İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

STRONG MIGRATION OF JAVA THREADS

M.Sc. Thesis by
AyşeGENÇ

Department :Computer Engineering
Programme :Computer Engineering

FEBRUARY 2010

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

STRONG MIGRATION OF JAVA THREADS

M.Sc. Thesis by
Ayşe GENÇ
(504061504)

Date of submission :25 December 2009
Date of defence examination:27 January 2010

Supervisor (Chairman) :Prof. Dr. Nadia ERDOGAN (ITU)
Members of the Examining Committee :Assis. Prof. Dr. Sanem Sarıel Talay (İTÜ)
Assis. Prof. Dr. Yunus Emre Selçuk (YTÜ)

FEBRUARY 2010

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

JAVA İPLİKLERİNİN BAĞLAM BİLGİSİYLE BİRLİKTE GÖÇ ETTİRİLMESİ

YÜKSEK LİSANS TEZİ
Ayşe GENÇ
(504061504)

Tezin Enstitüye Verildiği Tarih :25 Aralık 2009
Tezin Savunulduğu Tarih :27 Ocak 2010

Tez Danışmanı : Prof. Dr. Nadia ERDOĞAN (İTÜ)
Diğer Jüri Üyeleri : Y. Doç. Dr. Sanem Sarıel Talay (İTÜ)
Y. Doç. Dr. Yunus Emre Selçuk (YTÜ)

ŞUBAT 2010

FOREWORD

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

February 2010

Ayse GENC

Computer Engineer

TABLE OF CONTENTS

Page

SUMMARY.....xii
1. INTRODUCTION.....1
1.1Purpose of the Thesis.....2
1.2Background.....2
1.3Roadmap.....4
2. JVM CHARACTERISTICS.....5
2.1 The class file (byte code) format:.....5
2.2 Run Time Data Areas.....8
3. BCEL (Byte Code Engineering Library).....11
3.1 JavaClass.....12
3.2 Class repository.....13
3.2.1 Accessing class file data 13
3.2.2 Analyzing class data 14
3.3 ClassGen.....14
3.3.1 Types 14
3.3.2 Generic fields and methods 14
3.3.3 Instructions 15
3.3.4 Instruction lists 15
3.4 BCEL Projects.....17
4. BRAKES.....19
4.1 Implementation of Thread Serialization.....20
4.1.1 Serializable tasks.....20
4.1.2 Capturing state 21
4.1.3 Reestablishing state 22
4.2 Implementation of Transformer.....24
4.3 Brakes Projects.....25
5. PROPOSED EXTENTION OF BRAKES (RemoteBrakes).....27
5.1 Problems with Brakes.....27
5.2 Proposed Solution : RemoteBrakes.....28
5.3 Migration over different JVMs.....30
5.3 Applications developed with RemoteBrakes&RMI.....31
5.3.1 Distributed Matrix Multiplication 32
5.3.1.1 Class Diagrams 33
5.3.1.2 Output 38
5.3.2 Bouncing Balls.....38
5.3.2.1 Class Diagrams 39
5.3.2.2 Output 44
5.3.3 Analysis on Result of Implementations.....45
6.APPLICATION DEVELOPMENT BY USING REMOTEBRAKES.....47
6.1 PreRequsities.....47
6.2 Config file.....47

6.3 Transformer.....48
6.4 Tasks.....49
7. CONCLUSION AND RECOMMENDATIONS.....51
REFERENCES 53
APPENDICES.....55
CURRICULUM VITA.....75

ABBREVIATIONS

JVM	**:** Java Virtual Machine
ORB	**:** Object Request Broker
RMI	**:** Remote Method Invocation
BCEL	**:** Byte Code Engineering Library
JSR	**:** Java Specification Request
LPI	**:** Last Performed Instruction

LIST OF FIGURES

Page

Figure 2.1 :Compilation and execution of Java classes.....5
Figure 2.2 :Java class file format.....6
Figure 2.3 : Frame.....9
Figure 4.1 : State Capturing.....22
Figure 4.2 : State Reestablishment.....23
Figure 4.3 : Stack Frame Before invoking a method.....24
Figure 5.1 : Task-Context relation in Brakes.....28
Figure 5.2 : Task-Context relation in RemoteBrakes.....29
Figure 5.3 : RMI illustration.....31
Figure 5.4 : MatMultiMain class definition.....33
Figure 5.5 : MatMulti class definition.....34
Figure 5.6 : MatMultiMainServerImpl (remote object) class definition.....35
Figure 5.7 : MatMultiMainServerImpl (remote object) class definition.....37
Figure 5.8 : BounceThread class definition.....39
Figure 5.9 : Ball class definition.....40
Figure 5.10 : BounceThreadServerImpl class definition.....42
Figure 5.11 : The suspended ball and its expected path.....44
Figure 5.12 : The reestablished ball and its path.....45
Figure 6.1 : Configuration file of Brakes framework.....48

STRONG MIGRATION OF JAVA THREADS

SUMMARY

Today mobility is an important aspect of distributed applications and have several applications of use. For developing mobile applications, Java has been put forward as the reference platform. A Java program is compiled into machine-independent bytecode and can execute on any system, as long as a Java Virtual Machine (JVM) is installed on this system. Today, JVM is ported on almost every kind of hardware and software platform and can be seen as a universal machine and provides useful sevices for mobility of code and data.

But Java does not provide any service for serialization of the running applications. If a running application migrates to a new location, only by using object serialization and class loading mechanisms, the execution state of application is lost. It means, in the new location the migratory application can access its code and data but has to restart the execution from the beginning To migrate a running application, some state information has to be saved and transferred to the new location. In this new location, the application has to be restarted ideally in the exactly same state and at the same code position as it is before the migration.The main issue when building Java thread migration is to be able to access the thread's execution state, a state that is internal to the JVM and is not directly accessible to Java programmers.

There are some projects have chosen to add new functionality to the Java environment in order to export the thread state from the Java Virtual Machine. This solution grants full access to the entire state of Java thread but it is dependent to a specific extension of JVM which means this solution can not be used with the current Java virtual machines. In order to address Java thread migration, some projects use a solution on application level, without having a specific JVM restriction. In this approach, the application code is preprocessed before the execution in order to attach a backup object to the Java program executed by the thread, and to add new statements to this program. The new statements manage the execution state capture and restoration operation and store the state in backup object.

Since Java classes are compiled into transportable binary class files, the appropriate and platform-independent way to implement these improvements is not by developing a new compiler or changing the JVM, but by transforming the byte code. These transformations can be performed after compile-time, or at load-time. To deal with the necessary class file transformations, Byte Code Engineering Library (BCEL) API helps developers to implement their transformations properly. The Byte Code Engineering Library is designed to give users a possibility to analyse, create, and manipulate (binary) Java class files.

Brakes , a thread serialization technique, is described in the context of middle ware support. It presents a simple execution model that guarantees correct thread migration semantics when moving a thread to another location. But Brakes is not designed for migration between different JVMs. We will propose an extension to

Brakes, named RemoteBrakes, to be able to move the threads between different JVM instances and allow context-switching between tasks within different JVMs. This thread serialization mechanism is implemented by extracting the state of a running thread from the application code that is running in that thread. To extract the state of a running thread, a **byte code transformer** modifies the application code by inserting code blocks that perform the actual capturing and reestablishing of the current thread's state. So this thread migration mechanism is portable across standard JVM platforms.

By using Remote Method Invocation (RMI) of Java, an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine.

In this work, two different kind of applications are developed with RemoteBrakes&RMI integration. In this first application a distributed matrix calculation is demonstrated, one of the threads migrates in middle of its job and continues at the new JVM within a different task object. Second application is a Bouncing Balls application, one of the balls migrates to the other JVM with its task. With this application, migration is visual and you can track the path of the migrator ball.

The cost of Strong Migration of Java threads, is 11% to 20% space overhead on application codes.

JAVA İPLİKLERİNİN BAĞLAM BİLGİSİYLE BİRLİKTE GÖÇ ETTİRİLMESİ

ÖZET

Günümüzde mobilite/göç edebilirilik dağıtılmış uygulamalar için önemli bir konudur ve kullanılabileceği pek çok alan mevcuttur. Mobil uygulamalar geliştirmek için Java referans platform olmuştur. Bir Java programında makine bağımsız bytecode derlenir ve üzerinde Java Sanal Makinesi (JVM) kurulu olan tüm sistemlerde çalıştırılabilir. Bugün, JVM hemen hemen her tür donanım ve yazılım platformuna taşınmıştır, dolayısıyla evrensel bir makine olarak görülebilir. Aynı zamanda Java, kod ve veri taşınabilirliği için yararlı servisler sunmaktadır.

Fakat Java uygulamaları çalışma zamanında serileştirme için herhangi bir hizmet sağlamaz. Eğer çalışmakta olan bir uygulama, yalnızca nesne serileştirme ve sınıf yükleme mekanizmaları kullanarak yeni bir konuma göç ettirilirse, uygulamanın bağlam bilgisi kaybolur. Göçmen uygulamanın koduna ve verisine erişebilirsiniz ancak uygulamaya çalışmaya kaldığı yerden değil de baştan başlar. Çalışma zamanındaki bir uygulamayı göç ettirdiğimizde o zamana kadarki bağlamının korunmuş olmasını ve göç ettiği yerde kaldığı yerden çalışmasını bekleriz. Burada karşımıza çıkan sorun JVM'in bu yürütme bilgilerini programcılara vermemesi ve kendi içinde saklamasıdır.

Bazı mevcut projelerde Java ipliklerinin bağlam bilgilerine erişmek için JVM üzerinde değişiklik yapılmıştır. Böylelikle JVM'in normalde programcıya vermediği bilgilere erişilmiştir ancak Javanın platform bağımsızlığı ilkesi sekteye uğramıştır, çünkü bu bilgiye ulaşabilmek için değiştirilmiş JVM kullanmak zorunda kalınacaktır. Java ipliklerinin bağlam bilgilerine ulaşmak için diğer bir yöntem uygulama seviyesinde bu bilgilere ulaşmayı hedeflemiştir. Uygulama koduna eklenen bazı kod parçalarıyla ipliklerin bağlam bilgilerini saklanması ve gerektiğinde aktarıldığı yerde baştan yüklenmesi sağlanmıştır.

Java sınıflarının platform bağımsızlığı korumak için gerekli değişikliklerin bytecode seviyesinde yapılması en mantıklı çözümdür. ByteCode Mühendislik Kütüphanesi Uygulama Geliştirme Arayüzü(BCEL API) sayesinde bağlam bilgilerini alabilmek için gerekli değişiklikleri bytecode seviyesinde yapabiliriz. BCEL kullanıcılara bytecode'ları baştan yaratmak, üzerlerinde değişiklik yapmak ve analizi için gerekli tüm yapıları sunmaktadır.

Brakes İplik serileştirme mekanizmasıyla basit bir yürütme modeli kullanarak ipliklerin bağlam bilgisiyle göç ettirilmesi sağlanmaktadır. ByteCode Değiştirme mekanizmasıyla bağlam bilgisi saklayan ve yükleyen kodlar uygulamaya bytecode seviyesinde eklenir. Dolayısıyla Javanın platform bağımsızlığı korunmuş olur. Ancak Brakes farklı JVM'ler arasında göç için tasarlanmış değildir. Biz Brakes için, farklı

JVM arasında göçü mümkün kılan ve bağlamları farklı görevler arasında taşımaya da izin veren RemoteBrakes adında bir eklenti önermekteyiz.

Bir nesnenin başka bir Java sanal makinede çalıştıran yöntemleri çağırmasına olanak veren Uzaktan Yöntem Çağrısı (RMI) kullanarak JVMler arası göçü sağlayacağız.

Bu çalışmada, iki farklı türde RemoteBrakes & RMI ile entegrasyon uygulaması geliştirilmiştir. İlk uygulamada Dağıtılmış bir matris hesaplama gerçeklenmiştir, ve ipliklerden bir tanesi işinin ortasında JVM 'e göç edip çalışmasına başka bir görev nesnesi içinde devam edebilmektedir. Ikinci uygulama Seken Toplar uygulamasıdır, burada da bir topun (ipliğin ve görevin) başka bir JVM'e göç etmesi ve çalışmasına kaldığı yerden devam etmesi simüle edilmiştir. Bu uygulamayla topların hareketleri görsel olarak takip edilebildiğinden demo amaçlı bir uygulamadır.

Bağlam bilgisiyle beraber göç edebilmenin maliyeti kodların boyutunda %11 ile %20 civarında bir artıştır.

1. INTRODUCTION

Today mobility is an important aspect of distributed applications and have several applications of use:

- to dynamically balance the load between several machines in a distributed system.
- To reduce network traffic by moving client to closer servers
- to dynamically reconfigure distributed applications
- to implement mobile agent platforms
- to tackle user nomadism in mobile computing environments
- as a machine administration tool [1].

For developing mobile applications, Java has been put forward as the reference platform. A Java program is compiled into machine-independent bytecode and can execute on any system, as long as a Java Virtual Machine(JVM) is installed on this system[2]. Today, JVM is ported on almost every kind of hardware and software platform and can be seen as a universal machine and provides useful services for mobility of code and data[1]. The Java bytecodes can easily be transported and downloaded whenever needed by using the customizable Java class loading mechanism and this feature flattens the way of code mobility. In addition, Java's serialization mechanism allows migrating data state of objects, making object mobilty possible[2].

But Java does not provide any service for serialization of the running applications. If a running application migrates to a new location, only by using object serialization and class loading mechanisms, the execution state of application is lost. It means, in the new location the migratory application can access its code and data but has to restart the execution from the beginning[1]. To migrate a running application, some

state information has to be saved and transferred to the new location. In this new location, the application has to be restarted ideally in the exactly same state and at the same code position as it is before the migration. If migration shows this property, then it is called *transparent* or *strong migration*[5]. If the programmer has to provide explicit code to read and establish the state, then the migration is called *non-transparent* or *weak migration*[2]. Both mechanism require the capturing the state information and reestablishment of the saved state during the restart[5].

1.1 Purpose of the Thesis

The main objective of this study is to implement strong migration of Java threads between different JVMs by using Object Request Broker(ORB) such as Java Remote Method Invocation(RMI). In addition context-switching of threads is also going to be studied.

1.2 Background

The main issue when building Java thread migration is to be able to access the thread's execution state, a state that is internal to the JVM and is not directly accessible to Java programmers[1].

There are several projects addressing Java Thread migration. Sumatra[6], Merpati[7], CIA[8] and JavaThread[1] projects have chosen to add new functionality to the Java environment in order to export the thread state from the Java Virtual Machine. This solution grants full access to the entire state of Java thread and in addition it is dependent to a specific extension of JVM which means this solution can not be used with the current Java virtual machines[10].

In order to address Java thread migration, some projects such as Wasp[5], JavaGo[9],Brakes[2] and JavaGoX[4] use a solution on application level, without having a specific JVM restriction. In this approach, the application code is preprocessed before the execution in order to attach a backup object to the Java program executed by the thread, and to add new statements to this program. The new statements manage the execution state capture and restoration operation and store the state in backup object. This backup object can be serialized using object serialization[1]. In application level approach, the preprocessed code can be the Java

source code like in projects Wasp[5] and JavaGo[9] or can be the bytecode like in projects Brakes[2] and JavaGoX[4]. The key advantage of application-level implementationsis the portability of provided mechanisms to all Java environments[1]. Preprocessing the source code is limited compared bytecode transformation and it is not posible to extract the complete execution state, so those systems are therefore incomplete[1,2].

Whatever the level of implementation, all existing solutions impose a performance overhead on threads. The JVM level applications suffer from inducing a significant overhead on thread performance(+335%,+340%) because some of them extend the Java interpretation process and they do not support Java execution optimization. The application-level systems also suffer from the performance overhead (+88%,+225%) due to the statements added[1]. JavaThread[1] is focused on reducing this overhead[10].

In addition to the projects above, there is a Java Specification Request(JSR) in Java Community Process Program with name JSR323: Strong Mobility for Java[Url-3]. This request offers a generic way for thread mobility. The offered API transforms the class file to insert byte code instructions in order to enable the class to save its data and execution state before migration. Current status of this JSR is rejected with the following comments:

> "2008-01-17 Sun Microsystems, Inc. voted No with the following comment: Sun believes this could be an interesting area of research, but does not yet see the kind of community or vendor support that would motivate standardizing on a mechanism in the JCP."
>
> "2008-01-17 Intel Corp. voted No with the following comment:Generally, for standardization in JCP, the request should represent the interests of varied approaches to a problem or various alternative implementations where there is a community that needs standardization in order to progress."

The common consideration is that, if thread migration need increases, the vendors will need this support of Java and Thread Serialization may be standardized even in JVM level.

1.3 Roadmap

We have discussed the related work in section 1.2. The rest of the thesis is as follows: Section 2 describes the Java Virtual Machine characteristics that is necessary to understand thread migration. Section 3 gives information about the Byte Code Engineering Library used for modifying the bytecodes in Brakes. Section 4 describes the Brakes thread serialization framework. Section 5 describes our extended Brakes proposal, which is named as RemoteBrakes and last Section 6 explains how to develop an application with RemoteBrakes.

2. JVM CHARACTERISTICS

The Java virtual machine is an virtual computing machine. It has an instruction set and manipulates various memory areas at run time, similar to a real computing machine[Url-4].

The Java virtual machine knows nothing of the Java programming language, only of a particular binary format, the `class` file format. A `class` file contains Java virtual machine instructions (or *byte codes*) and a symbol table, as well as other additional information[Url-4].

To implement the Java virtual machine correctly, you need only be able to read the `class` file format and correctly perform the operations specified therein[Url-4].

Figure 2.1 illustrates the procedure of compiling and executing a Java class: The source file (`HelloWorld.java`) is compiled into a Java class file (`HelloWorld.class`), loaded by the byte code interpreter and executed[Url-2].

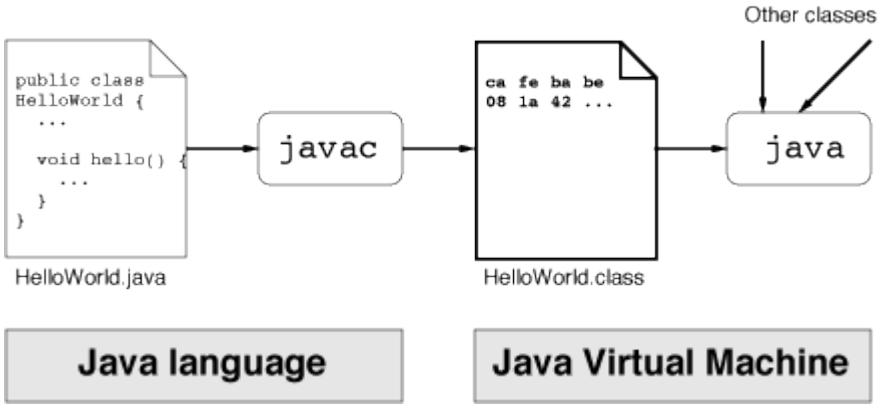


Figure 2.1 : Compilation and execution of Java classes

2.1 The class file (byte code) format:

Compiled code which is going to be executed by the Java virtual machine is represented using a platform independent binary format, which can be stored in a

file, known as the `class` file format. Representation of a class or interface is defined in the `class` file [Url-4].

Figure 2.2 shows a simplified example of the contents of a Java class file. It starts with a header containing a "magic number" (`0xCAFEBABE`) and the version number, followed by the *constant pool*, which can be roughly thought of as the text segment of an executable, the *access rights* of the class encoded by a bit mask, a list of interfaces implemented by the class, lists containing the fields and methods of the class, and finally the *class attributes*. Attributes are a way of putting additional, user-defined information into class file data structures[Url-2].

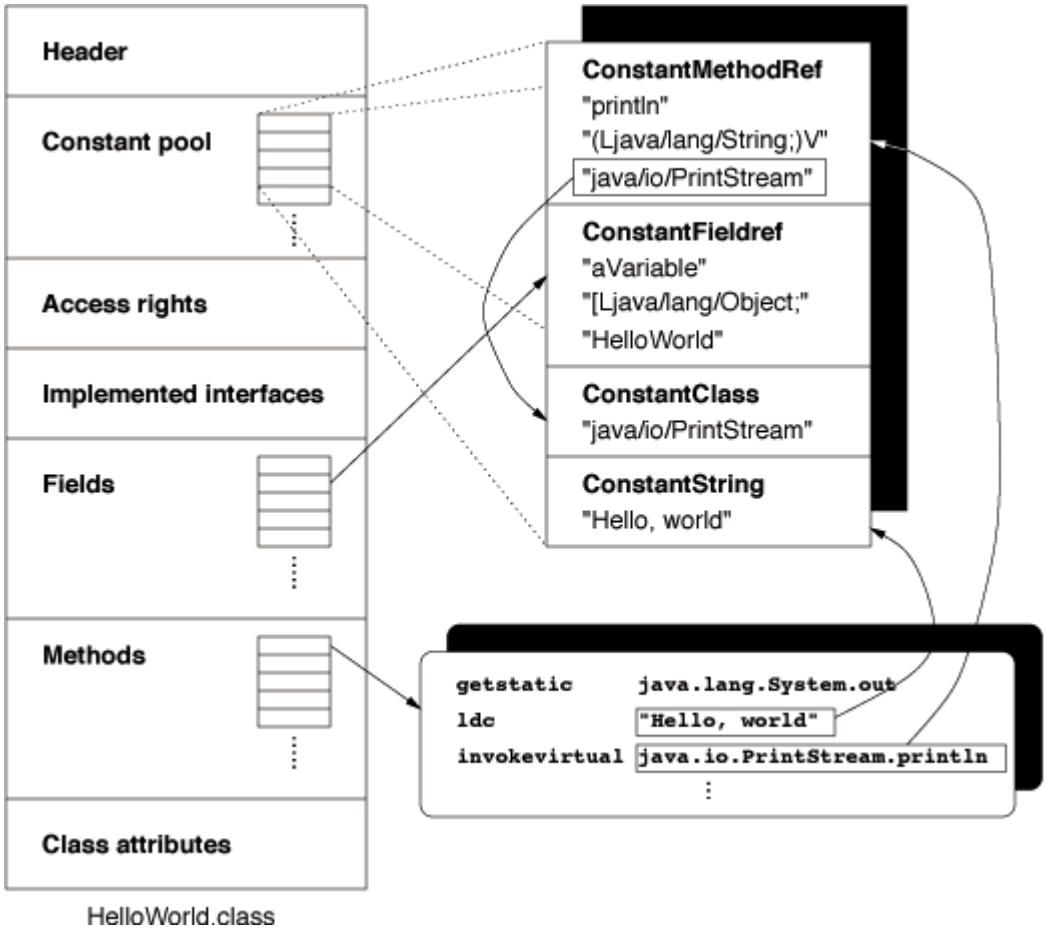


Figure 2.2 : Java class file format

The JVM is a stack-oriented interpreter that creates a local stack frame of fixed size for every method invocation. The size of the local stack has to be computed by the compiler. Values may also be stored intermediately in a frame area containing *local variables* which can be used like a set of registers. These local variables are numbered from 0 to 65535, i.e., you have a maximum of 65536 of local variables per

method. The stack frames of caller and callee method are overlapping, i.e., the caller pushes arguments onto the operand stack and the called method receives them in local variables[Url-2].

The byte code instruction set currently consists of 212 instructions, 44 opcodes are marked as reserved and may be used for future extensions or intermediate optimizations within the Virtual Machine[Url-2]. The instruction set can be roughly grouped as follows:

- Stack Operations
- Arithmetic operations
- Control flow
- Load and store operations
- Field access
- Method invocation
- object allocation
- conversion and type checking

Java is a type-safe language and the information about the types of fields, local variables, and methods is stored in so called *signatures*. These are strings stored in the constant pool and encoded in a special format. For example the argument and return types of the `main` method

```
public static void main(String[] argv)
```

are represented by the signature

([java/lang/String;)V

Classes are internally represented by strings like `"java/lang/String"`, basic types like `float` by an integer number. Within signatures they are represented by single characters, e.g., `I`, for integer. Arrays are denoted with a `[` at the start of the signature[Url-2].

2.2 Run Time Data Areas

The Java Virtual Machine defines various runtime data areas that are used during execution of a program. Some of these data areas (heap, method area) are created on Java virtual machine start-up and are destroyed only when the Java virtual machine exits. Other data areas are per thread. Per-thread data areas (pc, Java Virtual Machine Stack) are created when a thread is created and destroyed when the thread exits[Url-4].

Multiple thread execution is supported by the Java virtual machine. Each Java virtual machine thread has its own `pc` (program counter) register. At any point, each Java virtual machine thread is executing the code of a single method, which is called the 'current method' for that thread[Url-4].

A JVM is a stack-based machine[Url-5]. Each time a Java virtual machine thread created, its own *Java virtual machine stack* is also created at the same time as the thread. A Java virtual machine stack stores frames and holds local variables and partial results, and plays a part in method invocation and return. Because the Java virtual machine stack is never manipulated directly, exceptions are push and pop frames, frames may be heap allocated[Url-4].

The Java virtual machine has a *heap* that is shared among all Java virtual machine threads. Memory for all class instances and arrays is allocated by the heap runtime data area, which is created on virtual machine start-up. Heap storage for objects is reclaimed by *garbage collector*, which is an automatic storage management system, and objects are never explicitly deallocated[Url-4]

The Java virtual machine has a *method area* that is shared among all Java virtual machine threads. The method area is created on virtual machine start-up and it is logically part of the heap The method area is analogous to the storage area for compiled code of a conventional language. It stores per-class structures such as the runtime constant pool, field and method data, and the code for methods and constructors, including the special methods [Url-4].

A *runtime constant pool* is a per-class or per-interface runtime representation of the `constant_pool` table in a `class` file. It contains several kinds of constants.The runtime constant pool serves a function similar to that of a symbol table for a

conventional programming language, although it contains a wider range of data than a typical symbol table. The runtime constant pool for a class or interface is constructed when the class or interface is created by the Java virtual machine[Url-4].

A *frame* is used to store data and partial results, as well as to perform dynamic linking , return values for methods, and dispatch exceptions. A new frame is created each time a method is invoked and it is destroyed when its method invocation completes, whether that completion is normal or not (it may throw an uncaught exception). Frames are allocated from the Java virtual machine stack of the thread creating the frame. Each frame has its own array of local variables, its own operand stack , and a reference to the runtime constant pool of the class of the current method[Url-4].

In a given thread, there is only one active frame, which is the frame for the executing method. control. This frame is named the *current frame*, and its method is named the *current method*. And the class of current method is named as the *current class*[Url-4].

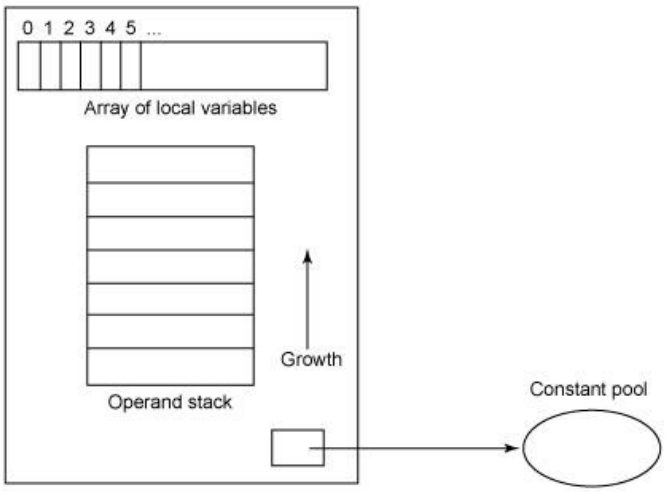


Figure 2.3 : Frame

When a frames method completes or invokes another method, it is not the current frame anymore. In case of a new method invocation, a new frame is created and it becomes current when control transfered to the new method. On method return, the current frame returns back the result of its method invocation, if there is a previous frame. The current frame is then no longer inuse, since the previous frame becomes the current one[Url-4].

A frame created by a thread is local to that thread and it is not possible to reference it by an other thread [Url-4].

Each frame contains an *operand stack*, which is a last-in-first-out (LIFO) stack. The lenght of the operand stack of a frame is calculated at compile-time and served with the code for the frames method[Url-4].

Initially, operand stack of current frame is empty. There are some Java Virtual Machine instructions to load constants/local variable's values/fields on to the operand stack and there are some instructions to take the operands from the operand stack. When it takes the operands, it operates them and pushes the results back on the operand stack. Another usage of the operands stack is to prepare parameters to be passed and to receive method results[Url-4].

3. BCEL (Byte Code Engineering Library)

There are many research projects and proposals are available for improving and extending Java language and JVM[13].

Since Java classes are compiled into transportable binary class files, the appropriate and platform-independent way to implement these improvements is not by developing a new compiler or changing the JVM, but by transforming the byte code. These transformations can be performed after compile-time, or at load-time. To deal with the necessary class file transformations, Byte Code Engineering Library(BCEL) API helps developers to implement their transformations properly[13].

The Byte Code Engineering Library is designed to give users a possibility to analyse, create, and manipulate (binary) Java class files[Url-2]. Those objects can be read from an already available file, be transformed by a program (e.g. a class loader at run-time) and dumped to a new file again. BCEL contains a byte code verifier named *JustIce*, which can throw you more information about what's wrong with your code than the standard JVM message[Url-2].

The BCEL API, also known as JavaClass, is a toolkit which is used for the static analysis and dynamic creation or transformation of Java class files. With BCEL API it is possible for developers to implement the wanted features on a high level of abstraction without a need for handling the internal details of the byte codes. BCEL is developed by using Java and freely available under the terms of the Apache Software License[Url-2].

The BCEL API mainly consists of three parts[Url-2]:

1. A component that generates a “static” view upon class files. This component is not for byte code modifications but for reading and writing class files from or to a file. It is effective for analyzing Java classes without having the source files. The main data structure is called `JavaClass`[13].

2. A package for dynamically generating or modifying `JavaClass` objects. Possible usage areas are may be to insert analysis code, to remove unnecessary information from class files, or to implement the code generator back-end of a Java compiler[13].

3. Various code examples and utilities like a class file viewer, a tool to convert class files into HTML[Url-2].

3.1 JavaClass

The "static" component of the BCEL API is available in the package `de.fub.bytecode.classfile` and represents class files[13]. All of the binary components and data structures defined in the JVM specification and explained in section 2 of this thesis are mapped to classes. The class hierarchy of JavaClass can be found at Appendix A1 as a UML diagram. Appendix A2 shows a detailed diagram of the `ConstantPool` components[Url-2].

`JavaClass` is the top-level data structure and in most cases it is created by a `ClassParser` object which is capable of parsing binary class files. Basic components of a `JavaClass` object is fields, methods, symbolic references to the super class and to the implemented interfaces[Url-2].

The constant pool is very important for all components because it serves as a central repository. `ConstantPool` objects contain an array of fixed size of `Constant` entries[Url-2].

The types of methods and objects defined by signatures. Access flags are encoded by an integer bitmask, e.g. `public static final` matches to the Java expression

int access_flags = ACC_PUBLIC | ACC_STATIC | ACC_FINAL;

As mentioned in section 2.1 already, several components may contain *attribute* objects: classes, fields, methods, and `Code` objects (introduced in section 2.3). The latter is an attribute itself that contains the actual byte code array, the maximum stack size, the number of local variables, a table of handled exceptions, and some optional debugging information coded as `LineNumberTable` and `LocalVariableTable` attributes. Attributes are in general specific to some data structure, i.e., no two components share the same kind of attribute, though this is not explicitly forbidden.

In the figure Appendix A1 the `Attribute` classes are stereotyped with the component they belong to[Url-2].

3.2 Class repository

Using the provided `Repository` class, reading class files into a `JavaClass` object is quite simple[Url-2]:

```
JavaClass clazz = Repository.lookupClass("java.lang.String");
```

The repository has methods providing the dynamic equivalent of the `instanceof` operator, and other useful routines:

```
if(Repository.instanceOf(clazz, super_class) {
  ...
}
```

3.2.1 Accessing class file data

Information within the class file components may be accessed via set/get methods. `toString()` method is defined for all of the components, that's why implementing a simple class viewer is quite easy. All of the examples used in this section have been produced this way:

```
System.out.println(clazz);
printCode(clazz.getMethods());
...
public static void printCode(Method[] methods) {
  for(int i=0; i < methods.length; i++) {
    System.out.println(methods[i]);

    Code code = methods[i].getCode();
    if(code != null) // Non-abstract method
      System.out.println(code);
  }
}
```

3.2.2 Analyzing class data

The *Visitor*[Url-6]design pattern is supported by BCEL, so it is possible to write visitor objects to traverse and analyze the contents of a class file[Url-2].

3.3 ClassGen

`ClassGen` supplies an abstraction level for creating or transforming class files dynamically by providing an interface to add methods, fields, and attributes. It makes the static constraints of Java class files like the hard-coded byte code addresses "generic". The generic constant pool, for example, is implemented by the class `ConstantPoolGen` which offers methods for adding different types of constants. Appendix A3 gives an overview of this part of the API.[Url-2]

3.3.1 Types

The `Type` class is used instead of the type signature syntax. Methods can use this class to define their return and argument types. Practical sub-classes are `BasicType`, `ObjectType`, and `ArrayType` which is composed of of the element type and the number of dimensions. Some predefined constants are also available for commonly used ones[Url-2].

3.3.2 Generic fields and methods

`FieldGen` objects represents the fields and they can be modified by the user. Generic methods have methods to add local variables, exceptions the method may throw, and exception handlers. Since exception handlers and local variables have references to byte code addresses, they also take the role of an *instruction targeter* in BCEL terminology. Instruction targeters has a method `updateTarget()` to redirect a reference. Generic (non-abstract) methods refer to *instruction lists* that is composed of instruction objects. References to byte code addresses are developed by handles to instruction objects. [13].

The maximum stack size required by the method and the maximum number of local variables used can be set manually or automatically decided by the `setMaxStack()` and `setMaxLocals()` methods[Url-2].

3.3.3 Instructions

Modeling instructions as objects enables programmers to obtain a high-level view upon control flow without thinking details like concrete byte code offsets. Instructions are composed of an opcode (sometimes called tag), length in bytes and an offset (or index) within the byte code. [Url-2].

Instructions are classified by sub-classing, the type hierarchy of instruction classes is illustrated by instruction API in appendix A4. The most critical family of instructions are the *branch instructions*, e.g., `goto`, that branch to targets somewhere within the byte code. It is obvious that this makes them candidates for playing an `InstructionTargeter` role. Instructions are also classified by the interfaces they implement[Url-2].

BCEL gives the opportunity to invent your own instructions[Url-2].

3.3.4 Instruction lists

Instruction handles encapsulates instruction objects and an *instruction list* is implemented by a list of those handlers.

Instruction references in the instruction lists are implemented by pointers to instruction handlers. With this implementation approach, it is very easy to append, insert or delete code areas. Since symbolic references are used, computation of concrete byte code offsets does not have to occur until the user has finished the process of generating or transforming code[Url-2].

Appending: It is possible to append instructions or other instruction lists anywhere to an existing list. The instructions are appended after the given instruction handle. All append methods return a new instruction handle which will be used as the target of a branch instruction, e.g.[Url-2]:

```
  InstructionList il = new InstructionList();
  ...
  GOTO g = new GOTO(null);
  il.append(g);
  ...
  // Use immutable fly-weight object
                        InstructionHandle            ih            =
il.append(InstructionConstants.ACONST_NULL);
  g.setTarget(ih);
```

Inserting: :It is possible to insert instructions anywhere into an existing list. They are inserted before the given instruction handle. All insert methods return a new instruction handle which will be used as the start address of an exception handler,e.g. [Url-2]:

```
InstructionHandle start = il.insert(insertion_point,
                                    InstructionConstants.NOP);
  ...

mg.addExceptionHandler(start, end, handler, "java.io.IOException");
```

Deleting: Deletion of instructions is also very direct; all instruction handles and the contained instructions within a given range are deleted from the instruction list and disposed[Url-2].

```
try {
  il.delete(first, last);
} catch(TargetLostException e) {
  InstructionHandle[] targets = e.getTargets();
  for(int i=0; i < targets.length; i++) {
    InstructionTargeter[] targeters = targets[i].getTargeters();
    for(int j=0; j < targeters.length; j++)
       targeters[j].updateTarget(targets[i], new_target);
  }
}
```

Finalizing: After instruction list modified and ready to be dumped to pure byte code, all symbolic references must be mapped to real byte code offsets. This is done by using `getByteCode()` method which is called by default by `MethodGen.getMethod()`. After that, `dispose()` should be called,so that the instruction handles can be reused internally. This helps to improve memory usage[Url-2].

```
InstructionList il = new InstructionList();

ClassGen  cg = new ClassGen("HelloWorld", "java.lang.Object",
                            "<generated>", ACC_PUBLIC | ACC_SUPER,
                            null);
MethodGen mg = new MethodGen(ACC_STATIC | ACC_PUBLIC,
                             Type.VOID, new Type[] {
                               new ArrayType(Type.STRING, 1)
                             }, new String[] { "argv" },
                             "main", "HelloWorld", il, cp);
...
cg.addMethod(mg.getMethod());
il.dispose(); // Reuse instruction handles of list
```

3.4 BCEL Projects

BCEL has been used in many projects.

Some of them can be listed as follows[Url-7]:

- Jboss
- Brakes: Thread serialization
- Transformations for Model Checking Distributed Java Programs
- Barat compiler framework
- Javacode coverage tool for Junit
- Heap Usage Profiling

4. BRAKES

Brakes thread serialization technique is described in the context of middle-ware support. It presents a simple execution model that guarantees correct thread migration semantics when moving a thread to another location[2]. This feature provided significant benefits for this thesis.

Brakes currently consists of two parts:

- A ByteCode transformer (based on version 1.4 of the ByteCode Engineering Library) which instruments Java classfiles so they are able to capture their current internal state at any given time
- A small framework which uses the ability of the 'patched' classes to allow Java threads to pause and resume whenever desirable.

Brakes is developed at the Katholieke Universiteit Leuven, Belgium, by the Distributed systems and computer Networks (DistriNet) research group and it is licensed under the Academic Free License (AFL)

The BRAKES project started as part of the CORRELATE Project and it is the main source of inspiration for the *JavaFlow* component in the Apacha Jakarta Project [Url-1]. *JavaFlow* has an additional dynamic bytecode transformer.

To be able to migrate a thread, its execution must be suspended and its Java stack and program counter must be captured in a serializable format which is then going to be sent to the target location. At the target location, the stack must be reestablished with serialized stack and the program counter must be set to the serialized code position and then the thread must be rescheduled for execution[3]. This is how strong thread migration defined.

4.1 Implementation of Thread Serialization

The thread serialization mechanism is implemented by extracting the state of a running thread from the application code that is running in that thread. To extract the state of a running thread, a **byte code transformer** modifies the application code by inserting code blocks that perform the actual capturing and reestablishing of the current thread's state. So this thread migration mechanism is portable across standard JVM platforms[3].

Brakes offers tasks as a complement to JVM threads, in order to perform this thread serialization mechanism practical for migration. The JVM thread is encapsulated by a task, whose execution state is same with its threads execution state[3].

4.1.1 Serializable tasks

Task serialization is possible at every execution point[3]. It is very import to clarify that tasks are not serializable at each moment, but only after the execution state capturing process is completed and before the execution state reestablishment process is initiated[2].

Context object is defined for each task and the captured/reestablished execution state is hold within this Context object[3]. This context object can be serialized by using Java object serialization mechanism[2]. Execution state is captured by means of the current Java stack and the last executed instruction (i.e. program counter index) into the task’s context. In order to capture and reestablish a task’s thread state, byte code instructions are inserted into the code of the application code[2].

Also, there are boolean flags for each task, that are representing the execution mode of the task[3]. A task can be in three different modes of execution:

- Running: normal execution mode of the task.
- Capturing: mode of capturing tasks current execution state into its context. When this mode is on, its flag `isSwitching` is set.
- Restoring: mode of reestablishing tasks previous execution state from its context. When this mode is on, its flag `isRestoring` is set[2].

Task deserialization is not similar to the Java object serialization, it is not automated and it needs to be initiated. This initiation is performed by calling two special primitives, which are each defined as a method on Task. These primitives are for requesting capturing and reestablishment of a task's execution state[2].

- `public void yield()`. This task serialization mechanism requires that state

capturing must be initiated within the execution of the task which is going to be migrated. This yield() method is called whenever the current executing task wanted to be suspended. The execution state must be captured in a serializable format[2].

- `public void resume()`. By invoking this method on a `Task` object, task is going to be scheduled with the task scheduler. When the task scheduler restarts the task (by calling `start()` on it), the execution state of the task is first reestablished to the resumed execution state. This method will be called at the target location after it receives the migrating thread[2].

There is a **TaskScheduler** that schedules the execution of tasks. The task scheduler controls the execution order of tasks. When the scheduler starts a task, the task is assigned to a JVM thread for executing itself. In principle, the execution model allows creating a new thread for each task, let the different threads run concurrently and rely on efficient JVM implementation for context switching support[2].

4.1.2 Capturing state

If a task enters its capturing state, then the current method returns after saving its stack frame. The execution state of an running task is the list of stack frames in the method call stack of the task's JVM thread, Brakes goes through that stack and performs state capturing for each stack frame[2]. This process can be summarized as in Figure 4.1.

In order to check the capturing flag, bytecodes are inserted in each method after each invokeinstruction occurence in the code [2,3]. Whenever this flag is set, the first thing that method will do is to save its stack frame and its last performed invoke instruction (LPI), and then directly return. [3]. For every method on the stack, Brakes saves the current state frame (the one which is before executing the method's LPI). There is an artifical program counter in the task's context which is referring to the LPI[2].

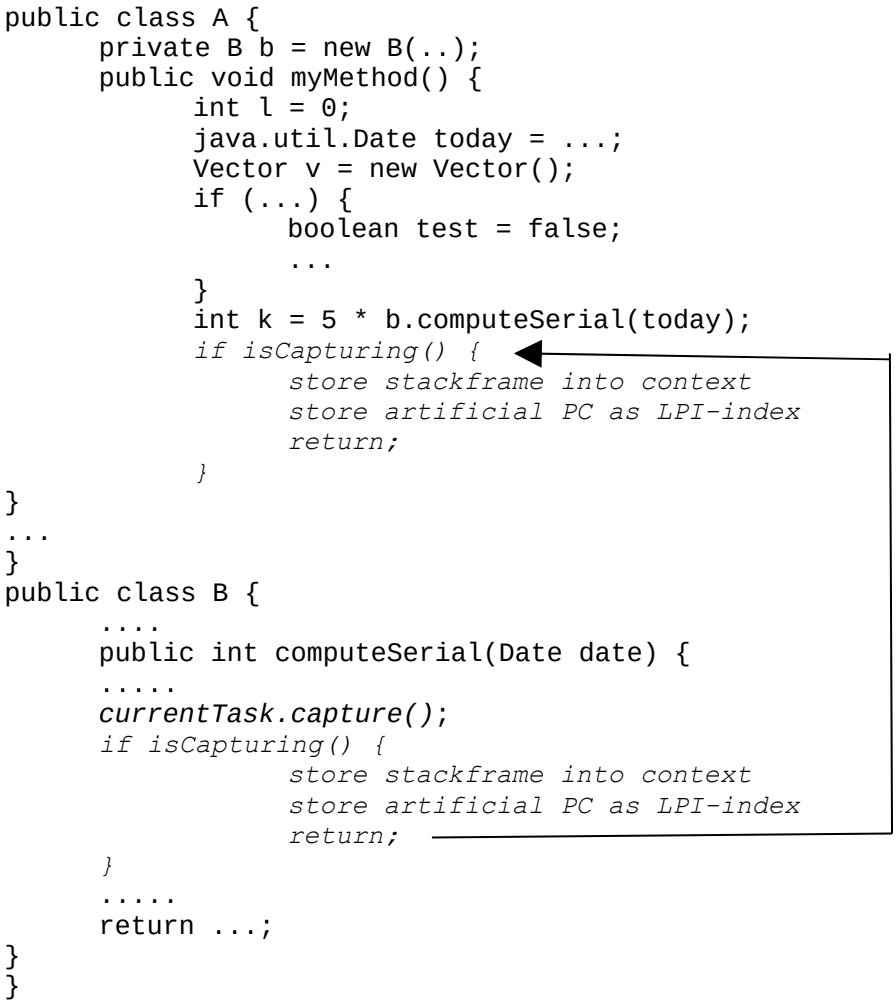


Figure 4.1 : State Capturing

4.1.3 Reestablishing state

When a task is going to be restored as shown in Figure 4.2, a new thread is started from the task[3]. Calling method resume() on a suspended Task object, reschedules this task object with TaskScheduler.Restablishment of task actually starts when the TaskScheduler schedules this task[2]. First of all, the stackframe is restored, and the LPI is checked to determine which method was the next the task was in. According to the inserted bytecode, each method restores the next stack frame, and checks the LPI for the method next to invoke. As soon as the thread reaches the captured method, the restoring flag is unset and the thread continues execution[3].

```
public class A {
      private B b = new B(..);
      public void myMethod(){
            if isRestoring() {
                  get LPI from context;
                  switch(LPI){
                  ...
                  case invokecomputeSerial:
                  load stackframe;
                  goto invokecomputeSerial;
                  case …:
                  ...
                  }
            }
            int l = 0;
            java.util.Date today = ...;
            Vector v = new Vector();
            if (...) {
                  boolean test = false;
                  ...
            }
            int k = 5 * b.computeSerial(today);
            }
}
public class B {
      public int computeSerial(Date date){
      if isRestoring() {
            get LPI from context;
            switch(LPI){
            ...
            case invokecomputeSerial:
            load stackframe;
            goto invokecomputeSerial;
            case …:
            ...
            }
      }
      .....
      currentTask.capture ();
      .....
      return ...;
}
```

Figure 4.2 : State Reestablishment

In order to prevent re-execution of an already executed method code during reestablishment, Brakes uses goto instructions for executed codes[2].

4.2 Implementation of Transformer

Transformer generates state capturing and state reestablishing code blocks in the application code. To be able to generate useful code blocks, the content of a method's call stack should be obtained before the invoke-instruction execution starts[2].

The type of the local variables visible in the scope of the instruction and the type of the values that are on the operand stack should be analyzed before the instruction is executed. This analysis is very complex because the type information is not clearly represented in the byte codes of the method. To perform this analysis, a type inference process is used which is similar to the one used in the Java byte code verifier, as described in the Java Virtual Machine specification[2,7].

When the type inference analysis is completed, rewriting process begins. Brakes transformer rewrites the original application code method per method. Each method is rewritten invoke instruction per invoke-instruction. Brakes transformer can distinguish instance method invocations, static method invocations, super-calls and constructor invocation[2].

Figure 4.3 shows the information captured and reestablished when the method's LPI is an instance method invocation.

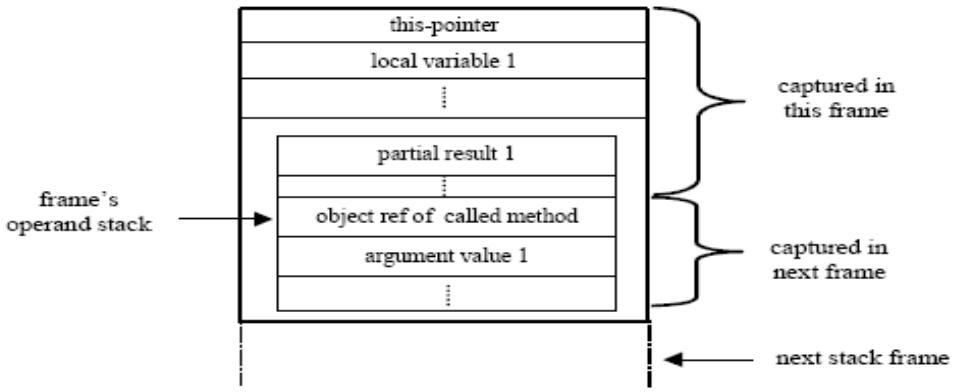


Figure 4.3 : Stack Frame Before invoking a method

4.3 Brakes Projects

There are some projects build on Brakes base.

- Distributed Brakes

In addition to Brakes, Distributes Brakes focuces on proper partitioning of distributed objects over the different machines, which is critical to the global performance of the distributed application. Methods for partitioning exist, and employ a graph-based model of the application being partitioned[12]

- MAG/Brakes

MAG/Brakes uses only the transformer of Brakes and added the following features:

- store the state information inside each object,

-the migration process can be initiated by an external source

- the user can inform to the transformer that ONLY the invocations to a specific method have to save the application's state, ignoring the other invocations.[11]

5. PROPOSED EXTENTION OF BRAKES (RemoteBrakes)

The objective of this study is to be able to implement a strong migration of threads between different JVMs.

5.1 Problems with Brakes

Brakes system has two versions:

- computation-serial: Also named as *standard version.* This version does not allow JVM context-switching and preemptive scheduling. In this standard version, concurrent thread execution is not posible. This version's prototype is presented at ASA/MA'2000 Conference[2,4]
- computation-parallel: This is an unoptimized addon on standard version and does allow JVM context switching and premptive scheduling between different threads. This version is not fully tested.

Since our intention is to migrate threads between different JVMs, the standard version of Brakes does not serve our purpose. The parallel version, which allows context switching will be the base of our implementation.

Current parallel Brakes system stays a 'proof-of-concept' and needs to be extendended for realworld usage[Url-1]. The following issues needed to be handled:

1. Brakes is designed for centralized programs that run on a single JVM. However, we intended to migrate threads between different JVMs.
2. Thread suspension and reestablishment is initiated within the task and it is not possible to initiate by an external controller. This will be a problem in case of a load balancing algorithm, where an external controller will decide to migrate a thread.

3. The contexts of the suspended tasks are saved but the saved contexts are used within the same task, it is not possible to migrate a context and resume a task with a new context.

This thesis proposes an extension to Brakes, to be able to move the threads between different JVM instances and to allow context-switching between tasks within different JVMs. We have named this new version of Brakes as *RemoteBrakes*.

5.2 Proposed Solution : RemoteBrakes

In order to migrate threads between different JVMs, we need to extend the current parallel version of the Brakes.

In the current parallel Brakes, each task is related with a Context object. When a task is suspended its Context is saved, and similiarly when a suspended task is resumed, its execution state is updated with by the context of its Context as in Figure 5.1. This works well if you are migrating threads on the same JVM and on the same order, because the thread-task mapping is internally handled inside a temporary Hashtable which is not serialized, so you don't have to specify with which particular context task has to resume. But in case of a migration between different JVMs, we may not be allowed to create a new task for each migrated thread but we may have to use an existing task.

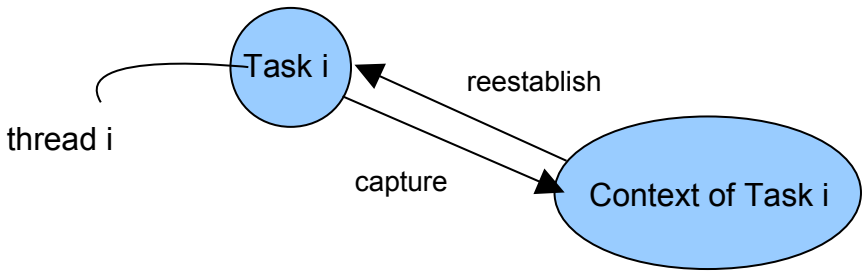


Figure 5.1 : Task-Context relation in Brakes

With our extension, a task can resume with a context of any other task. This can be possible by attaching the task information within the resume() method of Brakes. With this modification of the resume() method, named as resumeRmi(), we update the context of the task with the new task context so that it can continue execution with a different task's context, possibly in a remote location. For this reason, this Brakes extension is named as RemoteBrakes.

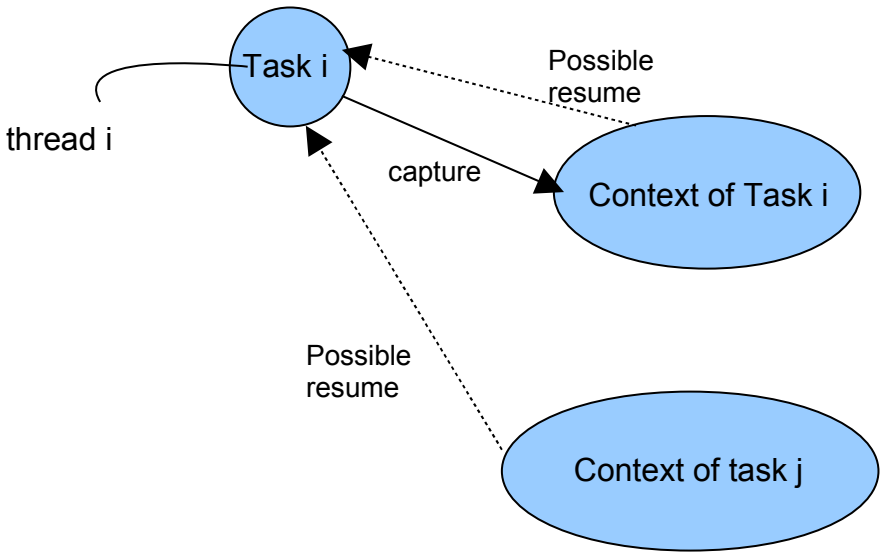


Figure 5.2 : Task-Context relation in RemoteBrakes

RemoteBrakes uses the same transformer structure of Brakes, hence no change is required on the transformer part of the middleware. The Transformer generates state capturing and state reestablishing code blocks in the application code. To be able to generate useful code blocks, the content of method's call stack should be obtained before the invoke-instruction execution.

RemoteBrakes has a revised thread Serialization mechanism as follows:

- `public void yield()`. This method is the same as that of Brakes. It is called whenever the current executing task wants to be suspended and as a result its execution state must be is captured in a serializable format. Please note that, this method is invoked indirectly on this task. In addition, this task serialization mechanism requires that state capturing must be initiated within the execution of the task which is going to be migrated. After state capturing is finished, a migratorcan migrate its serialized task to the target location.

- `public void resumeRmi( task )`. This method is called at the target location after it receives the migrating thread. Invocation of this method on a `Task` object, results in the update of the context of the current task object with incoming context information, after which the task is rescheduled by the task scheduler. When the task scheduler restarts the task (by calling `start()` method on it), the execution state of the task is first reestablished with the to the resumed execution of migrated task.

To be able to update contexts of tasks, a new method `contextSwitch()` is implemented on Context object. With this method, all information held on a context object is updated with new Context object variables.

In addition, task starter and scheduler methods are also updated to be able to achieve our goal at RemoteBrakes.

It is possible to use RemoteBrakes as it is in Brakes, by just calling Resume() on a suspended task, but of course without context-switching feature.

5.3 Migration over different JVMs

To be able to migrate threads between different JVMs, the applications should be implemented on an Object Request Broker architecture. By using Remote Method Invocation (RMI) of Java, an object running in one Java virtual machine can invoke methods on an object running in another Java virtual machine.

If an application integrates RemoteBrakes and RMI, then threads can be safely migrated within different JVMs without modifiying the JVMs or the communication structure of RMI.

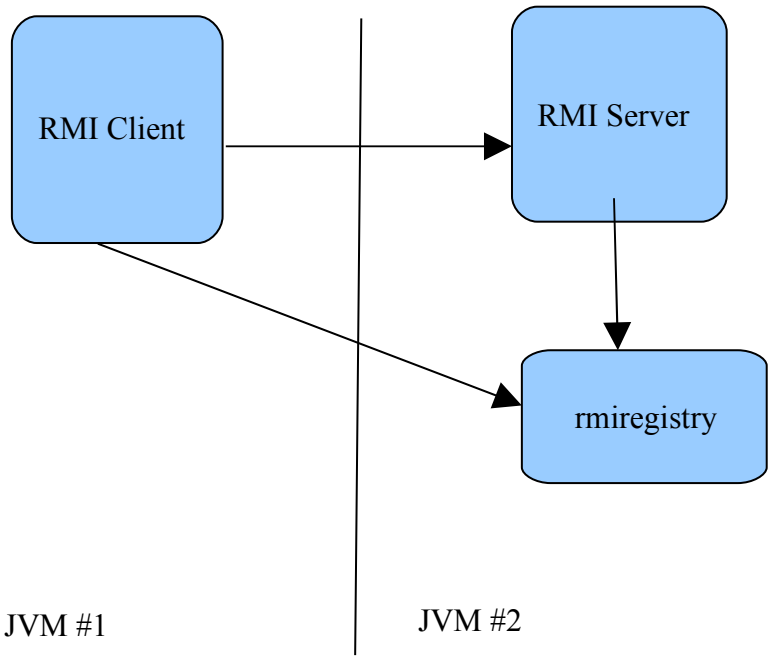


Figure 5.3 : RMI illustration

The illustration on Figure 5.3 depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it[Url-8].

5.3 Applications developed with RemoteBrakes&RMI

In this work, two different applications are developed with RemoteBrakes&RMI integration.

In this first application a distributed matrix calculation is demonstrated. The processing is carried out by three threads and one of the threads migrates in middle of its job and continues its execution at the new JVM within a different task object.

The second application is a Bouncing Balls application, where the observer can follow the movement of the bouncing balls. In this application, migration is visual and after one of the balls migrates to the other JVM with its task, one can track the path of the migrator ball.

5.3.1 Distributed Matrix Multiplication

RemoteBrakes and RMI integration is implemented in a distributed Matrix Multiplication application. In this application both strong thread migration and task context-switching is going to be demonstrated.

This application is packed with name Matrix. Classes/Interfaces of Matrix package:

- MatMulti: Performs matrix multiplication. This class implements java.io.serializable and java.lang.runnable
- MatMultiMain: Performs distribution of matrix multiplication into different threads. The tasks associated to multiplication threads are created and scheduled here.
- MatMultiMainServer: Remote interface specifies the methods that can be invoked remotely by a client.
- MatMultiMainServerImpl: Remote object implementing MatMultiMainServer interface. Implementation of method that can be called by remote client takes place here.
- MatMultiServer: RMI server. Creates a remote object and rebinds it to the rmiregistry
- Main: Main class of the application and also the RMI client object that looks for the rebinded name of remote object on RMI server. The migrator thread will be sent with this rebinded name.

In the RMI server application, there should be available RemoteBrakes tasks. It does not matter if they are running or suspended but the main issue is that the TaskScheduler of the RemoteBrakes should be active so that we can reestablish and reschledule the migrator thread.

In the DistributedMatrix Multiplication application, we are going to demonstrate the context switching of tasks so there is already a task related to a MatMulti thread.

5.3.1.1 Class Diagrams

MatMultiMain object distributes the matrix multiplication. It creates many MatMulti threads and their associated RemoteBrakes tasks. The initialization and scheduling of tasks are performed by startRunning() method. This class also provides methods for MatMulti thread suspension, reestablishment and migraton.

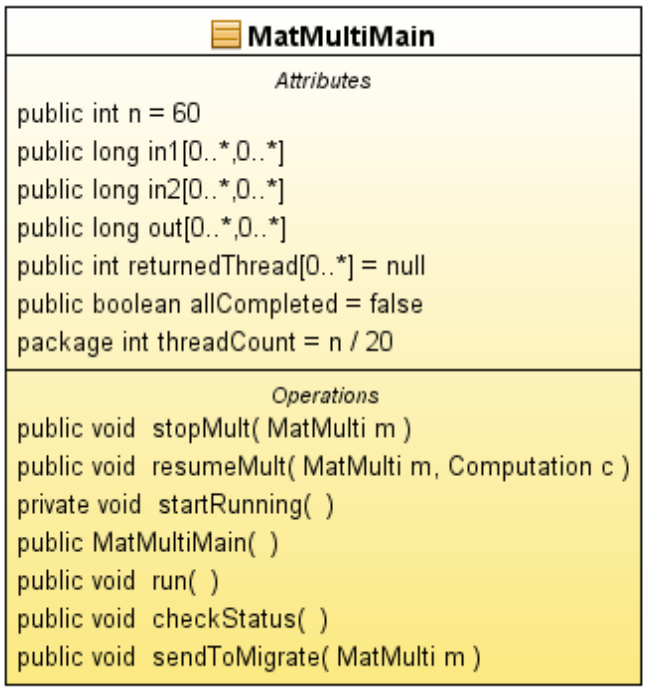


Figure 5.4 : MatMultiMain class definition

MatMulti objects constructor defines the rows and columns to be multiplied. There is boolean variable `wait`, which describes if the thread is going to be suspended or not. When the `wait` variable is set to true, the multiplication thread is suspended and becomes ready to be migrated.

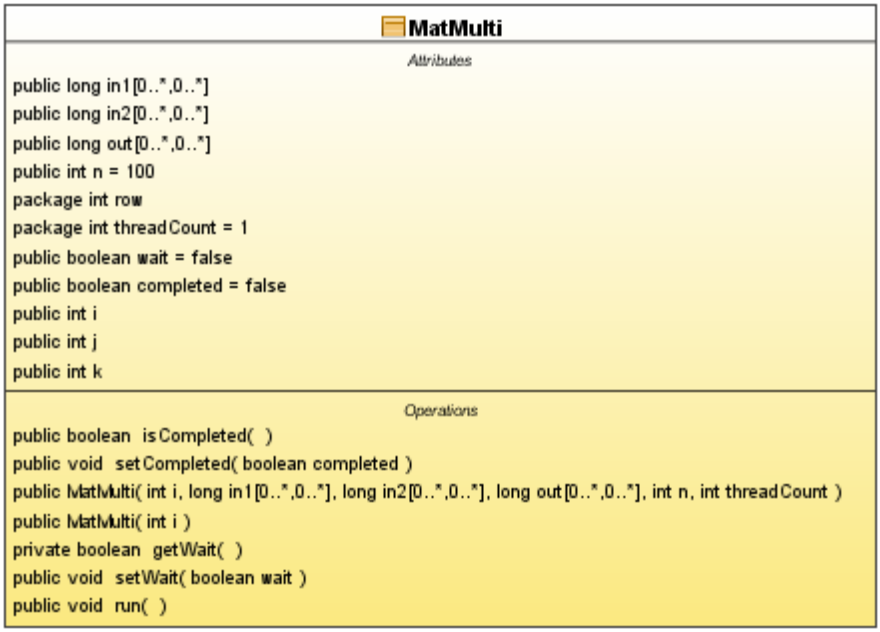


Figure 5.5 : MatMulti class definition

RMI remote object's (MatMultiMainServerImpl) constructor starts a matrix multiplication, encapsulates it with a task and schedules the task. In the method which going to be called remotely, the context of the task is exchanged with the context of the migrator task, and execution is resumed.

The remote method content is described below. `comp` is the task of current remote object and `c` is the task of the migrator task. By calling `comp.resumeRmi(c)` we are resuming the comp task with context of `c`.

```
public long[][] migrate(Computation c, MatMulti mat) throws
RemoteException {
...
/**
 * Remove flags of suspension
 */
        mat.setWait(false);
        matMulti.setWait(false);
/**
 * Current task is resuming with migrators Context
 */
        comp.resumeRmi(c);

/**
 * print out the calculated matrices
 */
      ........
 /**
 * return computed matrix values
 */
     .......
    }
```

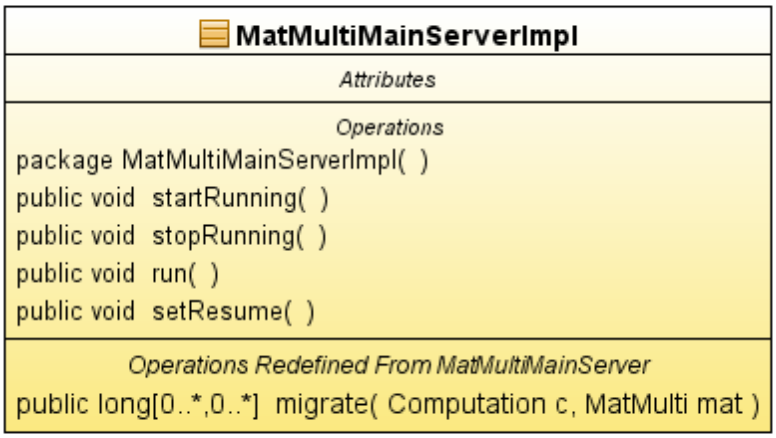


Figure 5.6 : MatMultiMainServerImpl (remote object) class definition

RMI server (MatMultiServer), creates a remote object and binds it to rmiregistry with a Name “migrator”.

```
MatMultiMainServerImpl remote = null;
System.setSecurityManager(new RMISecurityManager());
try {
  remote = new MatMultiMainServerImpl();
  try {
   Naming.rebind("migrator", remote);
  } catch (MalformedURLException ex) {
  Logger.getLogger(MatMultiServer.class.getName()).log(Le
vel.SEVERE, null, ex);
  }
 System.out.println("remote object is ready 2 b
accessed!");
 }catch (RemoteException re) {}
```

RMI Client and Main class of application is the same (Main). In the application code of this class, MatrixMultiplication is initiated and then one of the selected multiplication threads is resumed and migrated. The client looks up the remote object by its name “migrant” in the server's registry by calling

```
remoteObject = (MatMultiMainServer)
     Naming.lookup("rmi://127.0.0.1/migrant");
```

and then invokes a method on it

```
remoteObject.migrate( m1.comp[2], m1.mat[2]);
```

The main method of the class is as in Figure 5.7:

```
Public static void main(String args[]) throws
InterruptedException {

        MatMultiMain m1 = new MatMultiMain();
        m1.start();
/**
* Decision for migration
*/
        m1.sendToMigrate(m1.mat[2]);

/**
* Remote object creation
*/
        MatMultiMainServer remoteObject = null;
        try {
            remoteObject = (MatMultiMainServer)
Naming.lookup("rmi://127.0.0.1/migrator");
            out2 = remoteObject.migrate( m1.comp[2],
m1.mat[2]);
            System.out.println("MIGRATED!!!");

        } catch (Exception e) {
        }
        ;
/**
* Collect results form all threads
*/
........
/**
* Print out the results
*/
....
    }
```

Figure 5.7 : MatMultiMainServerImpl (remote object) class definition

5.3.1.2 Output

The distributed Matrix multiplication application multiplies 150*150 sized matrix with 3 threads:

6. thread1 :from row0 to 49
7. thread2: row 50 to 99
8. thread 3 from row 100 to 149

Thread 3 is suspended and migrated in the midle of its computation. The outputs at Appendix A.5 and Appendix A.6 show that the migrated thread has resumed its execution at the point exactly where it was suspended.

5.3.2 Bouncing Balls

RemoteBrakes and RMI integration is implemented in Bouncing Balls application. In his application, strong thread migration is going to be demonstrated visually.

Bouncing Balls application consists of a window and some balls which move diagonally and bounces when they hit an edge of the window. In this application, the balls are implemented by two different threads. We start the two balls at the same time, after they travel together for a while, one of them is suspended and migratedto other JVM. As its execution state is captured and reestablished at target location, we can observe that the path is following is the same as that of itsneighbour, only with a certain time delay caused by the timing overhead of migration process.

This application is packed with name BounceBall. Classes/Interfaces of BounceBall package:

- Ball: performs bouncing movement of balls. This class implements java.io.serializable and java.lang.runnable
- BounceThread: draws the window and creates balls. The tasks associated to ball threads are created and scheduled here.
- BounceThreadServer: remote interface specifies the methods that can be invoked remotely by a client.

- BounceThreadServerImpl: remote object implementing BounceThreadServer interface. Method that can be called by remote client is implemented here.
- BounceThreadServerApp: RMI server. Creates a remote object and rebinds it to the rmiregistry
- Main: Main clas of the application and also the RMI client object that looks for the rebinded name of remote object on RMI server. The migrator thread will be sent with this rebinded name.

5.3.2.1 Class Diagrams

BounceThread creates many Ball threads and their associated RemoteBrakes tasks. The initialization and scheduling of tasks are performed in its constructor code. This class also provides methods for Ball thread suspension and reestablishment.

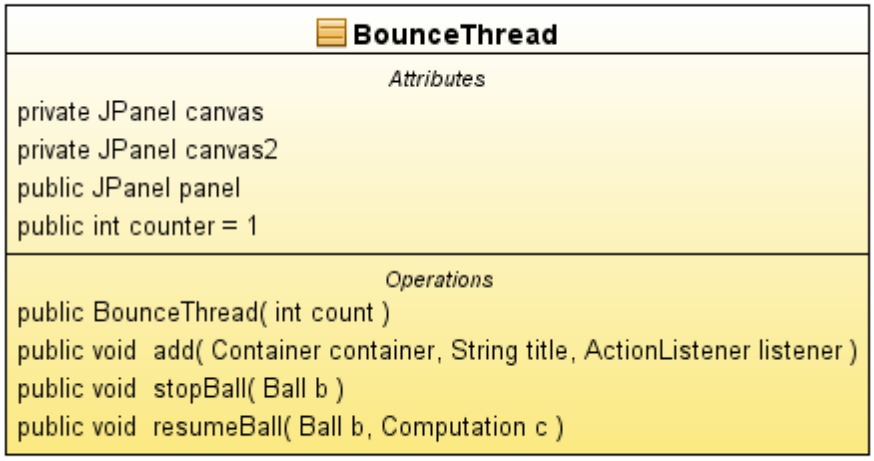


Figure 5.8 : BounceThread class definition

Ball objects create a ball on the window and Bouncing Algorithm is implemented by using draw() and move() methods. There is boolean variable `wait` , which describes if the thread is going to be suspended or not. When the `wait` variable is set to true, the ball thread is suspended and becomes ready to be migrated.

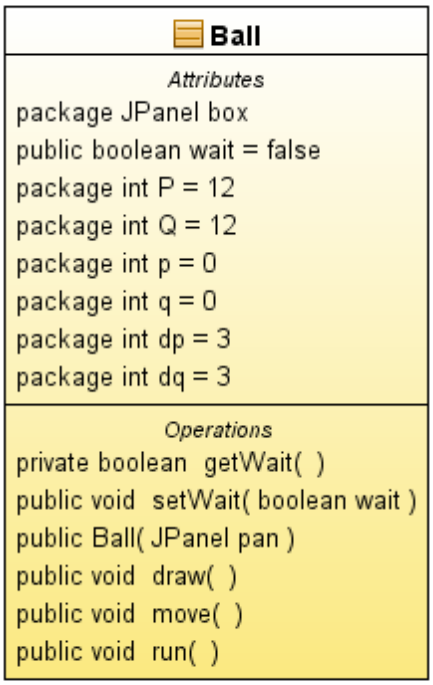


Figure 5.9 : Ball class definition

Since the Bouncing Balls application uses an additional window on which balls are moving we need to also define the window frame. In addition, in order to be able to run a task, RemoteBrakes should be initiated, at least scheduler should be started.

In RMI server(BounceThreadServerApp), we are creating the environment to be ready for a ball to move after creating a remote object and binding it to rmiregistry with Name “ball”.

```
        BounceThreadServerImpl remote = null;
        System.setSecurityManager(new RMISecurityManager());
        try {
            remote = new BounceThreadServerImpl();
            try {
                Naming.rebind("ball", remote);

            } catch (MalformedURLException ex) {
                Logger.getLogger(BounceThreadServerApp.class.g
etName()).log(Level.SEVERE, null, ex);
            }
            LocateRegistry.createRegistry(1099);
            System.out.println("remote object is ready 2 b
accessed!");
        } catch (RemoteException re) {
        }

/**
* creates the environment for a ball to move
*/
......
    }
```

The remote method content on RMI remote object (BounceThreadServerImpl) is described below.

```
public void Migrate(Ball ball,  c) throws RemoteException {
/**
* Migrated Ball is going to use same box(window)
*/
        ball.box = bounce.ball[0].box;
/**
* Resume the migrant thread with its own context
*/
        bounce.resumeBall(ball, c);}
```

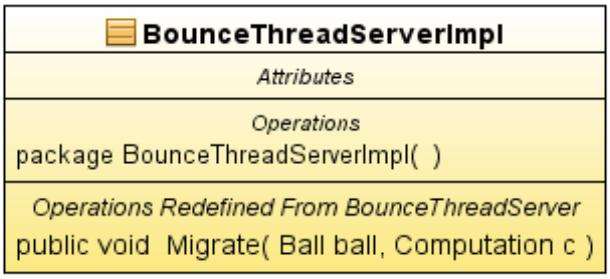


Figure 5.10 : BounceThreadServerImpl class definition

RMI Client and Main class of application is same (Main). In the application code of this class, BounceThread initiated and then one of the selected ball threads is resumed and migrated. The client looks up the remote object by its name “ball” in the server's registry by calling

```
remoteObject = (BounceThreadServer)
     Naming.lookup("rmi://127.0.0.1/ball");
```

and then invokes a method on it

```
remoteObject.Migrate(bt.ball[0],  bt.comp[0]);
```

The main method of the class is as follows:

```
public static void main(String[] args)  {

/**

* BounceThread creation, means there will be two balls running

*/

BounceThread bt = new BounceThread(2);

JFrame frame = bt;

frame.show();


/**

* Decision process for choosing the migrant ball

*/

bt.stopBall(bt.ball[0]);

BounceThreadServer remoteObject = null;

try {

remoteObject = (BounceThreadServer)
Naming.lookup("rmi://127.0.0.1/ball");

remoteObject.Migrate(bt.ball[0],
(be.ac.kuleuven.cs.ttm.computation.Computation) bt.comp[0]);

} catch (Exception e) {}

}
```

5.3.2.2 Output

The output of the Bouncing Balls application is illustrated ast Figure 5.11 and Figure 5.12.

Figure 5.11 shows the point that the ball is suspended, and thethin line gives the expected path ofthe ball.

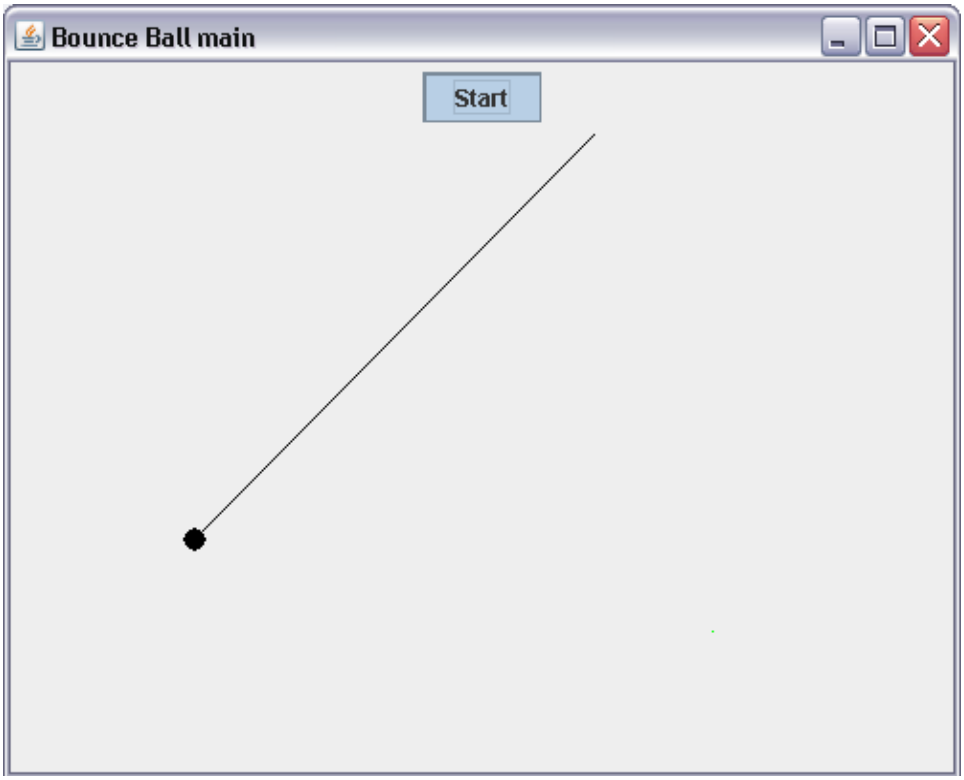


Figure 5.11 : The suspended ball and its expected path

Figure 5.12 shows the resuming point of the ball at server (which is exactle the same position it suspended at main application) and the new position of the ball aftersuspension, which is following the original path.

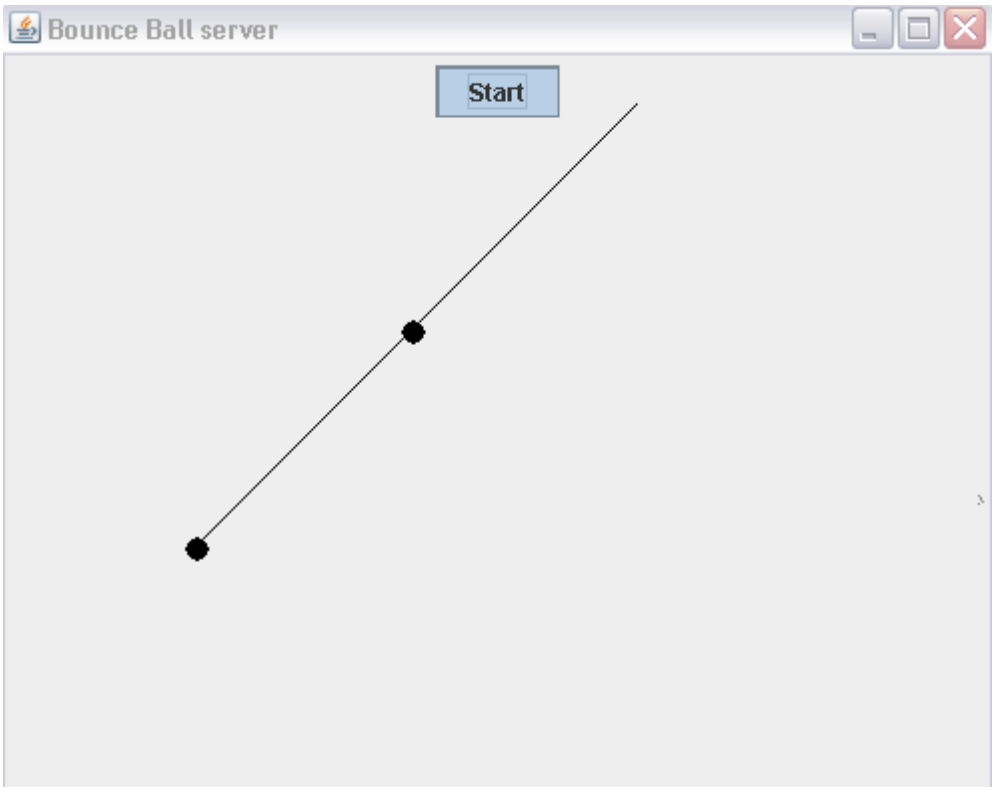


Figure 5.12 : The reestablished ball and its path

5.3.3 Analysis on Result of Implementations

With the implemented applications we can see that the execution state of the migrant thread is captured and reestablished correctly. It is also proved in Distributed Matrix Multiplication Application that we can exchange the contexts of tasks.

This successful results has a space overhead. The size of the .class files increased up to 11% to 20%

6.APPLICATION DEVELOPMENT BY USING REMOTEBRAKES

6.1 PreRequsities

RemoteBrakes is able to migrate the threads if they are implemented by using Runnable interface. Also, the threads should implement Serializable interface since they are going to be serialized.

```
public class Ball implements Runnable,Serializable {
....
}
```

6.2 Config file

There is configuration file of RemoteBrakes, which is inherited from Brakes. This file has always to be named as *rewriteinfo.prop* and should located at the directory of where you start your BRAKES-enabled application.

The content of configuration file is described in Figure 6.1

```
## This is the byte code transformer Config file
## The following entries are _MANDATORY_

# The Computation Class offers two static methods
COMPUTATIONCLASS = be.ac.kuleuven.cs.ttm.computation.Computation

# 1st static method: Switching -> true when the thread is breaking
up
SWITCHING = isSwitching

# 2nd static method: Restoring -> true when the thread is being
rebuilt
RESTORING = isRestoring

# The Context Class offers two ranges of static methods
CONTEXTCLASS = be.ac.kuleuven.cs.ttm.computation.Context

# 1ste range: the Push methods : one for every basic type
#                        one for "Object"
#                        one for "This"
PUSHMETHOD = curPush

# 2nd range: the Pop methods : one for every basic type
#                        one for "Object"
#                        one for "This"
POPMETHOD = curPop
```

Figure 6.1 : Configuration file of Brakes framework

In fact, in order to use RemoteBrakes this configuration above is enough. But if the developers need to extend RemoteBrakes, they have to consider about those mandatory entries.

6.3 Transformer

Transformer class analyses the class file and inserts additional bytecodes into the .class file in order to provide capturing and reestablishing operation. Therefore, all .class files should be rewritten by using `Transformer` object of RemoteBrakes.

By typing on comand prompt

```
java be.ac.kuleuven.cs.ttm.transformer.Transformer  Ball.class
```

`Ball.class.rewritten` is created, and this file should be renamed as `Ball.class`

By using a decompiler we can see the additional codes added by `Transformer`. When we analyse the decompiled code, we can see that at the beginning of each method, it is checked if it is re-establishing its state by checking the `isRestoring` flag as described in Figure 6.1. And after each method invocation, it is check if the task is suspending by checking `isSwitching` flag as described in Figure 6.1. The application code flow is controlled by labels.

The decompiled java code after the transformation can be found at Appendix A.7.

6.4 Tasks

Each runnable and serializable thread must be run in a `Computation` object, known as the task described in the preivous sections. Each `Computation` object should be created by using a `Factory` object both in Brakes and RemoteBrakes..

```
Ball ball1 = new Counter();
Ball ball22 = new Counter();

Computation comp1 =
     Factory.getInstance().createComputation(ball11);
Computation comp2 =
     Factory.getInstance().createComputation(ball22);
```

In RemoteBrakes, the Counter objects are going to be run by using the Computation objects.

```
comp1.startRmi();
comp2.startRmi();
```

Each computation object should be scheduled to be able to run.

```
Scheduler.getInstance().start();
```

The last instruction (Scheduler.getInstance().start()) never stops. A running computation can stop itself with the command.

```
Scheduler.getInstance().currentComputation().yield(true)
```

In RemoteBrakes, the originating thread can reschedule the paused computation with

```
comp1.resumeRmi(comp1)
```

or

```
comp1.resumeRmi(comp2)
```

The first resume operation, resumes the suspended task but the second resume operation resumes with context of comp2. Both usage is possible, developer should decide which one to use according to applciation requirements.

7. CONCLUSION AND RECOMMENDATIONS

In this thesis, strong migration of Java threads between different Java Virtual Machines is implemented with RemoteBrakes. In RemoteBrakes we can also switch execution status of threads.

During the preperation of this thesis, a lot of time has been spent on investigating the current Thread Migration approaches and there has been some communication with the developers of the current projects. At the end of this process, we decided to use Brakes system as a base. Brakes is a composition of a bytecode transformer and a framework, some projects updated the bytecode transformer of Brakes but we have only modified the framework. Our extension is named as *RemoteBrakes* and it supports migration between different JVMS and context-switching between tasks.

By integrating RMI and RemoteBrakes, it is possible to use migration in between many JVMs on network in a distributed application.

Our intention is the give an API which enables developers to use strong thread migration in their projects. That's why the last section describes the usage of the RemoteBrakes system. This can be used as a guide to develop new applciations with RemoteBrakes.

Current work includes two different applications, both of them are proof-of-concept projects, it is possible to add a decision mechanism to decide start of migration (as a result of CPU overload, issue of a resource sharing, etc.)

I believe this work will be helpful to whom need to migrate threads strongly in their projects. It is possible that Java can release an API for thread migration in the future, it depends on the requests of vendors.

REFERENCES

[1] **Bouchenak, S., Hagimont, D., Krakowiak, et al.** 2002: Experiences Implementing Efficient Java Thread Serialziation, Mobility and Persistence, INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

[2] **Truyen, E., Robben, B.,Vanhaute, B., et al.** 2000 : Portable support for transparent Thread migration in Java. In Procedings of ASA/MA2000

[3] **Coninx, T., Truyen, E.,Vanhaute, B., et al.** 2000 : On the use of threads in mobile object systems . InWorkshops of ECOOP 2000

[4] **Sakamoto, T., Sekiguchi, T., and Yonezawa, A.** 2000 : Bytecode transformation for portable Java Thread Migration in Java. In Procedings of ASA/MA2000

[5] **Fünfrocken,S.** 1998: Transparent Migration of Java-based Mobile Agents**.**. In *MA'98 Mobile Agents*, volume 1477 of LNCS,pages 26-37.Springer-Verlag

[6] **Acharya, A., Ranganathan, M., and Saltz, J.** 1996 : Sumatra: A Language for resource-aware mobile programs . In 2nd International Workshop on Mobile Object Systes (MOS'96)

[7] **Suezawa,T.** 2000 : Persistent Execution State of a Java Virtual Machine. In ACM Java Grande 2000 Conference,CA, USA.

[8] **Illman, T., Krueger, T., Kargl, F., et al..** 2001 : Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture. In 5th IEEE Int. Conference on Mobile Agents(MA'2001), USA.

[9] **Sekiguchi, T., Masuhara, H., and Yonezawa A.** 2000 : A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In 3rd International Conference on Coordination Models and Languages, Amsterdam.

[10] **Bouchenak, S., Hagimont, D.** 2002 : Zero Overhead Java Thread Migration . Techical Report INRA

[11] **Lopes, R. F., Silva F. J. D. A.** 2005 : Strong Migration in a Grid Based on Mobile Agents

[12] **Weyns, D., Truyen, E., and Verbaeten, P.** 2002 : Serialization of Distributed Execution-State in Java.

[13] **Dahm, M.** 1999 : Byte Code Engineering with the JavaClass API. Technical Report

Url-1 <*http://www.cs.kuleuven.ac.be/~eddy/BRAKES/brakes.html*>, accessed at 05.05.2009.

Url-2 <*http://jakarta.apache.org/bcel/manual.html*>, accessed at 06.07.2009.

Url-3 <*http://jcp.org/en/jsr/detail?id=323*>, accessed at 05.12.2009.

Url-4 <*http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html* >, accessed at 03.09.2009.

Url-5 <*http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/*>, accessed at 01.12.2009.

Url-6 <*http://www.exciton.cs.rice.edu/JAvaResources/DesignPatterns/VisitorPattern.htm*>, accessed at 11.12.2009.

Url-7 <*http://jakarta.apache.org/bcel/projects.html*>, accessed at 17.11.2009.

Url-8 <*http://java.sun.com/docs/books/tutorial/rmi/overview.html* >, accessed at 01.10.2009.

APPENDICES

APPENDIX A.1 JavaClass API UML Diagram

APPENDIX A.2 ConstantPool API UML Diagram

APPENDIX A.3 ClassGen API UML Diagram

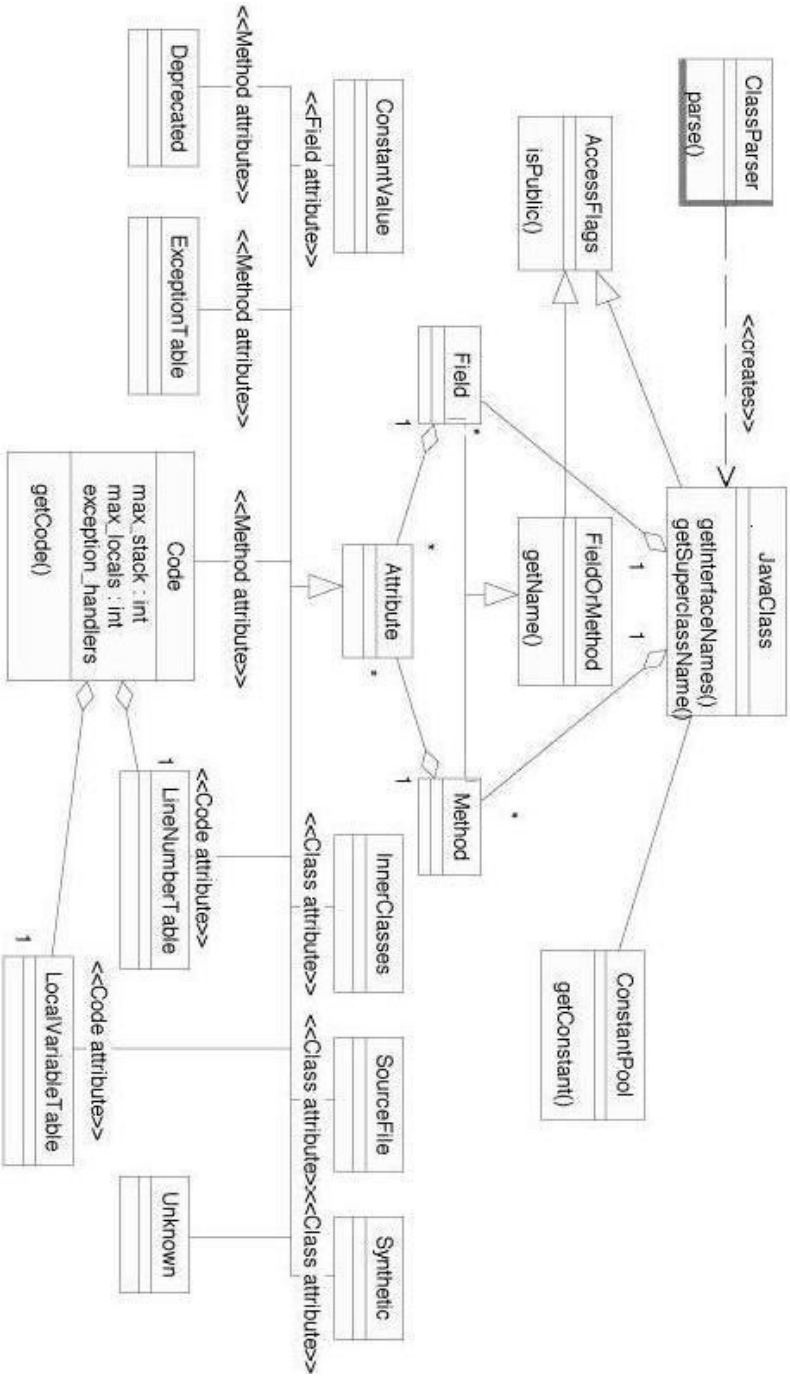
APPENDIX A.4 Instruction API UML Diagram

APPENDIX A.5 Server side output of Matrix Multiplication application

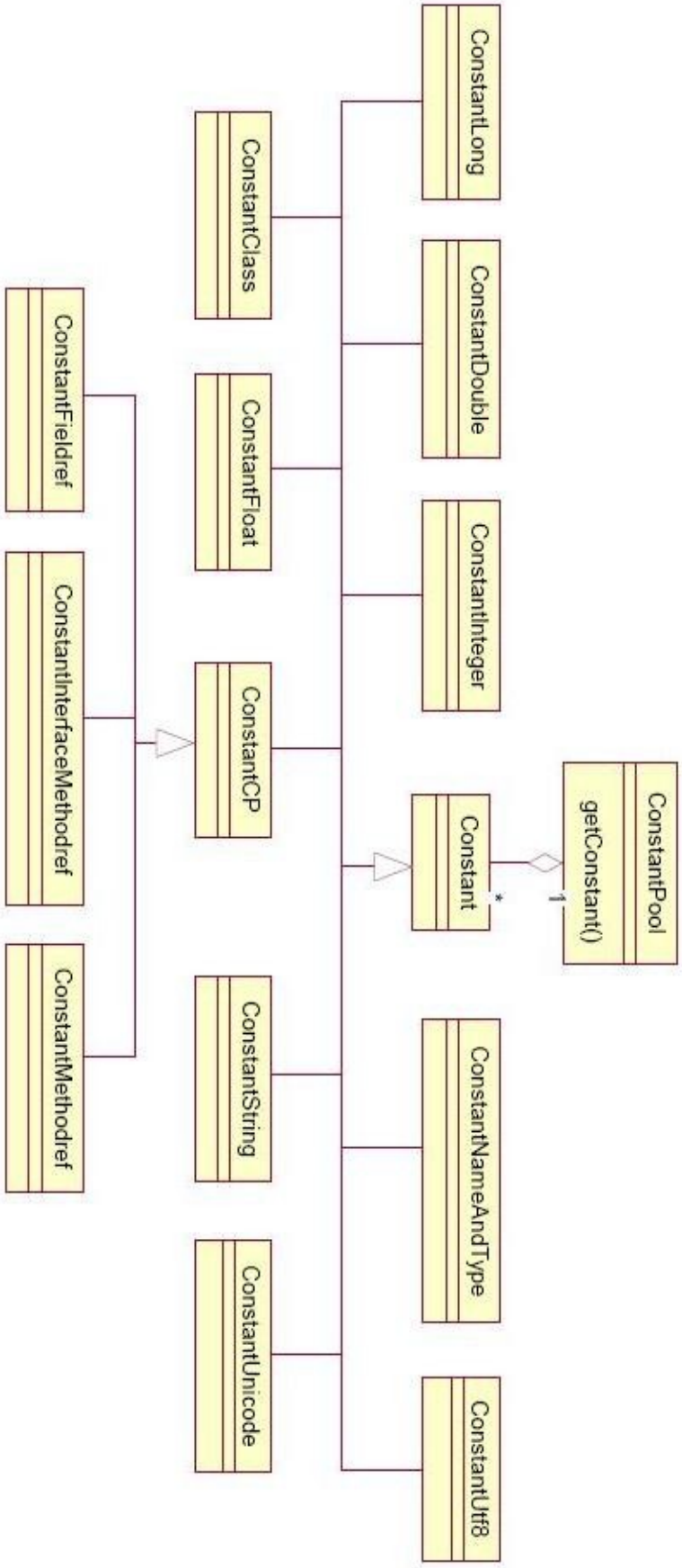
APPENDIX A.6 Client Side Output of Matrix Multiplication Application

APPENDIX A.7 Transformer updates

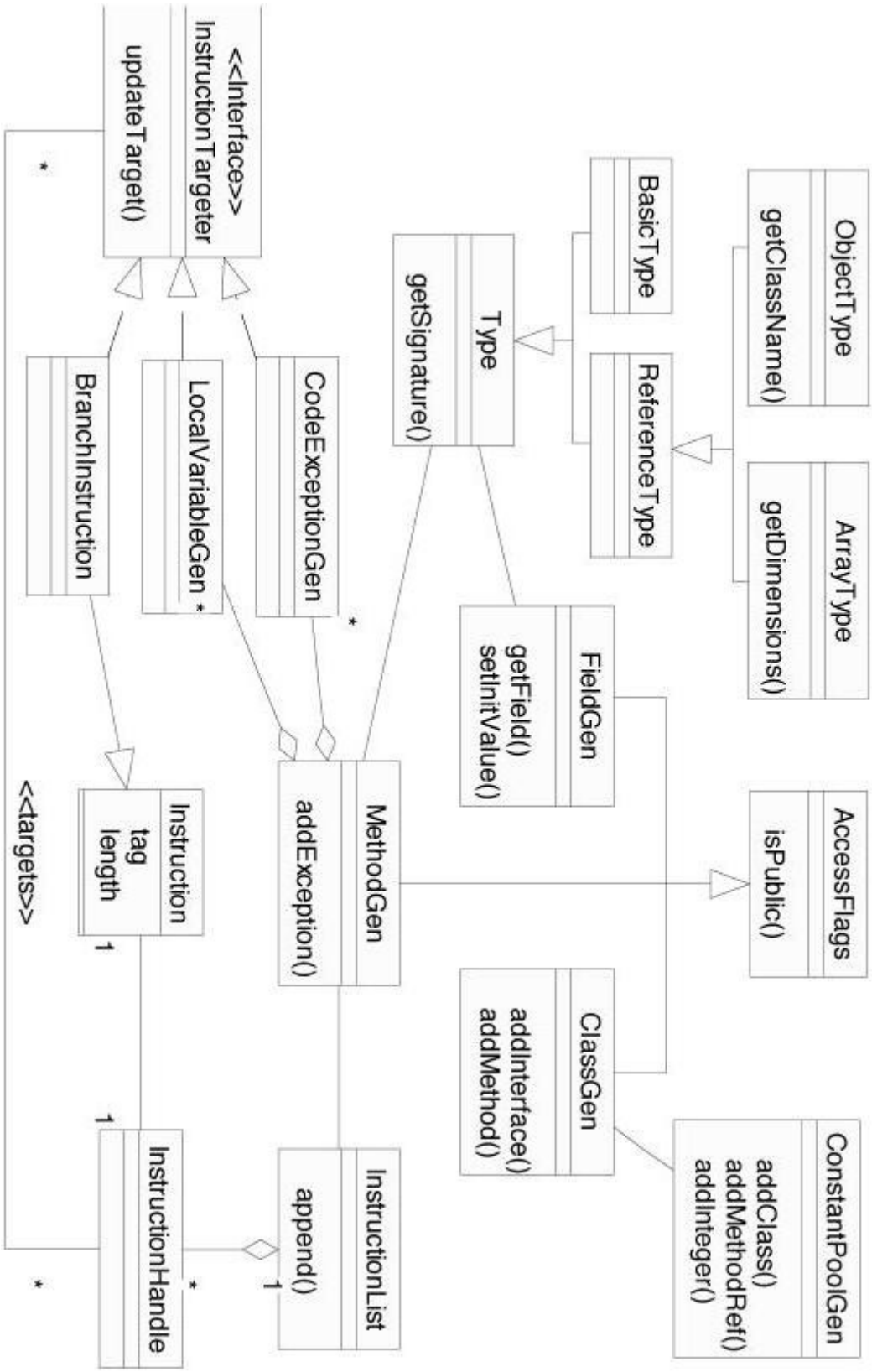
APPENDIX A.1 JavaClass API UML Diagram



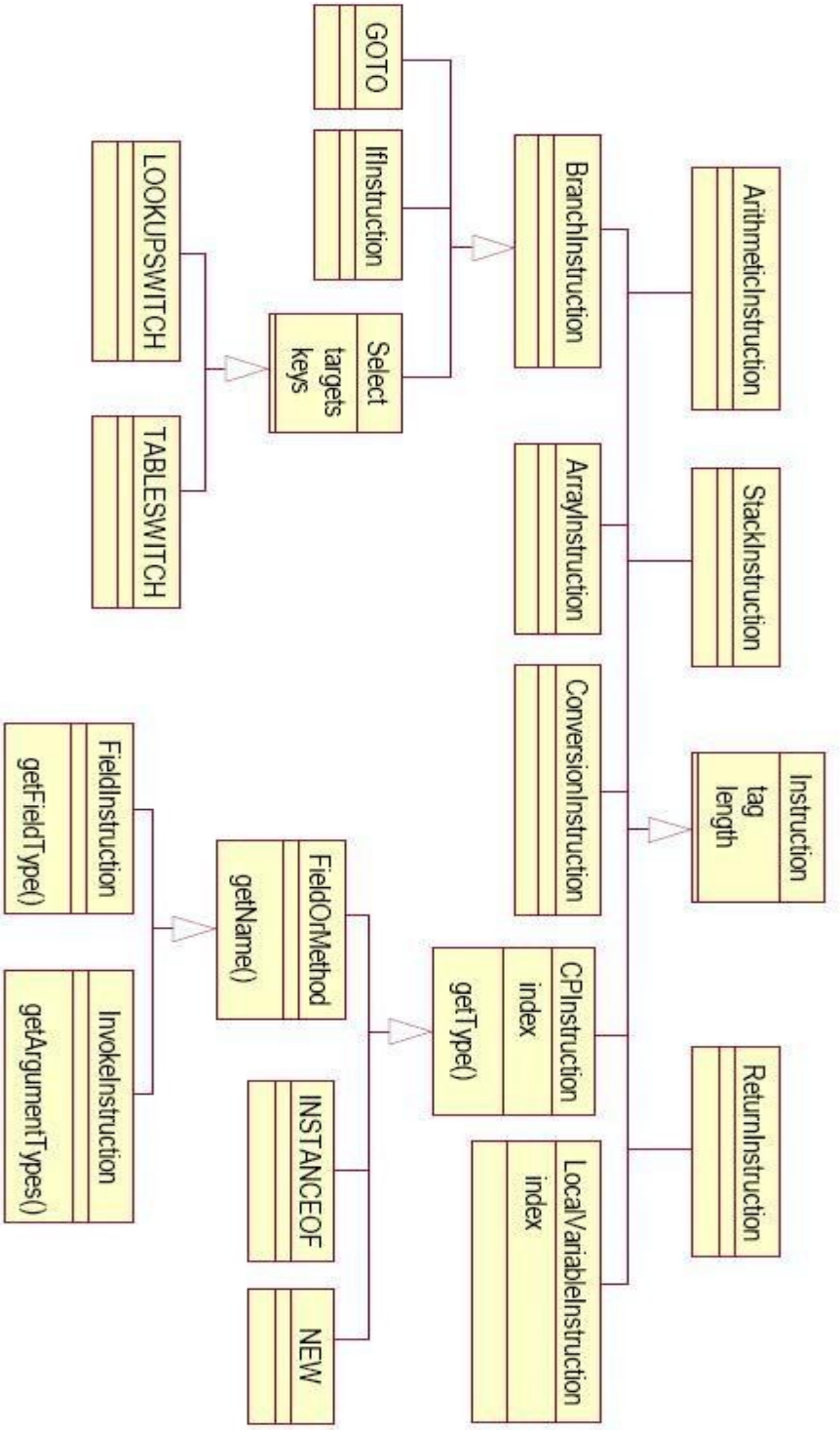
APPENDIX A.2 ConstantPool API UML Diagram



APPENDIX A.3 ClassGen API UML Diagram



APPENDIX A.4 Instruction API UML Diagram



APPENDIX A.5 Server side output of Matrix Multiplication application

```
Scheduler.schedule() Thread[Thread-1,5,main]
Thread[Thread-1,5,main] scheduled
------------------------------------------------------------
----- Remote object is ready --------------------------------
------------------------------------------------------------
remote object is ready 2 b accessed!
MatMultiMainServerImpl[UnicastServerRef [liveRef: [endpoint:
[127.0.0.1:1078](local),objID:[68f6196b:1254f6a2471:-7fff,
-6098002020478199434]]]]
------------------------------------------------------------
----- Initial matrix multiplication is initiated and task is
scheduled-----------------------------------------------------
------------------------------------------------------------
Scheduler: scheduling: Thread[Thread-2,5,main]
Computation.schedule: firstTime = true
thread Thread[Thread-2,5,main] registered -> Context =
be.ac.kuleuven.cs.ttm.computation.Context@affc70
running
stopped
thread Thread[Thread-2,5,main] deregistered
Computation.ended
------------------------------------------------------------
----initial task is suspended---------------------------------
------------------------------------------------------------
------------------------------------------------------------
----remote method invoked, task is being rescheduled----------
------------------------------------------------------------


Scheduler.schedule() RMI Thread[Thread-4,5,RMI Runtime]
ResumeRMI Thread[Thread-4,5,RMI Runtime]Thread[Thread-4,5,RMI
Runtime]
Scheduler: scheduling RMI: Thread[Thread-5,5,RMI Runtime]
Computation.schedule: firstTime = false
------------------------------------------------------------
----context switching-----------------------------------------
------------------------------------------------------------
```

```
tstack (Context
be.ac.kuleuven.cs.ttm.computation.Context@1fc4bec is size : 4

tstack (Context
be.ac.kuleuven.cs.ttm.computation.Context@1fc4bec is size : 3

Thread[Thread-4,5,RMI Runtime]

thread Thread[Thread-5,5,RMI Runtime] registered -> Context =
be.ac.kuleuven.cs.ttm.computation.Context@1fc4bec

running

istack (Context
be.ac.kuleuven.cs.ttm.computation.Context@1fc4bec is size : 2

astack (Context
be.ac.kuleuven.cs.ttm.computation.Context@1fc4bec is size : 2

tstack (Context
be.ac.kuleuven.cs.ttm.computation.Context@1fc4bec is size : 2

istack (Context
be.ac.kuleuven.cs.ttm.computation.Context@1fc4bec is size : 1

astack (Context
be.ac.kuleuven.cs.ttm.computation.Context@1fc4bec is size : 1

tstack (Context
be.ac.kuleuven.cs.ttm.computation.Context@1fc4bec is size : 1
----------------------------------------------------------------

----matrix multiplication starts from row: 100 coloumn: 47 and
calculating in1[100][83] * in2[83][47] ----------------------

----------------------------------------------------------------

Computation.resumed RMI
be.ac.kuleuven.cs.ttm.computation.Context@1fc4bec

k: 100 i: 47 j: 83

k: 100 i: 47 j: 84

k: 100 i: 47 j: 85

.

.

.

k: 149 i: 149 j: 148

k: 149 i: 149 j: 149

COMPLETED from 100 to 149

OUTPUT from migrate :

stopped

thread Thread[Thread-5,5,RMI Runtime] deregistered

Computation.ended
```

OUTPUT from migrate line :100

900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900

OUTPUT from migrate line :101

900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900

APPENDIX A.6 Client Side Output of Matrix Multiplication Application

```
-------------------------------------------------------------
-- thread – task initiation and scheduling-------------------
-------------------------------------------------------------
Scheduler.schedule() RMI Thread[Thread-1,5,main]
Thread[Thread-1,5,main] scheduled RMI
Scheduler.schedule() RMI Thread[Thread-3,5,main]
Thread[Thread-3,5,main] scheduled RMI
Scheduler.schedule() RMI Thread[Thread-5,5,main]
Thread[Thread-5,5,main] scheduled RMI
Scheduler: scheduling RMI: Thread[Thread-2,5,main]
Scheduler: scheduling RMI: Thread[Thread-4,5,main]
Scheduler: scheduling RMI: Thread[Thread-6,5,main]
Computation.schedule: firstTime = true
Computation.schedule: firstTime = true
Computation.schedule: firstTime = true
thread Thread[Thread-6,5,main] registered -> Context =
be.ac.kuleuven.cs.ttm.computation.Context@e09713
running
Started from 100 to 149
k: 100 i: 0 j: 0
k: 100 i: 0 j: 1
k: 100 i: 0 j: 2
k: 100 i: 0 j: 3
thread Thread[Thread-2,5,main] registered -> Context =
be.ac.kuleuven.cs.ttm.computation.Context@19b49e6
running
Started from 0 to 49
k: 0 i: 0 j: 0
k: 0 i: 0 j: 1


.
.
thread Thread[Thread-4,5,main] registered -> Context =
be.ac.kuleuven.cs.ttm.computation.Context@156ee8e
running
Started from 50 to 99
k: 50 i: 0 j: 0
```

```
k: 50 i: 0 j: 1


.
.
.
-----------------------------------------------------------------
-- thread3has ben suspended at row: 100 coloumn: 47 and
calculation in1[100][83] * in2[83][47]
-----------------------------------------------------------------
Wait has been set: k:100i: 47j: 83
Computation.block RMI
.
.
COMPLETED from 50 to 99
stopped
thread Thread[Thread-4,5,main] deregistered
Computation.ended
k: 2 i: 86 j: 133
k: 2 i: 86 j: 134
.
.
k: 49 i: 149 j: 149
COMPLETED from 0 to 49
stopped
thread Thread[Thread-2,5,main] deregistered
Computation.ended
.
OUTPUT line :99
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
```

```
-----------------------------------------------------------------

-- since the thread is suspended at line 100, the following
calculation is not done on client side------------------------
-----------------------------------------------------------------


OUTPUT line :100

900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 900 900 900 900 900 900 900 900 900 900 900 900 900
900 900 498 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0


OUTPUT line :101

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

APPENDIX A.7 Transformer updates

Original Ball.java:

```
class Ball implements Runnable, Serializable {

    JPanel box;
    public boolean wait = false;

    private boolean getWait() {
        if (wait) {
            Scheduler.getInstance().currentComputation().yield
Rmi(true);
        }
        return wait;
    }

    public void setWait(boolean wait) {
        this.wait = wait;
    }
    int P = 12, Q = 12, p = 0, q = 0, dp = 3, dq = 3;

    public Ball(JPanel pan) {
        box = pan;
    }

    public void draw() {
        Graphics g = box.getGraphics();
        g.fillOval(p, q, P, Q);
        g.dispose();
    }

    public void move() {
        if (!box.isVisible()) {
            return;
        }
        Graphics g = box.getGraphics();
        g.setXORMode(box.getBackground());
        g.fillOval(p, q, P, Q);
```

```
            p += dp;
            q += dq;
            Dimension d = box.getSize();
            if (p < 0) {
                p = 0;
                dp = -dp;
            }
            if (p + P >= d.width) {
                p = d.width - P;
                dp = -dp;
            }
            if (q < 0) {
                q = 0;
                dq = -dq;
            }
            if (q + Q >= d.height) {
                q = d.height - Q;
                dq = -dq;
            }
            g.fillOval(p, q, P, Q);
            g.dispose();
        }

        public void run() {
            try {
                draw();
                while (!getWait()) {
                    move();
                    Thread.sleep(15);
                }
            } catch (InterruptedException e) {
            }
        }
```

Decompiled Ball.class.rewritten:

```
class Ball
    implements Runnable, Serializable
{
```

```
    private boolean getWait()
    {
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isRestoring())
goto _L2; else goto _L1
_L1:
        Context.curPopInt();
        JVM INSTR tableswitch 0 2: default 79
    //                   0 36
    //                   1 46
    //                   2 62;
           goto _L2 _L3 _L4 _L5
_L3:
        this = (Ball)Context.curPopObject();
          goto _L6
_L4:
        this = (Ball)Context.curPopObject();
        (Scheduler)Context.curPopThis();
          goto _L7
_L5:
        this = (Ball)Context.curPopObject();
        (Computation)Context.curPopThis();
        false;
          goto _L8
_L2:
        if(!wait) goto _L9; else goto _L6
_L6:
        Scheduler.getInstance();
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isSwitching())
goto _L7; else goto _L10
_L10:
        JVM INSTR pop ;
        Context.curPushThis(this);
        Context.curPushObject(this);
        Context.curPushInt(0);
        return false;
_L7:
        currentComputation();
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isSwitching())
goto _L12; else goto _L11
_L11:
        JVM INSTR pop ;
        Context.curPushThis(this);
        Context.curPushObject(this);
        Context.curPushInt(1);
        return false;
_L12:
        true;
_L8:
        yieldRmi();
        if(be.ac.kuleuven.cs.ttm.computation.Computation.isSwi
tching())
        {
```

```
            Context.curPushThis(this);
            Context.curPushObject(this);
            Context.curPushInt(2);
            return false;
        }
_L9:
        return wait;
    }

    public void setWait(boolean flag)
    {
        wait = flag;
    }

    public Ball(JPanel jpanel)
    {
        wait = false;
        P = 12;
        Q = 12;
        p = 0;
        q = 0;
        dp = 3;
        dq = 3;
        box = jpanel;
    }

    public void draw()
    {
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isRestoring())
goto _L2; else goto _L1
_L1:
        Context.curPopInt();
        JVM INSTR tableswitch 0 0: default 44
    //                    0 28;
           goto _L2 _L3
_L3:
        this = (Ball)Context.curPopObject();
        (JPanel)Context.curPopThis();
          goto _L4
_L2:
        box;
_L4:
        getGraphics();
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isSwitching())
goto _L6; else goto _L5
_L5:
        JVM INSTR pop ;
        Context.curPushThis(this);
        Context.curPushObject(this);
        Context.curPushInt(0);
        false;
        return;
_L6:
        Graphics g;
```

```
        g;
        g.fillOval(p, q, P, Q);
        g.dispose();
        return;
    }

    public void move()
    {
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isRestoring())
goto _L2; else goto _L1
_L1:
        Context.curPopInt();
        JVM INSTR tableswitch 0 3: default 124

           goto _L2 _L3 _L4 _L5 _L6
_L3:
        this = (Ball)Context.curPopObject();
        (JPanel)Context.curPopThis();
          goto _L7
_L4:
        this = (Ball)Context.curPopObject();
        (JPanel)Context.curPopThis();
          goto _L8
_L5:
        Graphics g;
        g = (Graphics)Context.curPopObject();
        this = (Ball)Context.curPopObject();
        (Graphics)Context.curPopObject();
        (JPanel)Context.curPopThis();
          goto _L9
_L6:
        g = (Graphics)Context.curPopObject();
        this = (Ball)Context.curPopObject();
        (JPanel)Context.curPopThis();
          goto _L10
_L2:
        box;
_L7:
        isVisible();
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isSwitching())
goto _L12; else goto _L11
_L11:
        JVM INSTR pop ;
        Context.curPushThis(this);
        Context.curPushObject(this);
        Context.curPushInt(0);
        false;
        return;
_L12:
        JVM INSTR ifne 156;
           goto _L13 _L14
_L13:
        return;
_L14:
```

```
        box;
_L8:
        getGraphics();
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isSwitching())
goto _L16; else goto _L15
_L15:
        JVM INSTR pop ;
        Context.curPushThis(this);
        Context.curPushObject(this);
        Context.curPushInt(1);
        false;
        return;
_L16:
        g;
        g;
        box;
_L9:
        getBackground();
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isSwitching())
goto _L18; else goto _L17
_L17:
        JVM INSTR pop ;
        Context.curPushObject();
        Context.curPushThis(this);
        Context.curPushObject(this);
        Context.curPushObject(g);
        Context.curPushInt(2);
        false;
        return;
_L18:
        setXORMode();
        g.fillOval(p, q, P, Q);
        p += dp;
        q += dq;
        box;
_L10:
        getSize();
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isSwitching())
goto _L20; else goto _L19
_L19:
        JVM INSTR pop ;
        Context.curPushThis(this);
        Context.curPushObject(this);
        Context.curPushObject(g);
        Context.curPushInt(3);
        false;
        return;
_L20:
        Dimension dimension;
        dimension;
        if(p < 0)
        {
            p = 0;
```

```
            dp = -dp;
        }
        if(p + P >= dimension.width)
        {
            p = dimension.width - P;
            dp = -dp;
        }
        if(q < 0)
        {
            q = 0;
            dq = -dq;
        }
        if(q + Q >= dimension.height)
        {
            q = dimension.height - Q;
            dq = -dq;
        }
        g.fillOval(p, q, P, Q);
        g.dispose();
        return;
    }

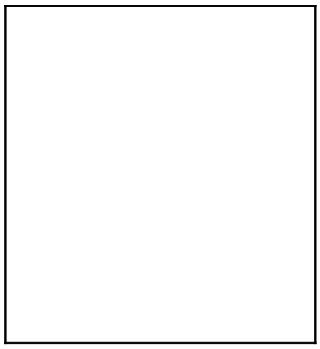
    public void run()
    {
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isRestoring())
goto _L2; else goto _L1
_L1:
        Context.curPopInt();
        JVM INSTR tableswitch 0 2: default 84

           goto _L2 _L3 _L4 _L5
_L3:
        this = (Ball)Context.curPopObject();
        (Ball)Context.curPopThis();
          goto _L6
_L4:
        this = (Ball)Context.curPopObject();
        (Ball)Context.curPopThis();
          goto _L7
_L5:
        this = (Ball)Context.curPopObject();
        (Ball)Context.curPopThis();
          goto _L8
_L2:
        this;
_L6:
        draw();
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isSwitching())
goto _L10; else goto _L9
_L9:
        Context.curPushThis(this);
        Context.curPushObject(this);
        Context.curPushInt(0);
        false;
```

```
        return;
_L10:
        this;
_L7:
        getWait();
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isSwitching())
goto _L12; else goto _L11
_L11:
        JVM INSTR pop ;
        Context.curPushThis(this);
        Context.curPushObject(this);
        Context.curPushInt(1);
        false;
        return;
_L12:
        JVM INSTR ifne 169;
           goto _L13 _L14
_L14:
        break; /* Loop/switch isn't completed */
_L13:
        this;
_L8:
        move();
        if(!
be.ac.kuleuven.cs.ttm.computation.Computation.isSwitching())
goto _L16; else goto _L15
_L15:
        Context.curPushThis(this);
        Context.curPushObject(this);
        Context.curPushInt(2);
        false;
        return;
_L16:
        Thread.sleep(15L);
        if(true) goto _L10; else goto _L17
        InterruptedException interruptedexception;
        interruptedexception;
_L17:
    }

    JPanel box;
    public boolean wait;
    int P;
    int Q;
    int p;
    int q;
    int dp;
    int dq;}
```

CURRICULUM VITA



Candidate's full name:	**Ayşe GENÇ**
Place and date of birth:	**Balıkesir 11.12.1981**
Permanent Address:	**Dikilitas mah. Emirhan cad. Kardesler sok. No:3/11 Besiktas/ISTANBUL**
Universities and Colleges attended:	**İTÜ Computer Engineering**