface lets the remote client know that RMI Server has a sendData method which can be called, and it will return a String value. In this case sendData will generate a method call from RMIServer to Controller and then to DataHolder (just like in logger, that is why beyond DataHolder is not shown), finally a string will be generated which includes all the profiling data. This string will be sent to remote client over the network.

6. RESULTS

This section will provide overhead comparison of JAAOP compared with other profilers.

6.1 Test Application

As a real time Java application Jake2 [16] will be used while testing in this study. Jake2 is a Java 3 dimensional (3D) game engine we have chosen for testing purposes. It is a gnu public licensed (GPL) quake2 game engine from ID Software and it is a pure Java porting of quake2.

The reason behind choosing such an application is its being a real time application, where performance is very important. We think a real time application will reveal the overhead of the profiler better when compared to a non-real-time application. .

6.2 Test Environment

For the testing of the profiler and the Jake2 application below environment is used;

- Operating system: Windows 7, 64-bit edition.
- CPU: Inter Core 2 CPU, 2.00 Ghz.
- System memory: 4GB.
- GPU: Nvidia FX5200.

6.3 Calculating the Overhead of a Profiler

In order to measure the overhead of a profiler, the actual method execution times of the application must be known. However, without the use of a profiler, there is no way to gather such data.

The required information is the profiler overhead ratio according to its non-profiled state. To get this information, actually, there is no need to know the execution time

of each method. Benchmarking the application will be sufficient to measure the profiler overhead.

Luckily, having a 3D game engine allows benchmarking. For this purpose, an already played and recorded demo map is used. It is repeated to measure the average frame per second (FPS) that jake2 can generate. This method is being used on current games by graphics vendors to prove how powerful their product is. To prevent any other environmental issues, each test is run at least 10 times repeatedly using the same configuration.

6.4 JAAOP Overhead Analysis

Before presenting the overall benchmark results of a number of profilers along with JAAOP, a small overhead analysis of JAAOP will be given.

First item to discover is the overhead of AspectJ alone. In order to produce that information, a single aspect which does nothing but just cuts across all the methods is applied to test application. This approach can reveal the overhead introduced by AspectJ because no operation is being performed in the aspect, it just cross cuts. Table 6.1 shows the results.

Table 6.1 : AspectJ isolated overhead

Invocation of Jake2	Frame per Second	%Overhead
Without AspectJ	110	0
With AspectJ around advice	29,6	73
With AspectJ before-after advices	55	50

It is interesting to see the difference between before-after couple vs. around, but more interesting is the %50 overhead of AOP in its best case (regarding to around). The overhead seen in Jake2 might not be applicable to non-real time systems, because Jake2 is a real time application and it can generate over 1000 method calls per second. For this analysis, %50 overhead due to AOP means JAAOP will have at least %50 overhead in full profiling of Jake2 plus its internal code overhead.

After looking at AspectJ's initial overhead, this section will cover the overhead introduced by the mechanisms employed in JAAOP core. Considering that JAAOP

profiles about 1000 methods that are invoked each second, the core operations are naturally another source of overhead.

The main operations carried out in JAAOP core and their overheads are shown in Table 6.2 :.

Table 6.2 : JAAOP core operations overhead

Operation	Frame per Second	%Overhead
Start Stop timer	53	2
Get method signature	43	11
Data map lookup and update	36	17
Total	21,5	30

It should be noted that Table 6.2. shows the additional overhead of each operation, this means do not forget about already existing AOP overhead. Operations and their calculations are as follows;

Start-stop time: This is the additional overhead that is brought by gathering the current system twice. One for start and one for stop, in order to calculate the method execution time. To calculate this, Jake2 application is profiled with a before after advice. The only operation carried within these advices was starting getting the time.

Method signature: This is the additional overhead of getting the signature of the advised point cut's method. Calculation of the value is done in a similar way with earlier operation. Having a before after advice couple who only gets the signature of the advised method.

Data Map: This is the additional overhead of doing a lookup in data map for each method and updating the value of in it. The method for calculating is by having a before after advice couple added into Jake2. The advices only do a data map lookup and a value update in it.

Total: This value is the additional overhead of JAAOP on top of AOP. The calculation here is done by profiling every method of Jake2 with JAAOP.

It makes sense the total value is the sum of operations values, because these operations are what is done in JAAOP's aspects. There might be additional operations in JAAOP which also adds overhead besides of these operations but their overhead values are so insignificant and there is no need to discuss.

With this analysis, we can say that JAAOP core seems to add an additional 30 percent overhead on top of already existing AOP/AspectJ overhead. Together AOP and JAAOP introduce a total of 80 % of overhead. However, it should be noted that 80 % overhead is due to profiling the whole application, which means cross cutting every single method.

6.5 Profiling a Real Time Application Jake2

Even though the analysis in the previous section shows that an AOP based Java profiler introduces a quite large overhead, other profilers also need to be benchmarked to reach a conclusion.

6.5.1 Overall profiling

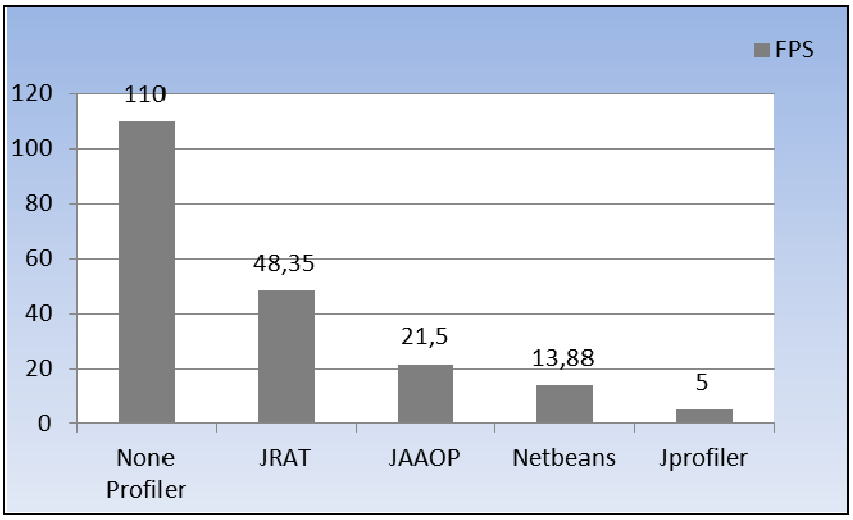

Figure 6.1 : FPS rate results for overall profiling of Jake2 (higher is better).

Figure 6.1 : shows the profiling benchmark results of Jake2 where all the classes and methods were being profiled. The lowest overhead belongs to Jrat and the highest overhead belongs to Jprofiler. JAAOP stands just in the middle of Netbeans and Jrat. One could expect Netbeans and Jprofiler to produce better results. However, the main reason for the lower scores is just the side effect of being a powerful profiler. The additional capabilities such as heap walking, memory profiling, CPU profiling, thread profiling, etc. result in higher overhead. It should be noted that both Jprofiler and Netbeans profiler are used with their minimum possible profiling options, trying to profile only method counts, method execution times, object initialization times, object counts and exceptions.

A similar explanation exists for Jrat's lower overhead; because Jrat has fewer capabilities then the rest, it scores better.

6.5.2 Partial profiling

Benchmarking is repeated on a case where we know exactly which points to profile. This approach is not very applicable for overall profiling and benchmarking of software but rather is a method to be used during development phase. As those exact points, we've chosen top 25 methods in the Jake2 application where the most time is spent. The total time spent in those 25 methods is around %80 of the execution time of Jake2 methods.

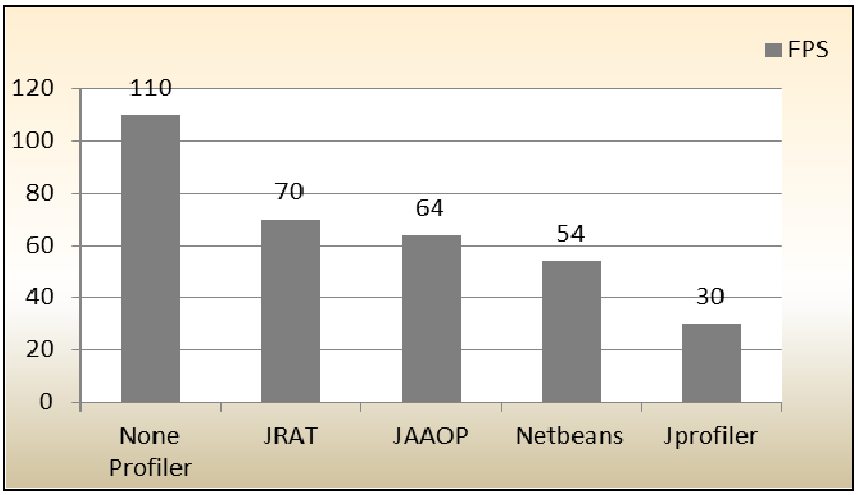

Figure 6.2 : FPS rate results for partial profiling of Jake2 (higher is better).

It is clear that the difference between JAAOP and Jrat is smaller when the numbers of profiled methods are limited. Jrat is only %6 better than JAAOP. A similar improvement is also true for Netbeans. Jprofiler still has the highest overhead among the profilers that are used.

7. CONCLUSION

The aim of this study is to investigate whether the use of AOP techniques with AspectJ contributes to the development of an efficient profiler. It seems that having a cross cutting nature AOP technology is a good choice for programming a profiler at the first place. Unlike the other alternatives, such as JVMPI/JVMTI, using AspectJ for implementing AOP behavior is really easy and efficient for a Java programmer.

Table 7.1: Overall profiler comparison

Profiler	Full Profiler Overhead	Limited Profiler Overhead	Method Profiling	Run Time Analysis	CPU Thread Memory Profiling
JAAOP	%80	%36	Y	Y	N
JRAT	%56	%42	Y	N	N
NETBEANS	%87	%51	Y	Y	Y
JPROFILER	%95	%72	Y	Y	Y

The overhead it introduces is the most important efficiency characteristic of a profiler. Table 7.1 summarizes the full and limited profiling overhead results for the profilers tested, along with their additional characteristic capabilities.

Those results indicate that Jrat is the profiler with the least overhead and the least capabilities. On the other hand, Jprofiler introduces the most overhead while it has the richest set of capabilities (not all are listed in Table 7.1). Netbeans has a fewer capabilities than Jprofiler and introduces less overhead during profiling. JAAOP stands between the two extremes, with capabilities, which are more than those of Jrat and less than those of Netbeans and Jprofiler. However, its profiling overhead is less than both Netbeans and Jprofiler.

There is a tradeoff between the capabilities that a profiler has against the amount of overhead it produces; a richer set of capabilities results in a higher overhead. The profiling results show that the overhead caused by JAAOP is at an optimum level,

when its set of capabilities is considered. It has produced satisfactory performance for real time profiling. The results we have obtained enforce our argument that using AOP techniques that is, separating the nonfunctional profiling logic of an application from its functional logic, helps to achieve an efficient and effective profiler. JAAOP, with its design and implementation decisions, proves the suitability of AOP techniques and the use of AspectJ for real time application profiling.

REFERENCES

[1] **Gosling, J., Joy, B., Steele, G., and Bracha G.,** 2005: The Java language specification, third edition. Addison-Wesley.

[2] **Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier J., Irwin, J.,** 1997**:** Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming,* Jyväskylä, Finland, June 9-13.

[3] **Dynamic Proxy Classes,** Sun Java API documentation, http://java.sun.com/j2se/1.4.2/docs/guide/reflection/proxy.html, accessed at 07.05.2010

[4] **Aspect Oriented Programming with Spring,** Spring reference documentation, http://static.springsource.org/spring/docs/2.5.x/reference/aop.html, accessed at 07.05.2010

[5] **JBoss AOP – Aspect-Oriented Framework for Java,** JBoss AOP Reference Documentation, http://www.jboss.org, accessed at 07.05.2010

[6**] Dynaop, a proxy-based Aspect-Oriented Programming (AOP) framework**, Dynaop project home, https://dynaop.dev.java.net/, accessed at 07.05.2010

[7] **Code Generation Library,** CGLIB home page, http://cglib.sourceforge.net/, accessed at 07.05.2010

[8] **AOP Allience White Paper**, The AOP Alliance Claim, http://aopalliance.sourceforge.net/white_paper/node6.html, accessed at 07.05.2010

[9] **AOPBenchmark,** http://docs.codehaus.org/display/AW/AOP+Benchmark, accessed at 07.05.2010

[10] **Hilsdale, E., Hugunin, J., Kersten, M., Kiczales, G., Lopes, C., Palm, J.,** 2000**:** Aspectj: the language and support tools. *Conference on Object Oriented Programming Systems Languages and Applications,* Minneapolis, Minnesota USA, October 15-19.

[11] **Russ, M.,** 2005: AspectJ Cookbook. O'Reilly, Sebastopol, CANADA.

[12**] Amitabh, S., Alan, E.,** 1994: ATOM: A System for Building Customized Program Analysis Tools, Western Research Laboratory, California, USA.

[13] **Rothman, J.,** 2000:What Does it Cost you to Fix a Defect ? And Why Should You Care?, http://www.jrothman.com/Papers/Costtofixdefect.html, accessed at 07.05.2010

[14] **JRat The Java Run Time Analysis Toolkit,** http://jrat.sourceforge.net, accessed at 07.05.2010

[15**] Netbeans Profiler,** http://profiler.netbeans.org, accessed at 07.05.2010

[16] **Jake2**, Bytonic software Jake2, http://bytonic.de/html/jake2.html, 07.05.2010