

Slovak University of Technology in Bratislava  
Faculty of Informatics and Information Technologies

FIIT-5212-5770

Erik Šuta  
**PERFORMANCE MONITORING OF JAVA  
APPLICATIONS**

Bachelor thesis

Degree course: Informatics  
Field of study: 9.2.1 Informatics  
Institute of Informatics and Software Engineering, FIIT STU, Bratislava  
Supervisor: Mgr. Pavol Mederly PhD.

2013, May



# ANOTÁCIA

---

Slovenská technická univerzita v Bratislave

FAKULTA INFORMATIKY A INFORMACNÝCH TECHNOLOGIÍ

Študijný program: 9.2.1 Informatika

## MONITOROVANIE VÝKONU APLIKÁCIÍ V PROSTREDÍ JAVA

Autor: Erik Šuta

Vedúci bakalárskej práce: Mgr. Pavol Mederly PhD.

Máj 2013

Výkonnosť aplikácií a ich práca so systémovými prostriedkami nepochybne patria medzi najdôležitejšie vlastnosti každého softvérového produktu. Skúmaním týchto parametrov sa zaoberá profiling, čo je oblasť softvérového inžinierstva, ktorej účelom je na základe zberu a analýzy dát o výkonnosti skúmaného programu počas jeho vykonávania dodať používateľom informácie, na základe ktorých budú schopní zlepšiť výkonnosť aplikácie. Profiling spadá pod dynamickú analýzu programu a v súčasnosti existuje viacero softvérových riešení, ktoré ponúkajú komplexnú funkcionality aj na veľmi špecifické profilovanie java aplikácií. Cieľom tejto práce však bolo implementovať profilovacie riešenie do istej miery odlišné od už hotových riešení. Odlišnosť našej implementácie spočíva v poskytnutí používateľom možnosť definovania profilovacích scenárov a možnosť nastavovať úrovne profilovania a filter volaní metód. Overenie nášho riešenia prebiehalo okrem iného aj v rámci monitorovania väčšej softvérovej aplikácie s otvoreným zdrojovým kódom.

Kľúčové slová: java, profilovanie, monitorovanie výkonnostných parametrov aplikácie, wicket, profilovacie scenáre, architektúra klient-server, java profilovací agent



# ANNOTATION

---

Slovak University of Technology Bratislava

FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Degree Course: 9.2.1 Informatics

## PERFORMANCE MONITORING OF JAVA APPLICATIONS

Author: Erik Šuta

Supervisor: Mgr. Pavol Mederly PhD.

2013, May

Performance of applications and their utilization of system resources is one of the most important properties of every software product. Research in this area is bound to profiling. Profiling is a part of software engineering. Based on collection and analysis of data about performance of analysed software solution during its execution, software developers are able to enhance the performance potential of their product, what is basically a purpose of profiling. Profiling is one of the forms of dynamic program analysis and there are many software solutions providing complex functionality to do even a very specific profiling of java applications. The goal of this bachelor thesis was to implement a software solution slightly different from current existing profiling solutions. We wanted to enable user definition of special profiling scenarios and an option to choose between various profiling levels. The definition of method call filters was also implemented. Our solution was evaluated by monitoring an extensive open-source software product.

Key words: java, profiling, application performance monitoring, wicket, profiling scenarios, client-server architecture, java profiling agent



## Statement

I declare that I have worked on this bachelor thesis on my own and that I have not used other literature than stated.

Bratislava, may 2013

.....

signature





## **Thanks,**

To Mgr. Pavol Mederly PhD., for his patience, time he invested to me and to making this thesis better and for precious and appreciated advices.



# Table of contents

---

<b>INTRODUCTION.....</b>	<b>1</b>
<b>1 PROBLEM ANALYSIS .....</b>	<b>3</b>
1.1 Java Virtual Machine Analysis .....	3
1.2 Profiling Principles Analysis .....	6
1.3 Analysing of Profiling Possibilities of Standard Java API .....	9
1.4 Analysis of Chosen Open-Source Profiling Solutions .....	13
1.5 Analysis of Commercial Profiling Solutions.....	15
<b>2 REQUIREMENTS ANALYSIS AND SPECIFICATION .....</b>	<b>17</b>
2.1 Functional Requirements .....	17
Client – server architecture .....	17
Real – time representation of CPU and memory usage of analysed application.....	18
User defined profiling scenarios .....	18
User defined profiling levels .....	19
User defined method calls filters .....	19
2.2 Non-functional Requirements .....	20
Intuitive usability and user friendly GUI .....	20
Platform dependency .....	20
Minimal possible profiling overhead .....	21
Easiness of extensibility .....	21
Integration with bigger open-source application.....	21
2.3 Use Case Analysis .....	22
<b>3 SOLUTION DESIGN .....</b>	<b>23</b>
3.1 Profiling Engine Architecture .....	23
Agent architecture .....	25
3.2 Profiling Features and Approaches .....	27
Profiling Scenarios.....	27
Profiling levels .....	29
Method call filters.....	29
3.3 Thin Client and User Interface.....	30
Web container and project management .....	30
Apache wicket .....	31
3.4 Technology Summary List.....	34
<b>4 IMPLEMENTATION .....</b>	<b>35</b>
4.1 Profiling Agent Implementation.....	35
Method call profiling.....	37
CPU usage profiling.....	37
Memory usage profiling .....	38
Thread activity profiling .....	38
Class profiling .....	39
4.2 Graphical User Interface Implementation .....	40
4.3 Profiling Scenarios Implementation .....	42
4.4 Profiling Levels Implementation.....	44
4.5 Method Call Filters Implementation .....	45
<b>5 EVALUATION .....</b>	<b>47</b>
5.1 Manual Tests .....	47
5.2 Profiling Overhead Testing.....	48
5.3 Tests with MidPoint .....	49

5.4 Automatic Tests .....	50
<b>6 CONCLUSION.....</b>	<b>51</b>
<b>BIBLIOGRAPHY .....</b>	<b>53</b>
<b>APPENDIX A – TECHNICAL DOCUMENTATION.....</b>	<b>55</b>
<b>APPENDIX B – USER GUIDE .....</b>	<b>63</b>
B.1 Site Map .....	63
B.2 Guides.....	64
<b>APPENDIX C – TEST RESULTS .....</b>	<b>67</b>
C.1 Manual Test Results.....	67
C.2 Profiling Overhead Measurement .....	68
<b>APPENDIX D – GLOSSARY .....</b>	<b>71</b>
<b>APPENDIX E - FIGURE LIST .....</b>	<b>73</b>
<b>APPENDIX F – SOURCE CODE (CD MEDIUM).....</b>	<b>75</b>
<b>APPENDIX G – RESUME .....</b>	<b>77</b>

# Introduction

---

Generally speaking, performance and efficiency of software products are one of the most important software attributes nowadays. Modern software solutions work with parallel requests from thousands of users simultaneously. Of course, every single user expects immediate responses to interaction with software. “Amazon: addition of one tenth of a second in response time is similar to 1% of revenues. Google: addition of half of a second in latency will decrease throughput for one fifth.” [1]. Time equals money, this well known phrase relates to modern software engineering in every way. Fast application can be developed only by using the most efficient approaches and by writing efficient source code.

The objective of this bachelor thesis is to design and implement tool that could be used to perform profiling of java applications. This tool should be unique in context of existing profiling solutions and should provide enhanced profiling functionality. Existing and known solutions do not fit our requirements, because they do not provide the ability to create specific profiling scenarios that would enable us to monitor only chosen functionality of analysed application. This is a limiting factor that we will try to overcome with this project.

This thesis is divided into several sections. In the first section, we performed analysis of profiling in general and existing profiling tools used to monitor java applications. The philosophy behind profiling is introduced here as well as the role of profiling in context of modern software engineering. In section two and three, we analysed requirements and specification of our tool and then proposed architecture and solution design. In section four, we discuss implementation phase of created application, which was also tested properly. Testing process and test results are shown and discussed in section five. Technical aspects, simplified user guide and detailed test results can be seen in appendixes. Source code of implemented profiling solution can be seen in attached CD medium.



# 1 Problem Analysis

---

There are many factors having influence to application performance. Software architecture (internal software structure, division into subsystems and realization of mutual relations), implementation (efficiency of written source code in context of specified requirements) and hardware and software infrastructure on which specific software is being executed belong among the most important ones. Performance attributes of developed software solution needs to be monitored during design and implementation phase as well as after deployment in customer environment. In general, it is almost impossible to create exact testing environment as customer is using.

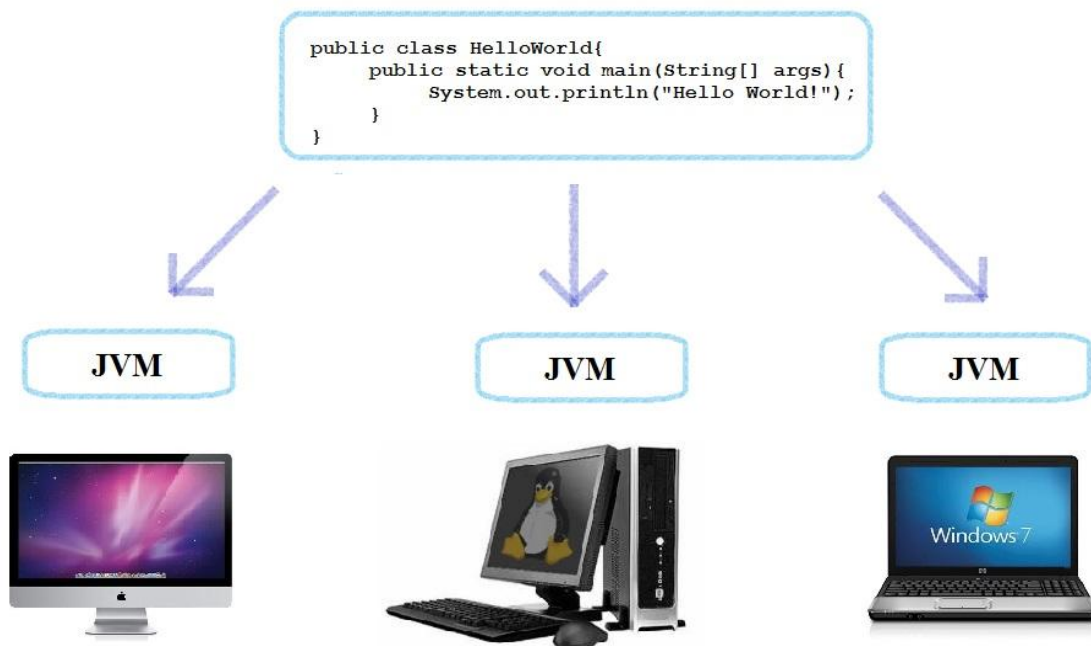
The main objective of this bachelor thesis is to create application that would be able to monitor and evaluate performance parameters of java applications. To accomplish this mission, we first need to analyse and deeply understand architecture and logic of java technology. Specifically, every step from writing source code to its execution on hardware configuration of computer needs to be understood. In this section, we will also discuss the profiling philosophy in general as well as its importance in context of software engineering. Analysis of current profiling techniques in context of java platform is included as well. Specifically, we focused on profiling support in standard Java API, analysis of existing and available open-source profiling tools as well as commercial profiling tools.

## 1.1 Java Virtual Machine Analysis

Java platform, respectively java programming language belongs among interpreted programming languages. In contrast with compiled programming languages, where source code written by programmer is compiled directly into binary form and then executed using instruction set of processor, in case of java, another phase needs to be taken into account. Java Virtual Machine (JVM) is the component that enters this process.

Java platform is connected to statement “write once, run everywhere”. This means, that program one written in java can be executed on different hardware and software infrastructures without additional source code modifications. Of course, this can be achieved only with prerequisite of installed JVM (see Figure 1). Every processor family possesses slightly different set of basic instructions as well as every operating systems communicates with underlying hardware layer using different system calls and solves problems using different approaches. Thanks to above stated facts, there exists large number of possible computer configurations. This logically indicates the conclusion that source code compiled on one computer cannot be executed on different machine. Java platform provides very interesting solution to this problem. As stated above, there is one additional component between writing source code and its execution. This

component is JVM and one of its main purposes is to reduce differences between various hardware and software configurations.

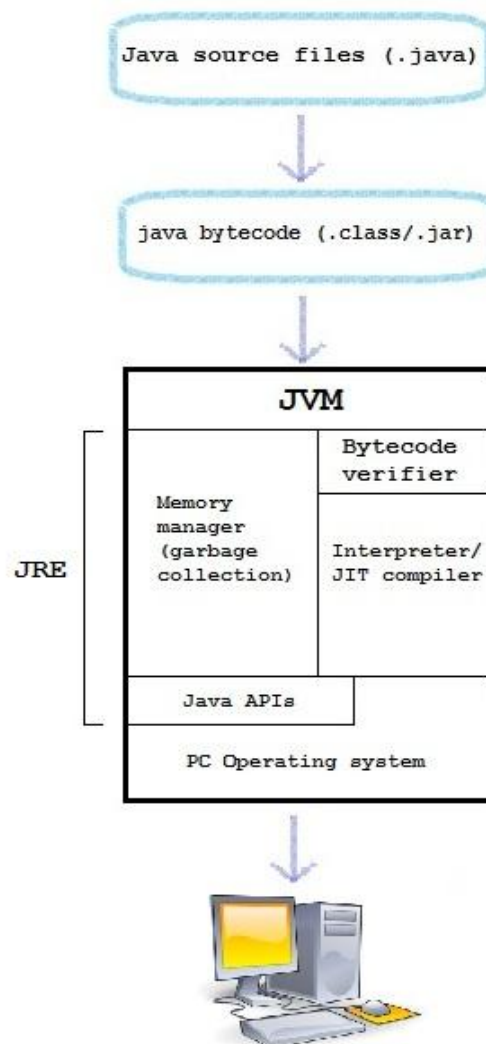


**Figure 1:** „write once, run everywhere“

JVM (Java Virtual Machine) is a program simulating the work of a computer system. This program runs in operating system (although, direct implementation on hardware is possible as well). It is a virtual computer. Same as any other program, it can be located inside computers main memory. In contrast with any other program located inside main memory, JVM contains functionality simulating the work of processor registers, stack or processor itself (JVM contains own instruction set). This enables JVM to behave exactly as full-featured computer. Similar to source code written in programming language C that is compiled to assembly code (symbolic instruction language); source code written in java is compiled into bytecode. “Understanding bytecode and what bytecode is likely to be generated by a Java compiler helps the Java programmer in the same way that knowledge of assembly helps the C or C++ programmer”. [3] This approach directly secures portability of java applications between various combinations of hardware and operating system combinations. It also brings several disadvantages that can be reflected in efficiency and performance of java applications. Improvement in this field has been secured by JIT (just in time) compilation technology. It brings dynamic approach to compilation, where written source code is analysed just before compilation and only most used parts of source code is directly compiled. If there is a need to use source code that has not been compiled yet, this code is dynamically compiled during the execution of application. This technology offers large number of optimization into the field of source files compilation. Let’s use a simple client-server application implemented in java as an example. On server side, the



most effort is made to its fast and efficient execution, so we will perform deep source code analysis and compile as much files as we can, even all of it, if needed. Totally different situation occurs on the side of client. Here, we prefer agile application start with sequential source code compilation during execution. Dynamic source code compilation can be performed on source file (.java) level, class level as well as using method code granularity. [2] JVM is an open-source technology that respects set of standards. If all these standards are respected, every single programmer or company can develop their own compatible JVM. At the moment of writing this thesis, there exists about 24 proprietary and 35 open-source implementations of JVM [4]. The most famous and known one is JVM HotSpot, implemented and distributed by software company Oracle. JVM HotSpot uses JIT compiler and it is written in C++ programming language. Source files of JVM are composed from about 250 000 lines of code [5]. The scheme of java source code transformation and compilation can be seen on Figure 2.



**Figure 2:** JVM work scheme.

## 1.2 Profiling Principles Analysis

The concept of profiling in context of software engineering is a form of dynamic analysis of software product. In case of static analysis, analysed application is not executed. On the other hand, when performing dynamic program analysis, analysis of basic applications performance attributes are being measured and monitored during its execution. Some examples of these attributes are the usage of main memory, time complexity of used algorithms (or the entire application), frequency and duration of individual method calls and operations. The purpose of profiling is to provide data necessary to perform performance optimization. Importance of this aspect is discussed in previous section.

History of profiling is written since 70's of 20<sup>th</sup> century. The first attempts to perform program profiling during its execution were made on IBM/360 and IBM/370 platforms using so-called sampling method. When using sampling technique, application execution is regularly interrupted by special instruction and process of performance data collection starts. Collected information is later evaluated to find ineffective algorithms or other parts of source code. Since 1979 UNIX based operating systems started to use simple tool called 'prof'. This tool was used to collect information about frequencies and execution times of individual functions. Later in year 1982, this concept was enhanced by 'gprof' tool. This tool was able to summarize more complex information composed of several function calls. [6]

Every single profiling tool analysis target application during its execution. While collecting performance data, one simple fact needs to be taken into account. Dynamic performance data collection itself changes normal execution of analysed application and it has direct influence to application performance. We can categorize profiling tools by data collection technique:

- Event-based profiling tools,
- statistical profiling tools,
- instrumenting profiler

### Event-based profiling tools

Using this approach to profiling, performance information are collected based on well defined set of events that occurs periodically during analysed application execution. Of course, we are speaking about software based events, such as method calls, thread switch, process rescheduling and others. The frequency and type of events causing interruption are provided by user. In contrast with statistical profilers, this approach provides far more profiling options and modifications.

## Statistical profilers

Main technique used in statistical profiling tools is sampling. During execution of analysed application we perform interrupts using special operating system instruction. After this interrupt, we perform performance data collection. In general, this approach is less accurate and specific than event-based approach. On the other hand, it has an advantage of less profiling overhead. In some cases, profiling tools using this approach are able to detect performance problems that are not detected using event-based approach.

## Instrumenting profilers

This profiling method is based on analysed application source code manipulation. These modifications are performed with purpose of performance data collection. This approach always has a negative impact on analysed application performance, since injected code needs to be executed as well as native application code. In contrast of previous profiling techniques, using instrumentation we can perform very specific analysis of individual parts of source code. The amount of profiling overhead can be minimized using efficient source code injection and by choosing most appropriate method of performance data collection in specific place and time. There are many ways of dynamic source code instrumentation:

- *Manual*: target application is instrumented manually by programmer. He can inject profiling code in any place of target application. In this case, profiling possibilities are limited only by skills and imagination of programmer,
- *automatic on source-code level*: Profiling code is automatically injected into target application. How and where this code is injected is decided by so-called instrumentation policy,
- *compiler assisted*,
- *binary translation*: compiled profiling code is injected directly into compiled target application,
- *runtime instrumentation*: target application is instrumented directly before its execution. Target program execution is fully under control of profiling tool in this case,
- *runtime jump injection*: similar approach as runtime instrumentation, but instead of full-featured profiling instructions are injected, in this case we only inject jump instruction to our profiling code that does not run within target application. This approach brings less profiling overhead and is more lightweight solution.

Profiling code injected into target application communicates with target JVM and sends requests to collect performance information. Target JVM using integrated profiling interface periodically replies by sending requested information. These data are then evaluated by

profiling agent and sent (in raw or modified form) to profiling application that can be on same computer, where target JVM is running or on different computer. Profiling application can be implemented in various ways. It can be simple CLI (command-line interface) application, graphical application etc. JVM profiling interface provides standard set of profiling events. These are:

- entrance and exit from method call,
- allocation, relocation or release of object,
- creation and deletion of heap,
- start and end of garbage collection cycle,
- method compilation,
- thread creation and deletion,
- class compilation,
- java monitor based events: entry wait, running and end of a monitor,
- monitor release,
- initialization and end of an JVM.

It is necessary to realize, what profiling really is about, realize the difference between debugging and profiling an application. Debugging is a necessary activity in every software development process. The purpose of debugging is to create an application with exactly the same functionality as stated in requirements of customer. The purpose of profiling is to enhance and improve performance attributes of analysed application. "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil"[7]. Application profiling is rather time-consuming activity, thus it is necessary to consider pros and cons of profiling before the developer starts with this activity.

As mentioned before, profiling collects performance data mostly about CPU activity and memory usage during execution of analysed application. Types of information collected related to memory usage:

- number of allocated object of certain type,
- pieces of code invoking allocation requests,
- methods responsible for allocation requests,
- objects allocated during application execution that are not used and cannot be freed by garbage collector. These objects are responsible for heap flooding and eventually, they can cause a memory leak, which can lead to out of memory errors,
- excessive allocation of temporary objects that can raise the amount of garbage collection cycles, thus lowering execution of application effective code. This can lead to stagnation of application performance.

Information collected related to CPU activity:

- number of method calls,
- CPU usage during execution of certain method. If there are different method calls inside of analysed method, it is possible to monitor their execution separately,
- number of CPU cycles related to certain method execution,
- user time related to execution of certain method including I/O (Input/Output) operations, locks, monitors etc. This time and generally all of this information mostly depends on underlying hardware layer.

### **1.3 Analysing of Profiling Possibilities of Standard Java API**

Basic profiling capabilities, respectively tools are built inside java standard Software Development Kit (SDK). Version J2SE 5.0 (Tiger) has introduced Java Virtual Machine Tool Interface (JVM TI) that provided functionality to examine state and monitor performance of java applications running on underlying JVM. JVM provides API (Application Programming Interface) that can be used to develop tools, which needs direct access to JVM such as profiling tools. During JVM initialization, special subroutines (libraries) are connected to it monitoring its state, registering events such as described above subsequently sending all this information to profiling tool, where further evaluation can be performed. Data in final form are then presented to user. JVM TI replaces the functionality of JVM PI (Java Virtual Machine Profiling Interface) and JVM DI (Java Virtual Machine Debug Interface) and improves them in many ways. JVM PI and JVM DI are not found in standard java API since version J2SE 6.0. JVM TI is the lowest possible layer accessing and handling performance information. It is a basic technology that is used by most existing profiling solutions.

J2SE provides a very simple profiling tool (which is also based on JVM TI technology). This tool can be used to monitor CPU activity and memory usage during execution of an application. Tool named HPROF is a simple command line program. It is a native library, which is dynamically attached to JVM during its initialization. Various types of CPU and memory usage can be selected using HPROF. Generated output of this tool can be in binary or text form. Binary output of HPROF tool is used by several existing profiling solution. It is an input to these tools later transformed to human readable form (in form of graphs and tables). This output may be more useful to user, than rather chaotic text file generated by HPROF. HPROF can be used from command line in following way:

```
java -agentlib:hprof[=options] ProfilovanáTrieda(.class) [-jar
súbor.jar]
```

In this section, we will show two very simple examples of work with HPROF tool. In the first one, we will present CPU activity monitoring, while in the second one, memory usage will be analysed.

### Example 1: CPU activity analysis

For the purposes of demonstration in this example, we have created very simple program that generates numbers of Fibonacci sequence using recursion. Source code can be seen on Figure 3.

```
package cpuTest;

public class CpuTestFibonacci {
    public static void main(String[] args){
        long vysledok = fibonacci(25);
        System.out.println(vysledok);
    }

    private static long fibonacci(int pocet){
        if(pocet <= 1){
            return pocet;
        }
        else{
            long medzivysledok = fibonacci(pocet-1) + fibonacci(pocet-2);
            return medzivysledok;
        }
    }
}
```

**Figure 3:** Example of java source code analysed by HPROF tool

Generated .jar file was subsequently executed using command line (we used sampling method. This technique was selected by attribute in `cpu=times` following example)

```
java -agentlib:hprof=file=fib_prof.txt,cpu=times -jar fib.jar
```

In generated text file, following information can be found:

```
CPU TIME (ms) BEGIN (total = 514) Sun Dec 02 23:33:33 2012
rank  self  accum  count trace method
  1  91.83%  91.83%  242778 301727 cpuTest.CpuTestFibonacci.fibonacci
  2   0.39%  92.22%     4 300800 sun.net.www.ParseUtil.decode
  3   0.39%  92.61%     6 300570 java.util.jar.Attributes$Name.isValid
  4   0.19%  92.80%     1 301569 java.io.FilePermission$1.run
  5   0.19%  93.00%     1 301513 java.lang.ref.Reference.<init>
```

```
TRACE 301727:
cpuTest.CpuTestFibonacci.fibonacci(CpuTestFibonacci.java:9)
cpuTest.CpuTestFibonacci.fibonacci(CpuTestFibonacci.java:9)
cpuTest.CpuTestFibonacci.fibonacci(CpuTestFibonacci.java:9)
```

Of course, this is just a simple example showing a few lines of profiling output. Generated file is much more detailed, but it serves well as a presentation of HPROF work. In generated file, we can see method calls sorted by their average percentage usage of CPU. Field „count“ does not show the number of certain method call. It shows the number of method occurrences in all captured stack traces.

## Example 2: Memory usage analysis

For the purposes of this example, we have created a simple program that continuously adds one million of Integer objects to ArrayList. The source code of created program can be seen on Figure 4.

```
package memTest;

import java.util.ArrayList;
import java.util.List;

public class MemoryTestHprof {
    public static void main(String[] args){
        List<Integer> list = new ArrayList<Integer>();
        for(int i = 0; i < 1000000; i++){
            list.add(new Integer(i));
        }
        System.out.println("Pocet objektov: " + list.size());
    }
}
```

**Figure 4:** Source code example used to demonstrate memory usage analysis using HPROF

Generated .jar file was subsequently launched from command line (we used sites analytic method using heap-sites attribute in command):

```
Java -agentlib:hprof=file=mem.txt,heap=sites -jar mem.jar
```

Following information can be seen can be seen in generated file:

rank	self	accum	bytes	objs	bytes	objs	trace	name
1	56.28%	56.28%	16000000	1000000	16000000	1000000	300404	java.lang.Integer
2	42.43%	98.71%	12063904	28	12063904	28	300405	java.lang.Object[]
3	0.06%	98.77%	16416	2	16416	2	300061	byte[]
4	0.01%	98.79%	3736	26	3736	26	300006	char[]

The first place in object memory usage is expected. The second place is quite a surprise. Let's take a close look at several stack traces.

```
TRACE 300404:
    java.lang.Number.<init>(<Unknown Source>:Unknown line)
    java.lang.Integer.<init>(<Unknown Source>:Unknown line)
    memTest.MemoryTestHprof.main(MemoryTestHprof.java:10)
TRACE 300405:
    java.util.Arrays.copyOf(<Unknown Source>:Unknown line)
    java.util.Arrays.copyOf(<Unknown Source>:Unknown line)
    java.util.ArrayList.ensureCapacity(<Unknown Source>:Unknown line)
    java.util.ArrayList.add(<Unknown Source>:Unknown line)
```

We can notice, that plenty of space is allocated for help arrays that are used by ArrayList operations during addition of individual Integer objects. Let's perform a simple modification in analysed program. During ArrayList definition, we will provide information about number of inserted elements to compiler. Modified source code can be seen on Figure 5.

```
package memTest;

import java.util.ArrayList;
import java.util.List;

public class MemoryTestHprof {
    public static void main(String[] args){
        List<Integer> list = new ArrayList<Integer>(1000000);
        for(int i = 0; i < 1000000; i++){
            list.add(new Integer(i));
        }
        System.out.println("Pocet objektov: " + list.size());
    }
}
```

**Figure 5:** Example of modified source code used during memory usage analysis with HPROF

The results of work with memory usage of modified application are different:

rank	self	accum	bytes	objs	bytes	objs	trace	name
1	78.59%	78.59%	16000000	1000000	16000000	1000000	300405	java.lang.Integer
2	19.65%	98.23%	4000016	1	4000016	1	300404	java.lang.Object[]
3	0.08%	98.31%	16416	2	16416	2	300061	byte[]
4	0.02%	98.33%	3856	26	3856	26	300006	char[]
5	0.01%	98.35%	2640	5	2640	5	300043	byte[]
6	0.01%	98.36%	2160	2	2160	2	300311	byte[]

This time we are not using help fields and objects during addition operations with ArrayList. With simple code modification, we have managed to significantly improve work with main memory.



## 1.4 Analysis of Chosen Open-Source Profiling Solutions

Offering in the field of open-source profiling solutions is pretty wide. Every single profiling tool is slightly different from others. Choosing one most suitable for our needs may not be an easy task. The objective of this thesis is not to describe all existing profiling solutions, nor their reviewing, but we will focus on some of them that were mostly interesting in the eyes of an author of this text.

Heap Analysis tool (HAT) – simple profiling tool that uses HPROF utility output as input (see section 1.3). HAT reads HPROFs output, performs detailed analysis and present gained performance knowledge in an interesting way. User is able to see the topology of objects captured from heap analysis. This tool is mostly oriented on detection of memory leaks, but it also provides basic information about application execution times.

JProbe – graphical profiling tool containing three separate units:

- JProbe ThreadAnalyser, tool oriented on detection of potential deadlocks or unwanted blocking of execution of individual threads.
- JProbe Coverage, tool oriented mostly on test purposes rather than profiling. It helps user to better understand performance of tested applications. It enables user to perform several tests, which results shows us, how many times certain lines of code were executed, what parts of code were not executed at all etc.
- JProbe Memory Debugger offers users complex and comprehensive statistics about memory handling of executed java application. It enables us to see what objects consumes the most memory or objects that cannot be freed by garbage collection cycle, thus creating memory leak.

JConsole is a simple graphic profiling tool that can be used to monitor state of JVM and java profile java applications running on it. It can be used to perform profiling on local or remote machine using remote access. JConsole is a tool closely linked to JVM and provides information about the usage of system resources by analysed java application. JConsole uses JMX (Java Management Extensions) technology. JConsole is a part of standard JDK (Java Development Kit) and it can be launched from command line using command 'jconsole'.

VisualVM is a graphical profiling tool that integrates several existing software tools from standard JDK providing complex profiling options. Profiling can be again performed on local or remote machine. It can be used to perform profiling during implementation and testing phase of

software development lifecycle as well as after deployment in customer and production environments. This tool is built on NetBeans platform and provides modular and easily extensible architecture.

List of another open-source profiling solutions:

- Cougaar Memory Profiler
- JTreeProfiler
- JRat
- Extensible Java Profiler
- JMP
- TomcatProbe
- Allmon
- Perf4j
- InspectIT
- JBoss Profiler ...

Eclipse Test & Performance Tools Platform (TPTP) project is closely connected to favourite IDE Eclipse. It is an open-source platform providing several frameworks and other services directly used to create testing and profiling applications or plugins. TPTP contains large number of separate complex tools that can be used in every phase of software development lifecycle. It supports several types of java applications, from embedded java systems to standalone or enterprise java applications. This platform is quite controversial as there is a lot of criticism on it. It is stated by many, that Eclipse TPTP has become a victim of its own robustness and over-engineering. There are many other profiling tools that can be used as eclipse plugins. Quick search of profiling tools in Eclipse Market shows us 24 profiling solutions for java applications.

Netbeans Profiler is a tool directly integrated with Netbeans IDE since version 6.0. This tool is based on results of research project JFfluid Sun Laboratories. This research project has revealed that with using profiling technique called 'dynamic bytecode instrumentation' it is possible to achieve minimal general profiling overhead. This tool also enables us to use so-called profiling points. These points can be injected into java source files by developer and profiling tool will retrieve the state of JVM specifically in these selected points of execution.

## 1.5 Analysis of Commercial Profiling Solutions

Of course, there are many commercial profiling solutions on the market. Our choice for the purposes of this section was YourKit java profiler. Author of this thesis spent several hours using this product while testing performance attributes of large open-source solution. YourKit java profiler is quite robust tool providing large amount of profiling possibilities. It is designed to work with different types of java applications, for example embedded systems, J2ME, J2SE or J2EE software products. It is also able to perform remote profiling over TCP/IP. We specifically enjoyed the functionality of so-called ‘profiling on request’. Using this function, we can tell profiler when to perform only basic profiling in order of reducing the profiling overhead and when to launch deep analysis (at night, for example). Using this function, developer is able to perform profiling providing deep information about analysed application performance and maximum possible reducing of overall profiling overhead.

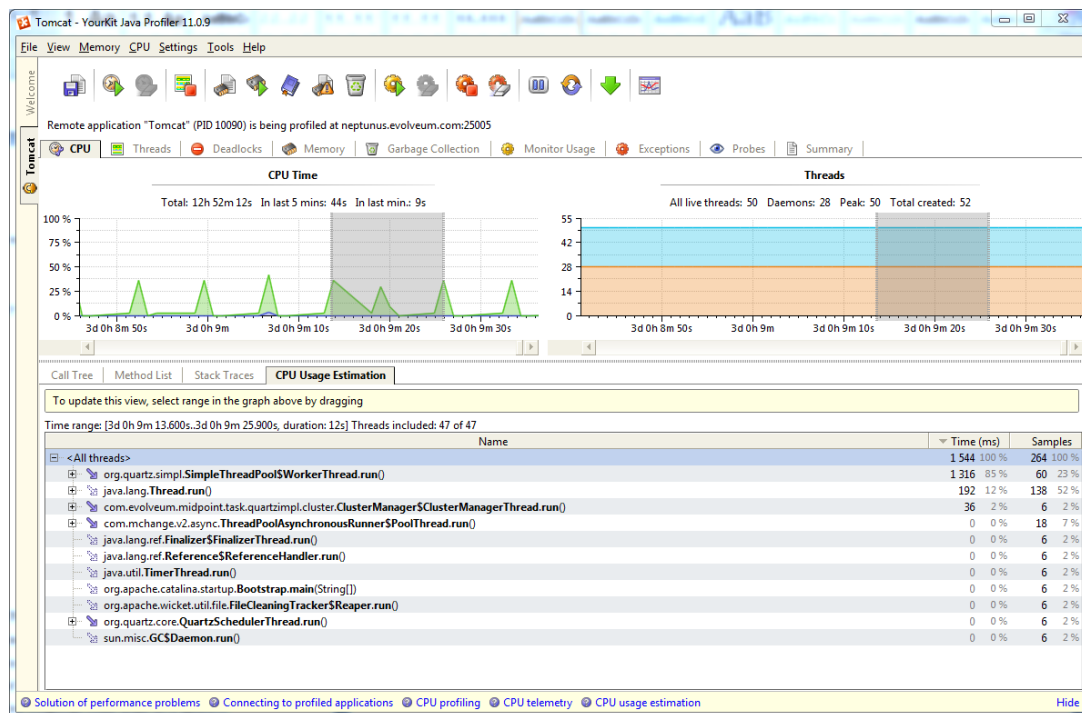
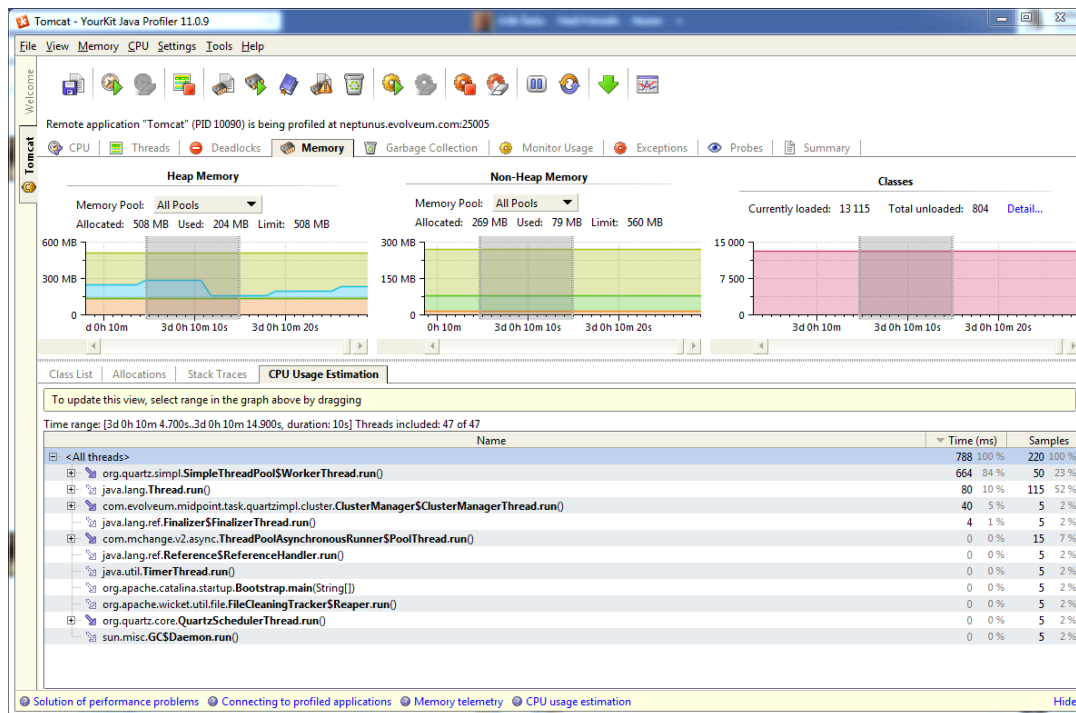


Figure 6: CPU usage profiling using YourKit profiler.

YourKit provides large amount of options when it comes to filter data collected by profiling. We can select the classes whose performance attributes are interesting to us. Of course, user can integrate YourKit profiler with most used java IDEs like Eclipse, NetBeans or IntelliJ IDEA.



**Figure 7:** Memory profiling using YourKit java profiler

Graphical design of this tool is very neat and work with it is fast and intuitive. There can be found large number of tutorial and very detailed information about the usage of this product. The support from developers is great as well. In general, support can be a disadvantage of many open-source solutions, not only in the field of java application profiling.

## 2 Requirements Analysis and Specification

---

The main objective of this bachelor thesis is to implement a software solution, which would be able to monitor and evaluate performance attributes of java applications during their execution. After reading the previous section devoted to profiling analysis of java applications and tools designed to serve this purpose, observant reader might get an impression that we are trying to develop something, that has already been created multiple times. Therefore, in this section, we will also discuss uniqueness of our solution in the context of already existing solutions. This section is divided into three tree sections, in the first one, functional objectives are discussed. In second section, we will focus on non-functional requirements and in the last one, use case model of our solution will be presented.

### 2.1 Functional Requirements

Based on several consultations and analysis of profiling in section one, these functional objectives were identified:

- Active profiling data collection using multiple profiling techniques,
- real-time representation and evaluation of basic performance attributes of CPU usage during analysed program execution (as listed in section 1.2),
- real-time representation and evaluation of memory usage during analysed program execution (as listed in section 1.2),
- real-time thread activity profiling,
- real-time class load statistics,
- real-time method call profiling,
- user defined profiling scenarios,
- user defined levels of profiling, in other words, granularity of profiling,
- user defined method calls filters based on method parameter values,
- solution will provide basic information about software and hardware configuration of a computer system and JVM, on which program analysis is running.

#### **Client – server architecture**

Client – server architecture is quite unusual for profiling solutions. Most existing java profilers are implemented as thick clients, but thin client approach can provide several advantages. This solution can be deployed on one of the servers of software company and all of its developers are

able to perform profiling of their code on unified profiling system, and since all profiled applications are executed on this server, developers can focus on performance problems rather than on several categories of system problems resulting from executing profiling on different software or hardware configurations. Unfortunately, this approach also requires that all analysed source code or every java application packed in jar format must be uploaded on profiling server before profiling can be executed.

### **Real – time representation of CPU and memory usage of analysed application**

Before collected data from analysed java application can be shown to user, we need to evaluate them and transfer them into understandable form that will provide enough information to find performance problem. Based on user defined filters and profiling attributes, data collected from profiling will be shown in form of interactive graphs and tables.

### **User defined profiling scenarios**

The concept of user defined profiling scenarios is quite simple and effective. Java applications, especially enterprise java applications are often very large and complex software solutions consisting of hundreds, even thousands of java classes and even more methods. Instrumenting all of this source code and following data transmission and evaluation can be very expensive in terms of usage of system resources, not to mention uselessness of this approach, when developer has clear idea, what component of analysed application is causing performance problem. User-defined profiling scenarios are the solution to this problem. Let's explain this concept on simple example.

Imagine, that analysed application offers a functionality to create a new user. This action is often composed of several sub actions like data validation and collection from user provided data in graphical user interface, communication with data storage, an attempt to write this data, thus create new entry in used data storage and several other sub actions. The whole process can be implemented in one method calling several other methods from different classes and together, they are responsible for creating new user. The concept of user-defined profiling scenarios offers mechanisms for analysing scenarios as described above, while ignoring every other operation performed by analysed java application, thus minimizing general overhead caused by profiling. Users of our profiling solution will be able to create, read, edit and delete these profiling scenarios, as well as assigning them to profiling project.

## **User defined profiling levels**

Profiling levels represents another mechanism offering developer a way to perform java application analysis in a very specific way. While when using profiling scenarios, we were able to define which classes and which methods we want to be profiled, with profiling levels we are able to define, what aspects of performance will be monitored. Let's again introduce this concept with simple example.

During testing of java application, a developer has noticed, that system memory consumption during execution of tested application grows too much. The logical conclusion of this situation is the obvious presence of a memory leak in tested application that needs to be located and fixed. In this scenario, the only profiling aspect, that developer is interested in, is analysis of memory usage of analysed java application, thus collecting and evaluating data about CPU usage, threads and classes is irrelevant, unnecessary and raises general profiling overhead. The way to solve this problem is the creation of profiling level and selection of all profiling aspects, that developer needs, thus skipping unneeded profiling aspects and minimizing general profiling overhead. Some of the already existing profiling solutions provide similar type of functionality, but we will try to extend it by enabling user to set profiling intervals for each aspect of profiling during creation of profiling level.

## **User defined method calls filters**

Java applications, especially enterprise java applications are often designed to be easily extensible. This often leads to writing methods that work with several kinds of objects or performs several possible scenarios based on their parameters defined on method call. Let's imagine method `addUser()`, that is designed and implemented to work with several databases, LDAP server, Active Directory server and some other forms of data storage solutions, while the currently used data storage is determined by value of method parameter, or combination of values of several method parameters.

Recently, developer has noticed, that work with MySQL database and actions performed upon this database are very slow. The first step taken would be a creation of profiling scenario and selection of method `addUser()`, which we want to analyse. This approach would lead to analysis of all `addUser()` method calls, not just calls working with MySQL database, so the question is, are we able to somehow define, that we only want to analyse `addUser()` method calls in context of MySQL database? Of course there is a solution, developer can create a method call filter based on values of method call parameters, thus defining the database, with what interaction needs to be analysed.

## 2.2 Non-functional Requirements

Except above defined functional objectives of our profiling solution, other non-functional objectives were defined during consultation sessions and based on profiling analysis discussed in section one of this text. These are:

- Our solution will be implemented as a web application, in other words, thin client architecture will be used,
- intuitive usability,
- user-friendly graphical user interface,
- platform dependency configuration,
- minimal possible profiling overhead,
- easiness of extensibility,
- integration with bigger open-source application.

### Intuitive usability and user friendly GUI

Profiling solutions are designed to serve application developers. Our solution will be designed specifically for java developers. The usage of this system assumes that user has certain level of knowledge in java development and may be confusing for users lacking this knowledge. Another assumption, that needs to be stated, is that user, who wants to perform profiling using our tool (or any other profiling solution in general), needs to have knowledge of analysed java application, otherwise, the user might not get wanted results.

Graphical user interface will be designed using modern technologies in context of web development. Our profiling solution will be easy to use. Design should not be confusing and should provide necessary hints when needed. User should easily figured out, how to perform wanted actions when interacting with our solutions GUI. We will try to aim on GUI ergonomics as well.

### Platform dependency

Profiling solutions in general performs lots of interaction with underlying operating system and the usage of system resources is frequent as well, thus platform independency in this context is a target that is not easily achieved. Our solution will not be platform independent in its default form, but it should be easily configured to run on every main platform. For further explanation on this topic, simple example will be shown.



Our solution is running on a Linux server without any GUI libraries that are necessary when creating graphical user interface with swing or awt. Since profiling is always performed on this server, user, who wishes to perform analysis of his thick client application designed with swing, will be disappointed, because his application won't be able to run on server without necessary libraries.

For purposes of this bachelor thesis, our solution will be designed for deployment on Microsoft Windows based platforms.

### **Minimal possible profiling overhead**

Our solution is designed to have as minimal profiling overhead as possible. Specific steps taken to achieve this target are described in section 1.1. It is impossible to achieve 0% profiling overhead, thus there will always be certain inaccuracy in dynamic application analysis.

### **Easiness of extensibility**

We will design this profiling solution to be easily extended with other functionality. Our solution may be used for years, thus changes in functionality might be needed and easiness of extensibility is a way to achieve them with as simple effort as possible. The way to achieve this non-functional objective is to take it into account from the very beginning of design of software solution not only in context of profiling solution, but generally in software development.

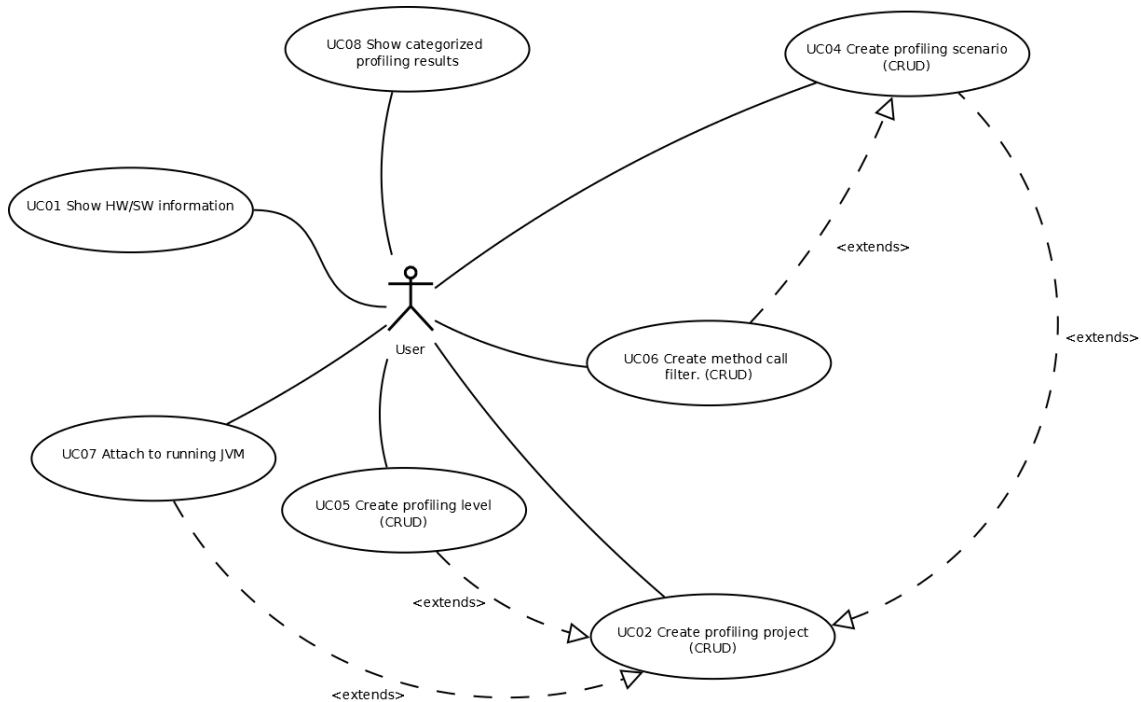
### **Integration with bigger open-source application**

The last, but not least non functional objective is integration with bigger open-source application. This objective is out of context of this bachelor thesis, but it needs to be stated for simple reason. Our profiling solution is designed in a way that ensures easiness of later integration with mentioned open – source application and selection of used technology is as well affected by this fact.

## 2.3 Use Case Analysis

In this section, we will briefly discuss use case analysis, which is an output of section 2.1 and 2.2. Based on functional, non functional objectives and consultations, these use cases were identified:

- UC01 Show Hardware/Software information,
- UC02 Create profiling project (CRUD),
- UC04 Create profiling scenario (CRUD),
- UC05 Create profiling level (CRUD),
- UC06 Create method call filter (CRUD),
- UC07 Attach to running JVM,
- UC08 Show categorized profiling results.



**Figure 8:** Use case diagram of implemented profiling solution

Relations between individual use cases are shown in figure 8 using a use case diagram. In this figure, reader can clearly see the possibility to include profiling scenarios, profiling levels and method call filters within profiling project. These possibilities give user dynamic set of rules and mechanism to analyse their java applications in a very specific view while keeping general profiling overhead on minimum. Use cases marked with CRUD identifier expresses compound use cases, what means that create, read, update and delete functionality is included in these use cases. Detailed and more technical description of this use case diagram can be seen in appendix A, Technical documentation.

## 3 Solution Design

---

In previous section, we have defined and discussed main objectives of our java applications profiling solution. In this section, we will try to choose the most suitable technologies and approaches that would solve individual sub problems and together meet requirements specified in previous section.

Chosen frameworks and technologies will not be discussed in much detail, that is not the purpose of this thesis, but we will state why each technology, framework or approach has been chosen. In cases, where completely new solution is created and implemented (in other words, we are not using already existing technology). We will try to describe our approach in more detail using UML besides classic text description.

### 3.1 Profiling Engine Architecture

In this section, we will focus on the most important component of every profiling solution, profiling mechanisms.

First of all, we need to find a way to tell analysed program (running in separate JVM) that we want to collect information about its performance, apply rules of analysis, collect data and send them to profiler server, where we will evaluate them and show them to user. To sum up, we need to possess a certain level of control over JVM, on which analysed application is running. There are several ways to achieve this. These techniques were described in section one. We have chosen dynamic code instrumentation for the purposes of this bachelor thesis, because it is the only way to perform a very specific performance analysis, which our profiling solution will provide.

In section 1, while analysing existing profiling solutions, we have noticed, that all profilers use java agents, programs that are created to provide control over JVM, where analysed application is executed. These agents can be connected to target JVM in two ways:

- On start of JVM using command line commands,
- dynamically during the execution of JVM with analysed application.

Java provides simple mechanisms for both ways of connection between profiling agent and target JVM. The first case has already been described. For the second case, java attach API can be used. This API can be found in `com.sun.tools.attach` package.

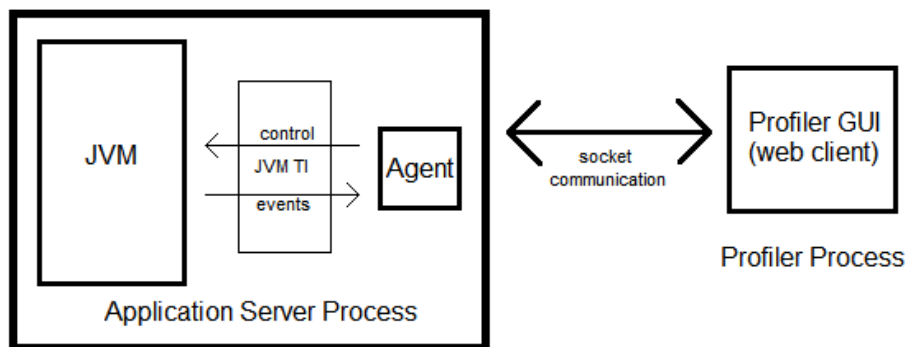
So far, we have managed to attach our profiling agent to analysed JVM. Next necessary step to take is to somehow tell target JVM, what performance information we want to collect. The answer can be found in standard java API again, we will use `java.lang.instrument`

package. Usage of this API provides us with functionality to transform every loaded java class into form necessary for purposes of profiling.

Another very important question, that needs to be answered, is how to transport data collected by profiling agent to our profiling application. Since profiling and analysed application both run in separate JVMs, therefore in separate process, we will need to find solution for inter process communication. There are several approaches that could be applied:

- Shared memory,
- pipes,
- queues,
- direct memory access (DMA),
- java sockets,
- java RMI (remote method invocation),
- java JMX (java management extensions), ...

The most suitable form of inter process communication for our profiling solution seems to be the usage of java sockets, as this concept provides ability to perform remote profiling, which is necessary for client – server architecture. So far, we have established the basic scheme for our profiling engine, it can also be seen in figure 9. Let's take a closer look on profiling agent and dynamic code instrumentation in next section.



**Figure 9:** Basic profiling architecture scheme. JVM TI stands for Java Virtual Machine Tool Interface.

## Agent architecture

Simply said, a java profiling agents is just a set of classes packed usually in a .jar (java archive) file, but it is the most important part of every profiling solution. Let's construct a list of tasks that our profiling agent will perform:

- Dynamic bytecode instrumentation of compiled classes,
- provision of profiling code for collecting data of selected aspects of profiling, specifically source code for CPU, memory, thread and class activity profiling,
- way to transfer collected profiling data to profiling server.

We have already covered the third point in text above, just to repeat, java sockets will be used as communication framework between profiling agent and profiling server.

Dynamic bytecode instrumentation or injection is a technique used to transform, or manipulate with compiled java classes. In the context of profiling, this technique is used to inject java code or bytecode that will catch events like method call or object creation and collect various profiling data. Basically, there are two possible approaches, how to perform dynamic bytecode injection:

- Direct bytecode instruction injection using ASM framework – this approach requires certain level of knowledge of bytecode programming, thus is not very suitable for developers without this kind of knowledge. On the other side, since we are injecting direct bytecode instructions, we are able to skip the phase, where transformed class needs to be recompiled. This also brings great level of risk, since we can easily inject incorrect bytecode instructions, what can knock off entire underlying JVM,
- java code injection – using this approach, we are able to inject lines of classic java source code. An example of framework with this functionality is Javassist API. Before class transformation is complete, Javassist recompiles injected source code, so all errors created during injection phase are revealed. This approach does not require any knowledge of java bytecode at all.

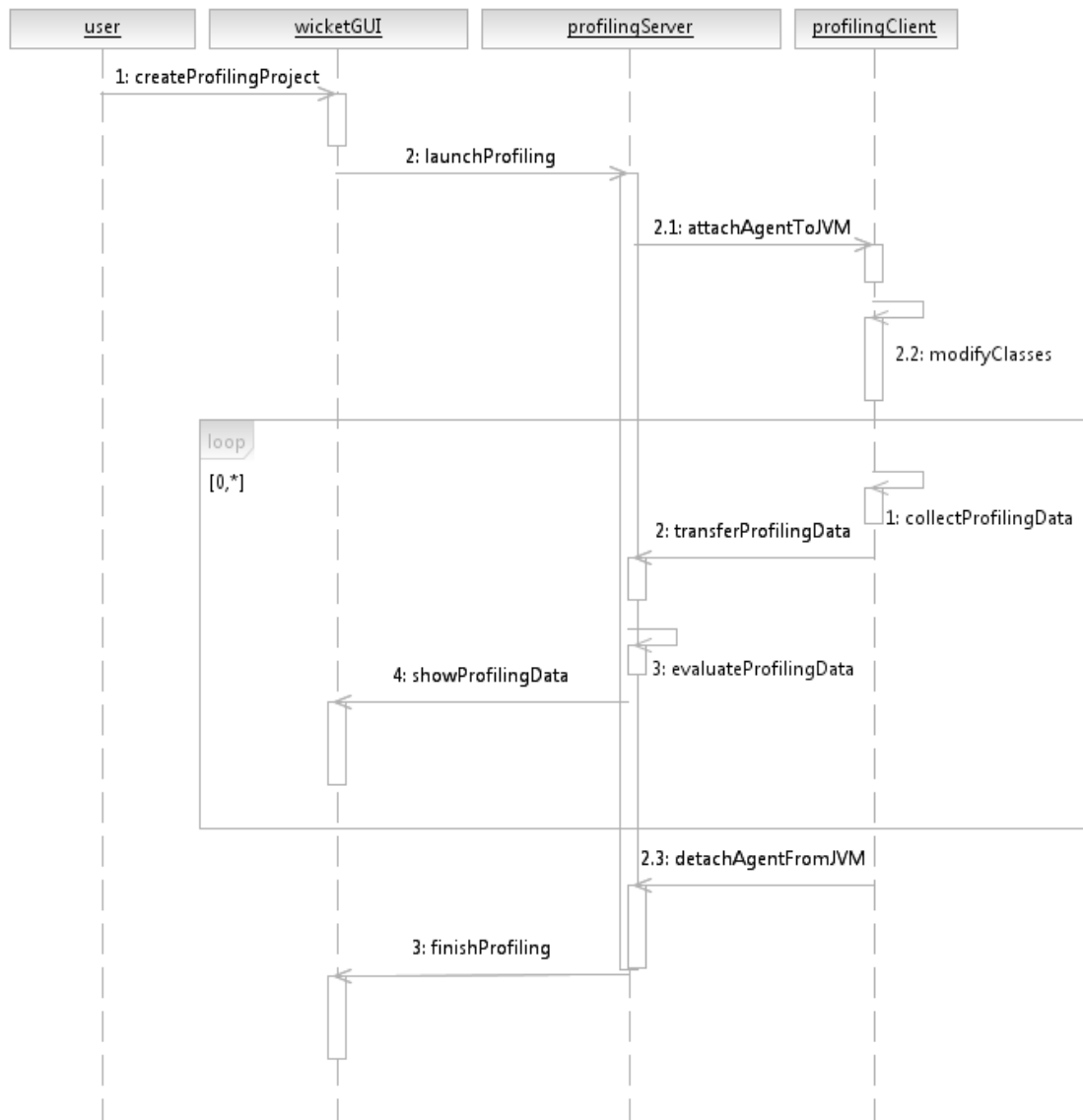
For obvious reasons, we have decided to use Javassist API.

Finally, we need to create a stack of technologies that will help us collect information about performance attributes of analysed java application. Answers can be found inside standard java API as well as in many great third party frameworks and technologies. Untimely, we have chosen several approaches for collecting performance information:

- Java Simon API – simple monitoring API. Will be used for measuring method call times,
- package `java.lang.instrument` – memory profiling, class usage profiling,
- package `java.lang.management` – Thread and CPU time profiling,

This stack of selected profiling technologies will help us collect information we need for profiling purposes. Of course, this list might not be final. There is a possibility that some selected technology might not be as suitable for our profiling solution as expected. This list will be finalized in next section.

So far, we have established basic concepts of profiling. These concepts are nothing new in the field of java application profiling as they are probably used by other already existing profiling solutions (some alterations are possible). This basic concept can be seen in figure 10. We have used UML sequence diagram to create this abstract profiling concept. It does not show profiling process with every detail, but that is not the purpose. In section 3.3, we will discuss features that will make our solution different from others.



**Figure 10:** Abstract profiling concept described using sequence diagram

## 3.2 Profiling Features and Approaches

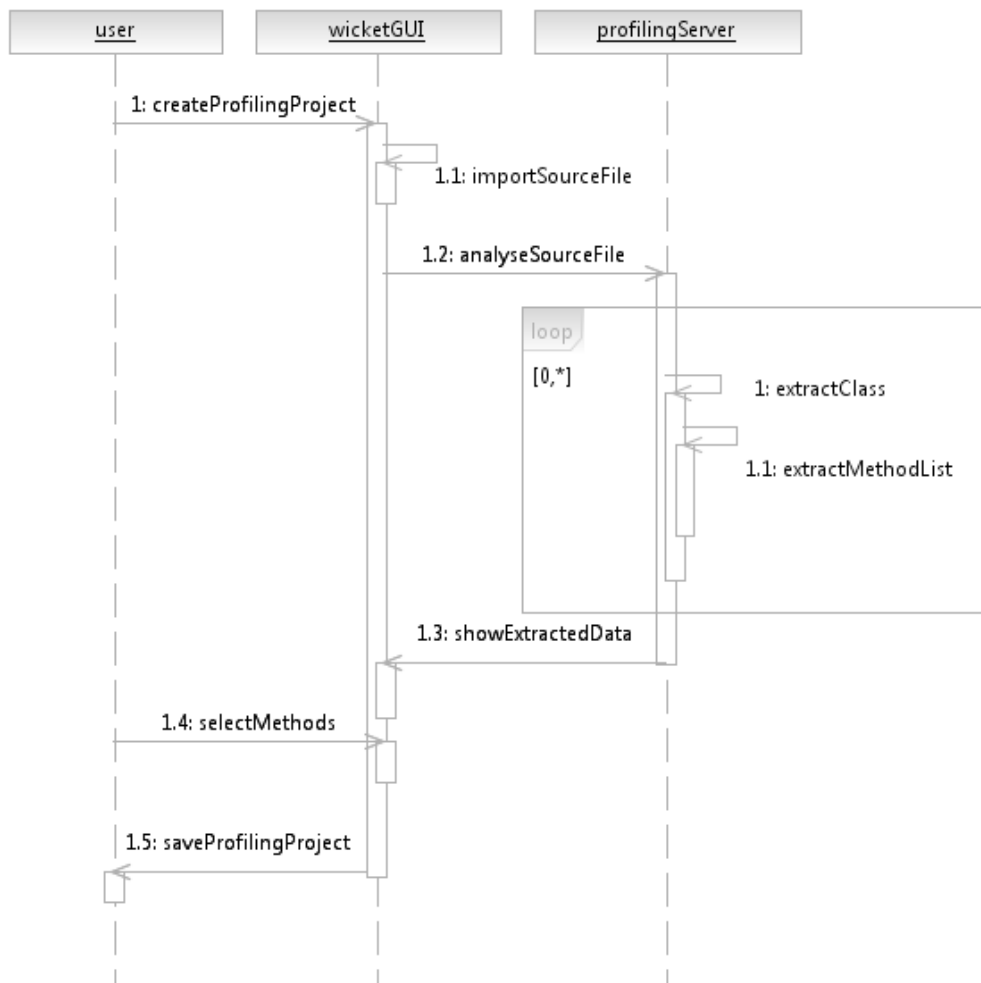
In this section, we will focus on features that are specific for our profiling solution and will bring new light to the wide field of java application profiling.

### Profiling Scenarios

The concept of user defined profiling scenarios is essentially extremely simple. Users using our profiling solution will be able to select list of classes and methods that will be instrumented and

profiled. In other words, they will be able to select exact application flow, or as we call it, profiling scenario, they want to analyse. So how do we want to implement it?

At first, we need to create list of classes and methods, from which user can specify profiling scenario. For this purpose, user will be asked to import source files of analysed application in form of .jar or .war file. Afterwards, our solution will perform analysis of this source files. Using java reflection API (`java.lang.reflect`) we are able to dynamically load classes from .jar/.war file and extract list of Class objects and list of Method objects for each Class. This information is then shown to user in user interface, where he can select analysed classes and methods. After user submits list of selected classes and methods and launches profiling, our solution will prepare and send information about profiling scenario to profiling java agent. Agent, now knowing about profiling scenario, will analyse only selected methods and classes. Since profiling scenario always works with some source code (.jar/.war file), it is always bound to specific profiling project, but profiling project can work with several profiling scenarios. Let's go through this concept once again using sequence diagram in figure 11.



**Figure 11:** Profiling scenario concept



## Profiling levels

The concept of profiling levels is essentially simple as well as the concept of profiling scenarios. Using profiling levels, user is able to select what attributes of profiling he is interested in. Profiling aspects:

- Memory profiling,
- thread activity profiling,
- class summary profiling,
- CPU time profiling,
- method time profiling.

After this selection is complete, instruction packet will be sent to profiling agent and only data relevant to selected profiling aspects will be collected, thus minimizing overall profiling overhead on analysed application.

## Method call filters

In section 2.1, we have stumbled upon an interesting group of problems. There often exist methods, whose functionality depends upon parameter type or value. These methods often works with certain easily extensible group of objects, but when it comes to profiling, we only want to perform analysis of this method in context of interaction with certain parameter type or value. Method call filters are the easiest way to achieve this. Our solution offers two basic types of method call filters (other filter types may be added in future):

- Filter based on parameter value defined by user when creating profiling scenario,
- filter based on parameter type defined by user when creating profiling scenario.

This filtering may be performed on profiling agent or on profiling server while evaluating profiling data received from agent. This choice depends on user as well. If user is completely certain, that he will only need certain type of profiling data, agent method call filtering option may be selected. This action ensures minimal data transfer between agent and server, but it also means, that user will not be able to define another method filter when viewed profiling data in graphical user interface.

Method call filter application can be also done in the phase of profiling data evaluation on server. This will lead to higher data transfer between agent and server, but user will be able to define different method call filter when viewing evaluated profiling data. We will leave this option to user. Furthermore, during implementation phase of this project, set of performance

tests was performed. Test results are discussed in section V and detailed test results can be seen in appendix C.

### **3.3 Thin Client and User Interface**

As stated in previous section, thin client architecture in context of profiling application can be considered as rare solution. We believe that this approach brings several advantages that can't be achieved by standard rich client profilers.

- Hardware and software infrastructure unification – it is not an unusual situation, when an application in development behaves perfectly on your computer, but acts completely different on your colleagues' machines. These problems can make debugging and performance tuning a very unpleasant task. In case of thin client approach, all developers can perform profiling of their code on unified machine that can be easily transformed to simulate customers' environment. If a developer is not satisfied with HW/SW configuration, on which profiling server is running, profiler deployment to localhost can be performed easily,
- multiple user parallel profiling – most thick client profiling solutions are able to perform parallel profiling of multiple java applications, but using web application as profiling tool brings the advantages of using sessions, so multiple users from multiple machines (from local network, or from the other side of the globe,...) are able to perform multiple parallel profiling tasks (Although these profiling would probably be quite inaccurate due to high load caused by high number of profiling tasks),
- simple integration with java enterprise web applications – one of the main goals of this project is to create a profiling tool able to perform performance analysis during execution of analysed application in customer and production environments as well as in test environments. The best approach to do this is to integrate profiling tool with your enterprise web application.

#### **Web container and project management**

Before developers are able to perform profiling tasks with their applications, our solution needs to be deployed using a web container or an application server. The usage of full featured application server for purposes of our project would be a huge overkill. For this purpose, web container Apache Tomcat will be used. Apache Tomcat is an open source software implementation of the Java Servlet and Java Server Pages technologies developed by Apache

Software Foundation. Before the project can be deployed in our web container, it needs to be build and packed into .war file (web application archive). For this task and for overall project management we will use Apache Maven. Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation form from a central piece of information. Both these technologies were chosen, because we have some experiences with their usage in context of bigger open – source software product and both solutions have properties required by the nature of our project.

## **Apache wicket**

Next step on a road to successful completion of established objectives is choosing the best technology for the task of creating graphical user interface. We will need a technology to connect presentation layer with our profiling engine.

A brief survey of a field of java web application development frameworks shows us many alternatives:

- Apache Struts, Apache Struts 2,
- JSF (Java Server Faces),
- Spring MVC,
- Apache Wicket,
- GWT (Google Web Toolkit),
- Apache Tapestry, ...

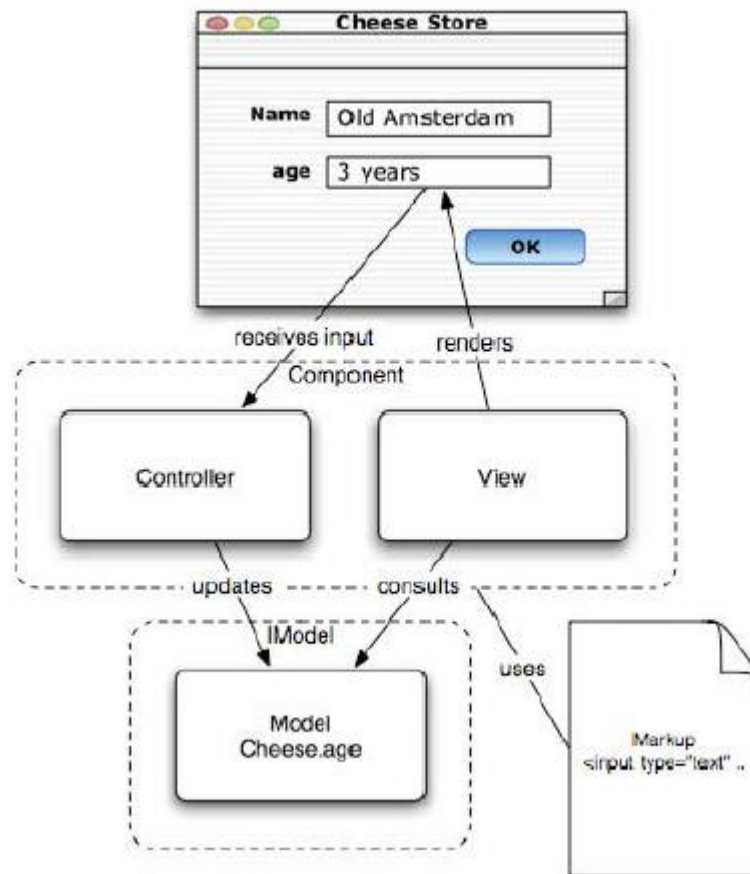
The list could continue for another half of a page, but we believe, that reader of this text already has an idea, that in the field of java web application development, there is no lack of good frameworks. Therefore, the task to choose the most suitable one will not be easy. Let's establish the set of attributes that our framework should possess:

- Component based – nature of created application indicates significant use of forms and other components, which could be reused thorough the application,
- form validation support,
- native Ajax support,
- java code outside of presentation layer,
- steep learning curve – by learning and working with selected framework, we should be rewarded with gained skill and experience as much as possible, ...

After performing few experiments with several java web application frameworks, we have chosen framework that seems to be the most suitable technology for our project. We have chosen Apache Wicket framework for these reasons:

- POJO Component model – individual pages are represented by real java objects supporting encapsulation, inheritance, events and other object oriented principles. [13] Created components can be reused very easily,
- separation of presentation and java code – for every html page, there is java file containing java components. Minimum interaction with markup code is required,
- native Ajax support components working properly,
- native form creation mechanisms and validation mechanisms,
- MVC based framework (more in Figure 12),
- allows to easily create state-full application over stateless HTTP protocol,
- steep learning curve, lots of tutorials, books and other study materials,
- large and supportive community.

We have chosen Apache wicket framework because it enables us to create web applications with advanced functionality using mostly java code, without very deep knowledge of web design or web development. We also find it very easy to debug wicket based web applications, since wicket provides very useful error messages. We will integrate this profiling tool with MidPoint open-source identity management system. GUI of MidPoint is also implemented in Wicket, thus the choice of Wicket will provide more easiness in later integration. For reasons stated in this three section and many others, we believe that Apache Wicket web application framework is the best choice for our project.



**Figure 12:** Wicket Model – View – Controller implementation. Wicket components represent View and Controller parts, using classic html code as identifiers. These components are linked with Model part represented by arbitrary java object [12].

Since our profiling solution will be later integrated into larger open – source system using already created GUI and because of lack of deeper web development skills and experiences on the side of an author of this thesis we have decided to choose already existing template and join it with our profiling engine instead of spending significant amount of time on design and implementation of a custom web design from scratch. This approach will saves a lot of time, which can be invested into creating even better profiling engine. Chosen web template will still need many changes and modification to perfectly suit our solution and more importantly

### 3.4 Technology Summary List

During solution design of our profiling application, we have discovered the need of usage for following list of APIs and frameworks:

- Apache Tomcat, web application container,
- Apache Maven, project management tool,
- Apache Wicket, web application framework,
- attach API (`com.sun.tool.attach`),
- instrumentation API (`java.lang.instrument`),
- java sockets API (`java.net`),
- Javassist API (dynamic bytecode injection),
- java Simon, simple but powerful monitoring API,
- java management extensions (JMX, `java.lang.management`),
- java reflection API (`java.lang.reflect`).

This technology list may not be final, as stated in the beginning of this section, but if there are any changes to occur, we expect that these changes will be minor. Most of the presented technologies and APIs should prevail and together build our profiling solution.

## 4 Implementation

---

In this section we will focus on implementation phase of our profiling solution. We will discuss the implementation details of all major aspects of developed profiling tool. Since the scope of this project is quite extensive, we will not focus on every implementation detail and every line of written source code. Technical description of developed profiling tool can be found in Appendix A.

### 4.1 Profiling Agent Implementation

From functional point of view, profiling agent is the most important part of every profiling tool and our solution is not an exception. We can say that we spent most of implementation time on development of our agent and profiling logic in general. Java profiling agent is basically a small group of classes, which together performs simple set of tasks necessary for every profiling tool. An abstract concept of profiling with usage of java agent can be seen in figure 11. In this section, we will look closely on details of agent implementation. At first, we need to connect our agent to JVM, where analysed application is running. We can complete this task in two ways. The first is described in section 1. Just to sum up, we can start JVM process with agent already attached:

```
java -javaagent:agent.jar[=options] foo(.class) [-jar foo.jar]
```

Second, most common way especially in context of profiling enterprise java applications is dynamic attachment during runtime of analysed application. For this purpose, java Attach API needs to be used:

```
VirtualMachine vm = VirtualMachine.attach(pid);
```

Where variable `pid` represents `String` or `VirtualMachineDescriptor` form of JVM process that we want to analyse.

Now, when our agent is attached to analysed JVM, we need to inject effective profiling code. First, we need to take every loaded class and verify, if this class meets user requirements (profiling scenarios etc.), then we perform method extraction for each class and insert pieces of active profiling code before and after this each method of loaded class. All this is done using special methods. Depending on agent injection method, we can use `premain()` method (agent connected on process start) or `agentmain()` method (agent attached dynamically). For these purposes, we are using java instrumentation API (`java.lang.instrument`) and `Javassist`

API. To better understand these principles, take a close look in Figure 13, where we present strip of code of our profiling agent. After profiling is done, classes in form before code injection are loaded back to JVM and profiling agent is detached from JVM.

```
@Override
public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
                        ProtectionDomain protectionDomain, byte[] classFileBuffer) throws IllegalClassFormatException {

    //...

    try{
        classPool.insertClassPath(new ByteArrayClassPath(className, classFileBuffer));
        CtClass cc = classPool.get(className.replace("/", "."));
        CtMethod[] methods = cc.getMethods();

        //..

        for(int i = 0; i < methods.length; i++){
            methods[i].insertBefore("injected code ...");
            methods[i].insertAfter("injected code ...");
        }

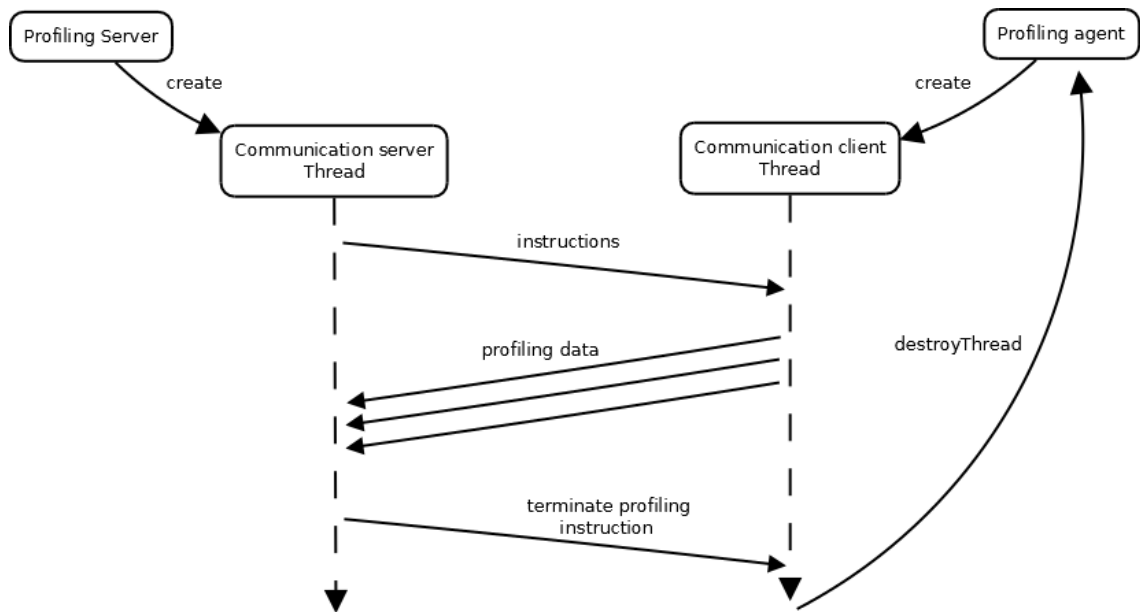
        byte[] newClassfileBuffer = cc.toBytecode();
        return newClassfileBuffer;

    }catch(Exception e){
        //...
    }
    return null;
}
```

**Figure 13:** Class loading transformation technique

The last piece of puzzle in context of profiling agent functionality is communication with server and profiling data transfers. For this purpose, we are using java sockets. Profiling server represents socket server and analysed JVM represents socket client. Parallel connections of socket clients are enabled. For each communication, separate thread is created on agent as well as on profiling server and both threads are destroyed after profiling is no longer active. These threads are responsible for sending instruction packets (server => client) and profiling data packets (client => server). Simple communication scheme between profiling agent and profiling client can be seen in Figure 14.





**Figure 14:** Profiling agent ↔ profiling server communication scheme

## Method call profiling

Method call profiling is deeply explained in several places in this document. This functionality has been implemented using event-based profiling technique. After profiling agent is attached to target JVM and bytecode is dynamically injected before and after methods (explained above), this code is called every time specific method call event occurs and thus we are able to catch method call run times. Method call profiling can be specialised using user defined profiling scenarios and method call filters.

## CPU usage profiling

In context of CPU profiling in our monitoring application, we monitor these performance parameters of CPU usage:

- Average CPU usage – percentage expression of consumption of CPU computing capacity,
- overall elapsed time – computing time elapsed since our agent has been attached to analysed JVM (I/O operation times and other non CPU application activity is counted here),
- overall CPU time – amount of time application spent using CPU computing capacity.

For these purposes, we used Java management API (`java.lang.management`), specifically `OperatingSystemMXBean` and `RuntimeMXBean` objects, which provided necessary performance data.

```
OperatingSystemMXBean operatingSystemMXBean = (OperatingSystemMXBean)
ManagementFactory.getOperatingSystemMXBean();

RuntimeMXBean runtimeMXBean = ManagementFactory.getRuntimeMXBean();
```

## Memory usage profiling

In terms of memory profiling, we profile these performance attributes:

- Heap memory usage – used, free and maximum available memory at specific time (these attributes are measured in bytes),
- Non – heap memory usage - initial, maximum, committed and used memory pool size at specific time (measured in bytes as well),
- Garbage collection activity – objects pending finalization count and garbage collection runs.

We are able to extract all these data using `System.Runtime` class and Java management API (`java.lang.management`), specifically `MemoryMXBean` and `MemoryUsage` objects.

## Thread activity profiling

For thread activity profiling purposes, we monitor these attributes in specific moment of analysed application execution:

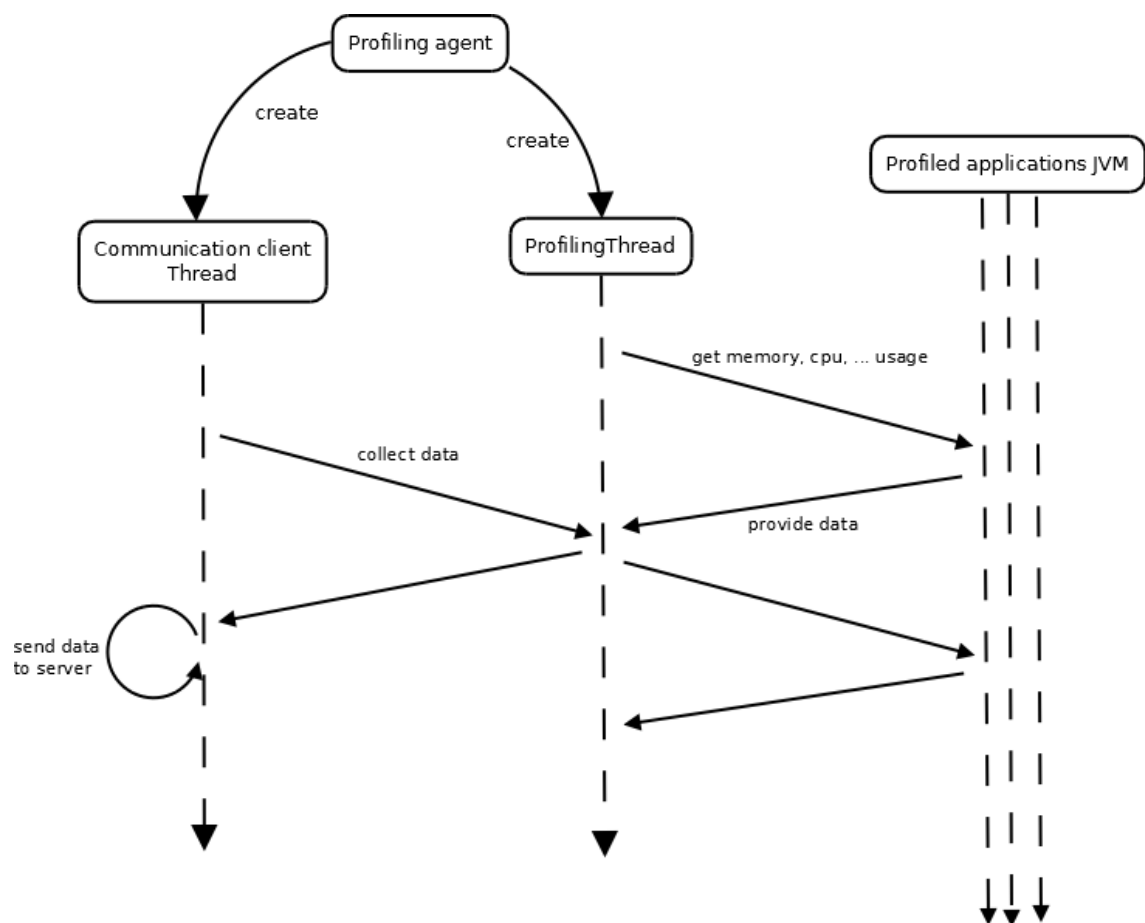
- Current thread count,
- current thread ids,
- current thread names,
- stack traces for current threads and further information about all current threads.

Again, we used Java management API (`java.lang.management`) for thread activity profiling, specifically `ThreadMXBean`.

## Class profiling

In specific time of execution of profiled application using our profiling tool, we also monitor count of current loaded classes and their names. For these purposes, we used Javassist API, specifically `Instrumentation` object and `getAllLoadedClasses()` method.

We used sampling approach when implementing CPU usage, memory usage, thread and class activity profiling. This technique enables us to interrupt analysed application in time intervals (which can be defined when creating custom profiling level) and collect all wanted data. Enhanced profiling scheme in context of thread activity between profiling agent and profiled java application can be seen Figure 15 (This figure loosely ties to figure 14).

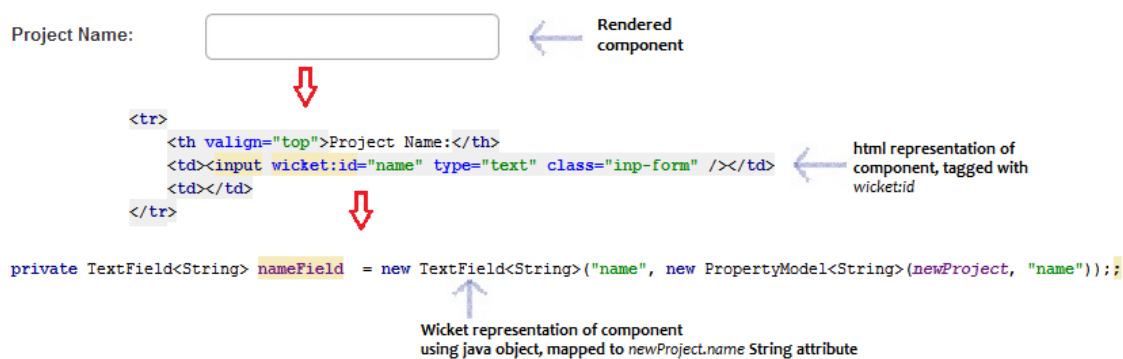


**Figure 15:** Profiling agent thread activity scheme during application profiling.

## 4.2 Graphical User Interface Implementation

As we mentioned in previous sections, we wanted to invest as much time as possible to creation of meaningful, powerful and effective profiling engine, but the presence of eye catching and neat graphical user interface is an important indicator, if developed application will be successful and used. We decided to implement our GUI using open source web application template. Finding template appropriate for requirements of our profiling tool was not easy, but after long search we managed to find perfect web application template. We have decided to work with free admin skin found on [www.netdreams.co.uk](http://www.netdreams.co.uk) [14].

Of course, this was just the beginning of GUI implementation, since we needed to create connection between presentation layer and profiling engine written in java. As stated in previous section, we used Apache Wicket web application framework for this purpose. Working with this framework was very pleasant, as every dynamic web component is represented by wicket component object, which can be easily mapped to arbitrary application data. To illustrate work with wicket and to show reader, how exactly our GUI works, we decided to include an example of interaction with one component from our application. It is a classic text field component that represents name of created profiling project. In markup code, it is represented by classic `<input>` tag with text type. The important part is the `wicket:id` attribute that describes, on what java component it is connected. In java code, we managed to map this component to `ProfilingProjectType` object named `newProject`, specifically `String` attribute „name“. The scheme of this work can be seen on figure 16.



**Figure 16:** Wicket GUI functionality explained

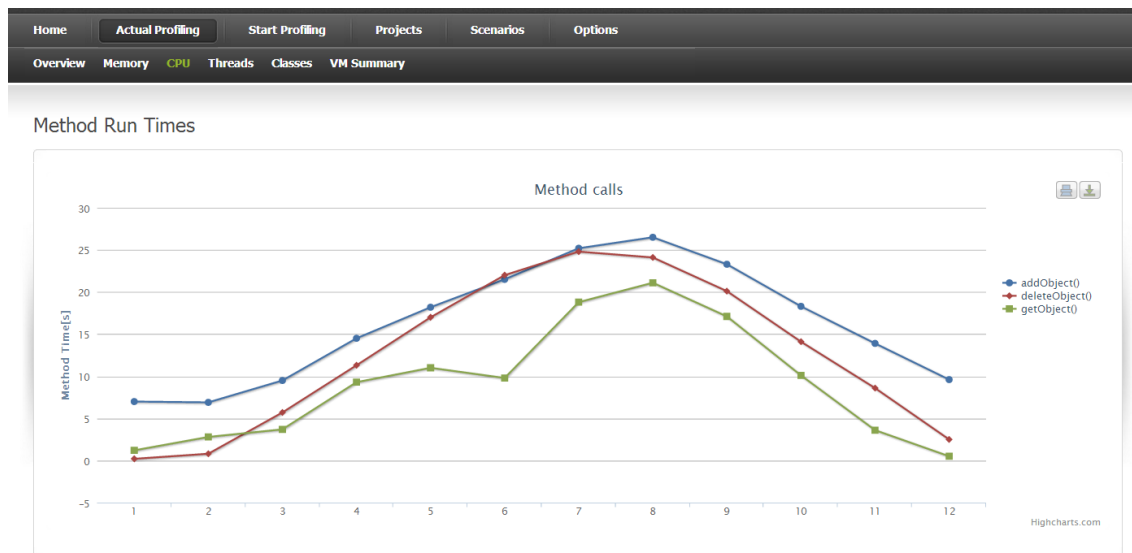
Using this approach and other features of wicket framework, we have managed to create GUI visually and functionally suitable for our profiling tool. The example of looks of our application can be seen on figure 17.

The screenshot shows the 'New Project' form in the Profiler application. The form is organized into several sections:

- Project Name:** A text input field.
- Description:** A larger text area for project details.
- CPU Settings:** A dropdown menu set to 'Classic'.
- Memory Settings:** A dropdown menu set to 'Classic'.
- Profiling Scenario:** A dropdown menu set to 'none', with 'Edit' and 'Create' buttons below it.
- Insert JAR:** A text input field with a blue button labeled 'Insert jar file to be profiled' to its right.
- File option:** A checkbox labeled '(Save whole jar file (only jar path is saved, when this option is not selected))' which is currently checked.
- VM Options:** A text area for specifying JVM options.
- Attach option:** A checkbox labeled 'Attach on start (if not selected, agent will attach after program start)' which is currently unchecked.
- Submit:** A green button at the bottom of the form.

**Figure 17:** GUI appearance of our profiling tool.

We are aware of importance of neat graphical user interface. Profiling tool can be mastered in way of used profiling features, but if it fails to present gathered and evaluated data or if it is hard to read in this data, what directly complicates recovery of performance problem, tool is sentenced to fail. For these reason, we have worked hard to find the best way to show users collected profiling data and we finally decided to use Wicket Charts framework [15]. We think that showing profiling data in form of charts brings even more clarity and easiness into finding that specific memory leak or other performance problem. Charts used in our solution are automatically updated using AJAX features. These charts also provide various ways of interactivity with users, like data enlargement, specific data values in every chart point, direct chart print or export into image file etc. Example of chart directly from profiling can be seen on figure 18.



**Figure 18:** Chart representation of gathered profiling data.

### 4.3 Profiling Scenarios Implementation

The concept of user defined profiling scenarios is explained in section 3.3. To sum up, main idea of this concept is to grant user functionality to select the list of classes and methods that would be profiled. The first implementation step was the analysis of user provided set of classes in form of inserted .jar/.war file (this can be easily extended into list of class files, maven or ant project). To analyse inserted class files, we need to load them to profilers underlying JVM. Dynamic class loading is not an easy task, but we managed to complete this challenge using java reflection API (`java.lang.reflect`), specifically, we used simple reflection hack, where we instantiated `URLClassLoader` method `addURL` after changing its visibility from protected to public. This step was necessary to load classes from URL that were extracted from .jar/.war file. The exact code strip can be seen below:

```
private static final Class[] parameters = new Class[] { URL.class };
JarFile jar = new JarFile(jarFile);

URLClassLoader sysLoader = (URLClassLoader)ClassLoader.getSystemClassLoader();
Class sysClass = URLClassLoader.class;
Method method = sysClass.getDeclaredMethod("addURL", parameters);
method.setAccessible(true);
method.invoke(sysLoader, new Object[] {jarFile.toURL()});
```

After this analysis phase, results are shown in GUI, where user can simply select, which classes and methods are going to be profiled. After selection is complete, the creation of profiling

scenario is complete and user is redirected back to profiling project configuration page. The important thing to say in context of profiling scenarios is that every profiling scenario must be bound to concrete profiling project, since it is built from analysis of set of java classes and this set is bound to profiling project. One profiling project can work with multiple scenarios and the user is able to select current working scenario before profiling is launched. After this action the communication is started between profiling server and target JVM (see section 4.2 for details). First packet that is sent from profiling server contains information about selected profiling scenario. Agent receives this packet, extracts important data and takes them into account in the phase of dynamic bytecode injection (see section 4.2). Profiling agent performs bytecode injection only in case when transformed class or method is present in the received list of profiled classes and methods (in other words, if it is in profiling scenario). Classes and methods not present in this list are not instrumented, thus profiling is not executed. In figure 19, we can see an example of profiling scenario creation in GUI. The screenshot shows one class `test.Help` containing two methods, specifically `printHelloWorld()` and `doSomething()` methods. Other methods are present, because in java, every class extends `Object` class in default. User has an option to select, if these methods will, or will not be shown during profiling scenario creation. Another interesting functionality that may be implemented in the future (but is not in scope of this bachelor thesis) may be the creation of complex inheritance schema of provided source files.



**Class:** `test.Help`

**Methods:**

- ☐ `printHelloWorld`
- ☐ `doSomething`
- ☐ `wait`
- ☐ `wait`
- ☐ `wait`
- ☐ `equals`
- ☐ `toString`
- ☐ `hashCode`
- ☐ `getClass`
- ☐ `notify`
- ☐ `notifyAll`

**Submit**

**Figure 19:** User defined profiling scenario creation example

## 4.4 Profiling Levels Implementation

As mentioned in previous sections, profiling levels are used to define which aspects of performance we want to examine during profiling. The list of profiling aspects can be seen in section 3.3, so we will not state it here. Profiling levels are not bound to specific profiling object, so we can use one level with multiple projects in contrast to profiling scenarios. In addition to the ability of choosing, which performance aspects will be monitored, we also give user the power to define intervals of examination of these aspects.

This functionality gives users great power and it needs to be used very cautiously. There exists a trade-off between profiling accuracy and efficiency, let's explain this on simple example. User has created profiling level, where he wants to monitor CPU and thread activity in 20 milliseconds time intervals, in other words, 50 times a second. Every 20 milliseconds, agent will gather information about current thread and CPU activity, but 20 milliseconds is enormous amount of time for computer. Several hundred thousand operations have been executed during that time, but we only have profiling data from precise time moment, so we can only assume, that retrieved stack trace or CPU usage has been there for entire time interval. Of course, in evaluation phase, we can use statistics methods to interpolate values between time intervals, but exactness of this method is very imprecise. To collect precise data about CPU and Thread activity, we would need to perform analysis several times every millisecond. Of course, gathering these information takes time and compute resource. To sum up, it depends on every developer, how precise profiling data he needs and how big profiling overhead can he tolerate in order to collect this data.

Memory and CPU usage, thread activity and class state examination are all implemented using sampling technique. More precise scheme of this process can be seen on figure 15. On figure 20, we can see profiling level creation in graphical user interface.



**Add/Edit Profiling Level**

Level Name:

Description:

Methods: ☐ Method call profiling

CPU: ☒ CPU profiling

Interval[ms]:

Memory: ☐ Memory profiling

Interval[ms]:

Threads: ☒ Thread activity profiling

Interval[ms]:

Stack depth:

Classes: ☐ Class load profiling

Interval[ms]:

**Figure 20:** Creation of profiling level using profilers graphical user interface.

## 4.5 Method Call Filters Implementation

Method call filters are simple way to define, which calls of examined method are interesting for us and which are not. To do this, we need to examine parameters and their values for every single call of profiled method and based on these filters, we will decide, if data collection will, or will not be performed. In current version, we support only method calls created from basic data types and their wrapper classes. These filters can be assigned to profiling scenarios and will be applied (as well as profiling scenarios themselves) only in case, when method call profiling is selected (while creating profiling level). User can create several method call filters for every investigated method and define desired values.

In future versions of this tool, more precise method call filtering mechanism will be implemented containing functionality to pass expression language as method a parameter filter (both for parameter value and type), but this is not in the scope of this bachelor thesis.



## 5 Evaluation

In this section, we will focus on testing and evaluating implemented profiling solution. Testing is necessary and extremely important phase in software development lifecycle. For testing purposes, we have chosen several approaches:

- manual testing of chosen test scenarios,
- profiling overhead testing with for this purposes designed application,
- profiling overhead and general testing with midPoint
- Automatic testing with unit tests,

### 5.1 Manual Tests

We have created several test scenarios in several categories that were used to evaluate functionality of designed product and its responses to user actions. We tested CRUD (create, read update, delete) operations with all important objects connected to profiling mechanisms, specifically profiling scenarios, profiling levels, profiling projects and method call filters. Tests were designed by classic test scheme. First, we wrote down steps needed to take to perform test scenario. Next we wrote down expected result. A test is considered accomplished when real response of our application is exactly the same as expected test result. In other case, when real test results are even slightly different from expected results, the test is considered as failure. We also added several tests that should lead to negative response from implemented profiling tool. For example, when creating profiling project, we need to provide name for this project as well as source code form in .jar/.war file. If these conditions are not met, profiling project should not be created and user should see comprehensible error message that will tell him, what is wrong. In figure 21, we can see example of few test scenarios along with their expected and real results. More manual test scenarios can be seen in appendix C.

#	Test Scenario description	Expected Result	Test result
1	Add profiling level: 1. In GUI, click Options, then Add profiling level. 2. Fill level name, description and select arbitrary profiling aspects and their intervals. 3. Click on Submit button	Profiling level should be added to repository and should be seen on Profiling levels page.	✓
2	View all profiling levels: 1. Click Options, then Profiling levels tab.	List of all existing profiling levels should be displayed	✓

**Figure 21:** Example of test scenarios. More can be seen in appendix C.

## 5.2 Profiling Overhead Testing

In context of dynamic java application profiling, one of the most important profiling tool attributes is its profiling overhead. Profiling overhead is discussed multiple times in this document. In this section, we will focus on testing profiling overhead of our implemented profiling solution. Profiling overhead testing and measurement were made on java application specifically implemented to server test purposes. This application obtains several methods, specifically:

- `doSomething()` method – this method contains large number of numeric computation,
- `memoryTest()` method – this method contains large number of memory operations, specifically work with `ArrayList` containing one million `String` objects of length one hundred. This `ArrayList` is refilled five hundred times during every method call.

These tests were performed on following HW/SW configuration:

- Operating system – Windows 7 6.1 64 bit,
- JIT compiler – HotSpot 64-Bit Tiered Compilers,
- JVM – Java HotSpot 64-Bit Server VM version 20.9-b04,
- Vendor – Sun Microsystems Inc.,
- CPU – Intel Core i5-2500K CPU,
- number of cores - 4
- RAM – 8,00 GB

To minimize measurement inaccuracies caused by other operation system processes, we performed these measurements one hundred times and calculated average values, but we still experienced a lot of fluctuation in test results, thus the accuracy of these tests is not 100%. It should still give reader very clear information about designed profiling solution overhead. Example of test results can be seen on figure 22.

#	Method	Memory [ms]	CPU [ms]	Thread [ms]	Class [ms]	#1[s]	#2[s]	#3[s]	Overhead [s]	Overhead [%]
1	-	-	-	-	-	2,735	4,770	170,10	-	-
2	Y	1000	1000	1000	1000	2,737	4,832	171,40	1,3	1,007
3	Y	100	100	100	100	2,745	4,847	171,86	1,76	1,010
4	Y	10	10	10	10	2,750	4,979	174,61	4,51	1,026

**Figure 22:** Profiling overhead testing and measurement results.

In table above, we can see overall measured profiling overhead results. This table needs some explanation:

- In first row, we can see values for raw execution of tested application, in other words, without our profiling agent attached,
- values in Memory, CPU, Thread and Class columns means profiling interval,
- value in Method column means, if method run times were monitored,
- #1 – `doSomething()` method average execution time,
- #2 – `memoryTest()` method average execution time,
- #3 – tested application average execution time,
- overhead – average overhead, slowdown in comparison of raw execution represented in seconds and percents.

More detailed table containing test results can be seen in appendix C, specifically C.2

### 5.3 Tests with MidPoint

Another step in testing phase is testing with midPoint. MidPoint is an open-source identity management tool. The main purpose of midpoint is synchronization of several identity repositories, their management and presentation in unified form. Developed profiled tool will be later integrated with midPoint, specifically in release 2.3. Current stable version of midPoint is 2.1; version 2.2 will be released in June 2013.

We tested our profiling solution with midPoint on the same HW/SW configuration as stated in previous section. We used Tomcat 7.0 web container and LDAP server OpenDJ for these tests. During development of this product, we prepared several unit tests using TestNG, mostly focusing on performance testing, for example:

- Adding large number of users to repository,
- deleting large number of users from repository,
- listing large number of users from repository,
- parallel adding large number of users to repository in several threads, etc.,

During testing with midPoint, we hooked our agent directly into Tomcat web container, collecting and evaluating data from execution of these tests. We tested use of profiling scenarios, profiling levels and method call filters and measured profiling overhead generated by dynamic performance analysis. All test results were positive and we believe that this profiling tool is ready to be integrated with MidPoint.

## 5.4 Automatic Tests

Another significant approach to testing software products are automatic tests. In java, unit tests are used for this purpose. We have implemented automatic tests in our solution using maven project management tool and `TestNG` framework, which provides similar functionality as `JUnit` framework with several new features.

Scenarios in automatic tests are not that different from scenarios used in automatic testing. Main difference is, that in manual tests, living person needed to use applications GUI to perform actions defined in test scenarios, while in automatic tests, these test scenarios are written as standard java methods with corresponding annotations and no interaction with GUI is needed, we simply call specific methods to complete test scenarios (same methods are called in manual testing, except they are invoked by user interaction with GUI). For automatic tests, we also need to prepare set of test data, for example create necessary profiling objects. We can create them programmatically in code of current test code, or prepare them in form of XML and parse them during test execution.

## 6 Conclusion

---

This bachelor thesis focused on profiling and creating specific profiling tool that would provide functionality to focus on analysed java application in very detailed and specific way. In first sections, we spoke about profiling domain in general. We introduced existing approaches to profiling and existing techniques in this field. We also described existing profiling solutions and tools. We analysed standard Java API profiling tools, open – source solutions and commercial applications used for purposes of dynamic code analysis. Analysed tools did not meet our requirements, thus we decided to create own profiling tool.

In next sections, we analysed requirements and specified objectives of this project. Later, we designed profiling application using event-based and sampling profiling techniques. We introduced concept of profiling scenarios, an easy way to define, which methods of examined application were analysed. This feature was not enough for simple reason, we only wanted to profile specific method calls and thus we created method call filters technique. This technique combined with profiling scenarios enables us to profile exactly what we want. Another introduced technique, yet not new as previous concepts, was usage of profiling levels that allows us to select, which profiling aspects (method call examination, CPU, memory usage, thread and class activity) we want to analyse. We enhanced this concept with user defined profiling intervals.

We also implemented the proposed solution. Application was implemented using client – server architecture. We discussed implementation in detail in section 4. Implemented solution was tested properly. Specifically, we performed series of manual tests based on created test scenarios. Some of these scenarios were also implemented as automatic unit tests. We focused on testing general profiling overhead. For this purposes, we performed exhaustive number of performance tests which results showed us small percentage of profiling overhead. Finally, we tested this solution with MidPoint. All test results satisfied our requirements.

There is still a lot of work on implemented profiling tool and it can be (and will be) enhanced in many ways, the most important ones being these:

- More dynamic user interface, AJAX integration,
- XML form of repository profiling objects,
- usage of expression language with in method call filters,
- automated creation of profiler home directory (deployment enhancement),
- remote application profiling over TCP/IP,
- implementation of special file format allowing us to dump result of profiling.

The work on this project is far from being over. It will be integrated with open-source identity management solution midPoint in one of next stable releases (2.3).





# Bibliography

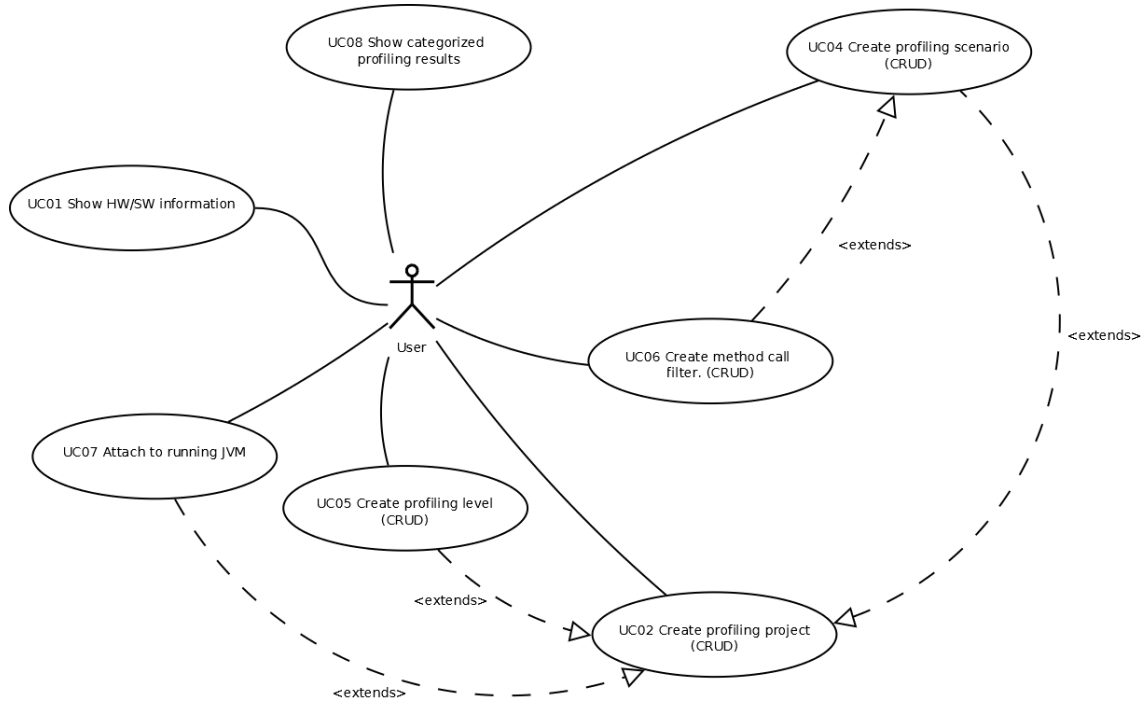
---

- [1] BROWNE, J.: Brewer's CAP Theorem. julianbrowne.com, 2009, [quoted. 8. may 2013; 23.30h]. Available online (world wide web): <http://www.julianbrowne.com/article/viewer/browsers-cap-theorem>
- [2] DIEHL, S.: A Formal Introduction to the Compilation of Java. New York, NY, USA : John Wiley & Sons, Inc., 8<sup>th</sup> January 1998. 297-327 p.
- [3] HAGGAR, P.: Practical Java Programming Language Guide. Indianapolis, IN 46290 : Addison-Wesley, 4th July 2001
- [4] J.: JVMs, JDKs and JREs. <http://java-virtual-machine.net/>, 2008, [quoted. 8th May 2013; 23.47h]. Available online (world wide web): <http://java-virtual-machine.net/other.html>
- [5] B., P.: The HotSpot Group. <http://openjdk.java.net/>, 2013, [quoted. 8th May 2013; 23.50h]. Available online (world wide web): <http://openjdk.java.net/groups/hotspot/>
- [6] L. Graham, S. - B. Kessler, P. - K. McKusick, M.: gprof: a Call Graph Execution Profiler. <http://docs.freebsd.org/>, [ quoted. 8th May 2013; 23.53h]. Available online (world wide web): <http://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf>
- [7] KNUTH, D.: Structured Programming with go to Statements. University of California, San Diego : Pearson, ISBN 13:978-0-13-283031-7, December 1974. 268. p.
- [8] BURD, B.: for Dummies, 5th edition. Indianapolis, Indiana : Wiley Publishing, Inc., ISBN: 978-1-118-12832-9, 2011. 432 p.
- [9] ECKEL, B.: Thinking in Java, 3rd edition. Electronic Book : Prentice-Hall, December 2002
- [10] Apache maven project management tool, available online: <http://maven.apache.org/guides/>
- [11] Apache Tomcat web container, available online: <http://tomcat.apache.org/>
- [12] DASHORST, M. - HILLENUS, E.: Wicket in Action. Dreamtech Press, 9788177228847, 2008. 388 p.
- [13] APACHE, T.: Apache Wicket Features. <http://wicket.apache.org/>, 2013, [ quoted. 9th May 2013; 00.08h]. Available online (world wide web): <http://wicket.apache.org/meet/features.html>
- [14] DREAMS, I.: Free admin skin available. <http://www.netdreams.co.uk/>, 2013, [ quoted. 9th May 2013; 00.11h]. Available online (world wide web): <http://www.netdreams.co.uk/index.php/blog/2010/02/18/free-admin-skin-available-for-download/>

- [15] Wicket-charts JavaScript Charts with Apache Wicket and JSF. code.google.com, 2013, [ quoted. 9th May 2013; 00.14h]. Available online (world wide web): <https://code.google.com/p/wicked-charts/>
- [16] E.: midPoint. <http://www.evolveum.com/>, 2011, [ quoted. 9th May 2013; 00.16h]. Available online (world wide web): <http://www.evolveum.com/midpoint.php>
- [17] M., Bieliková.: Ako úspešne vyriešiť projekt. Bratislava : Slovenská technická univerzita, ISBN 80-227-1329-5, 2000. 158 p.

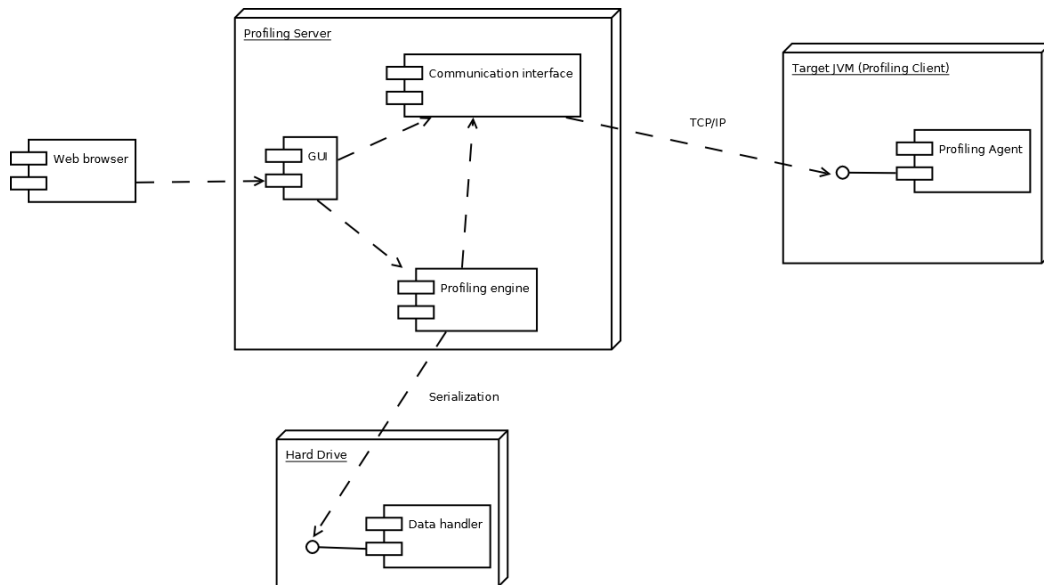
## Appendix A – Technical Documentation

In this chapter, we will provide more detailed explanation of UML diagrams attached to this document. We will also include many other UML diagrams describing implemented profiling tool. Not every single detail of provided diagrams is described. More UML diagrams are can be seen on attached CD medium.



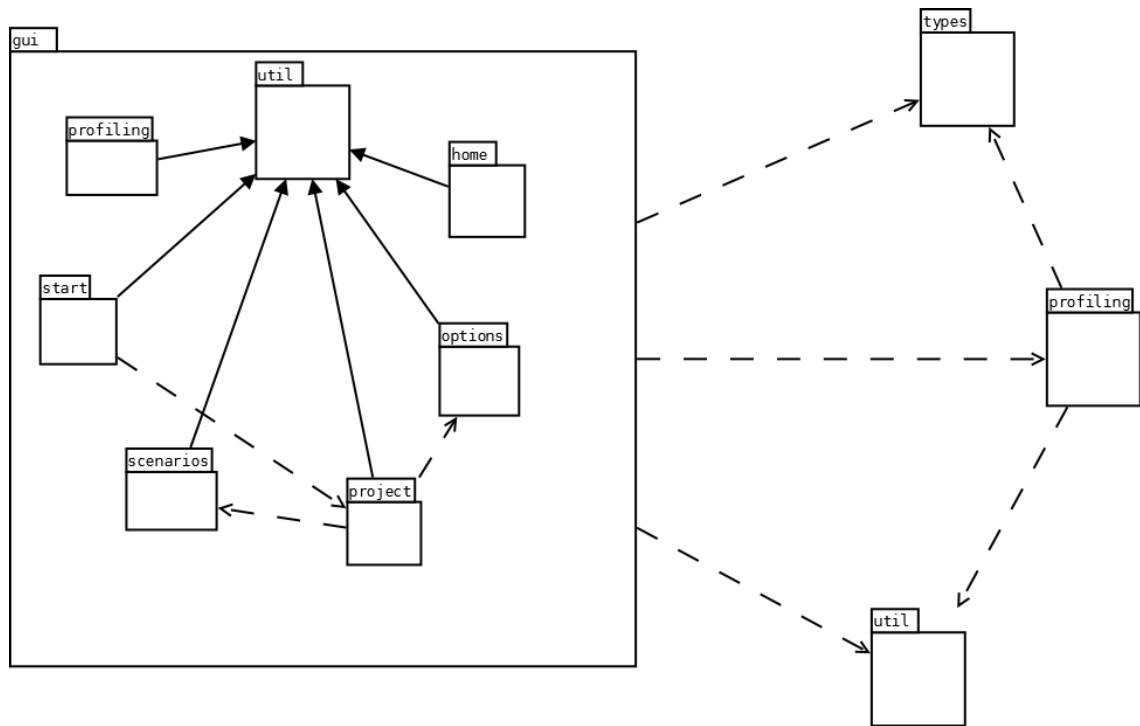
**Figure 23:** Use case diagram of implemented profiling solution. Same as Figure 8.

In Figure 23, which can be also seen in section 2.3, we provide use case diagram. It represents an output from use case analysis based on products specification and requirements analysis. This diagram shows several use cases. Every use case represents some action between user and implemented profiling solution. Use cases labelled with CRUD label are complex use cases that could be divided into four smaller use cases, specifically create, read, update and delete use case. When creating profiling project, user is able to perform several actions leading to its deep specification, for example adding profiling scenario (UC04 Create profiling scenario CRUD) or adding profiling level (UC05 Create profiling level CRUD). The creation of profiling scenario can be modified by creating several method call filters (UC06 Create method call filter CRUD). User is also able to select running JVM while creating profiling project (UC07 Attach to running JVM).



**Figure 24:** Component diagram representing architecture of implemented profiling tool.

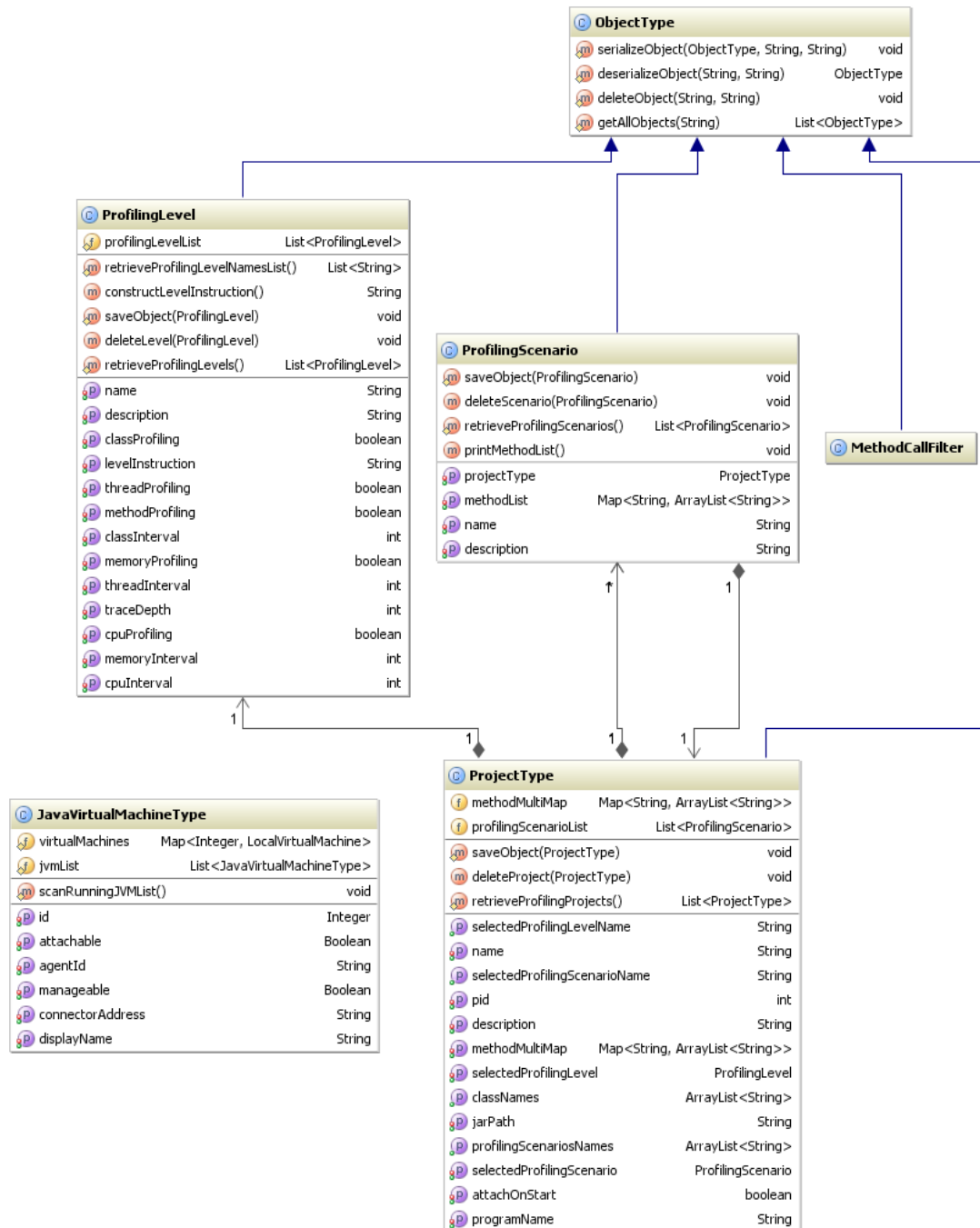
On Figure 24, we can see component diagram. This diagram is a representation of implemented profiling tool. Profiling server, the main component, is composed from three smaller components, specifically Communication interface, GUI (Graphical User Interface) and Profiling engine. Communication interface periodically communicates with profiling agent, which is a component attached to target JVM during performance of profiling. Communication protocol TCP/IP is used. Profiling engine communicates with Data handler component. This component is a representation of data layer of our solution. GUI component also communicates with Web browser component.



**Figure 25:** Package diagram containing main packages of implemented profiling solution.

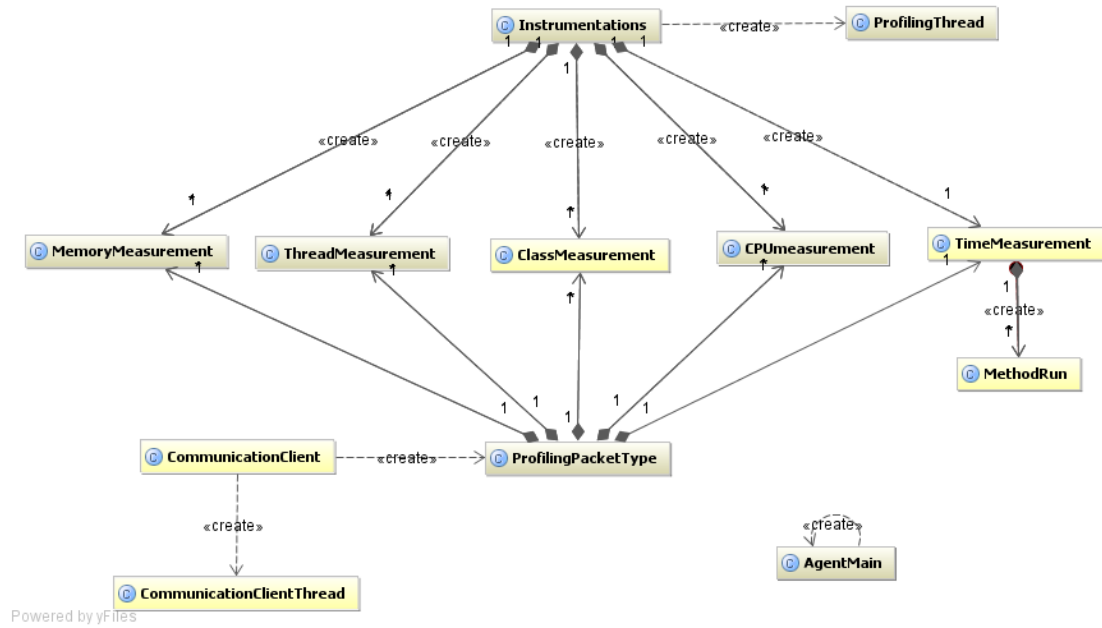
On Figure 25, we can see package diagram representing package composition of implemented profiling tool. `Types` package contains classes representing objects used in our solution. Profiling package contains profiling engine functionality. Package `util` contains utilization functionality of our application. GUI package contains seven smaller packages. Every one of these packages contains several class and html files. This package represents code structure of created GUI. We can see, that packages `home`, `options`, `project`, `scenarios`, `start` and `profiling` extends `util` package. We can also see other relations between packages. Filled arrows represent inheritance and not-filled arrows represent dependencies between components.

On Figure 26, we can see detailed class diagram containing representation of package `Types`. Classes in this diagram are representation of objects that are used in our profiling solution. We can see that every object extends `ObjectType` class. This type is a superclass containing functionality that is used by every object in `Type` package. Very detailed description of class and object attributes, constructors and provided methods can be seen on Figure 25.



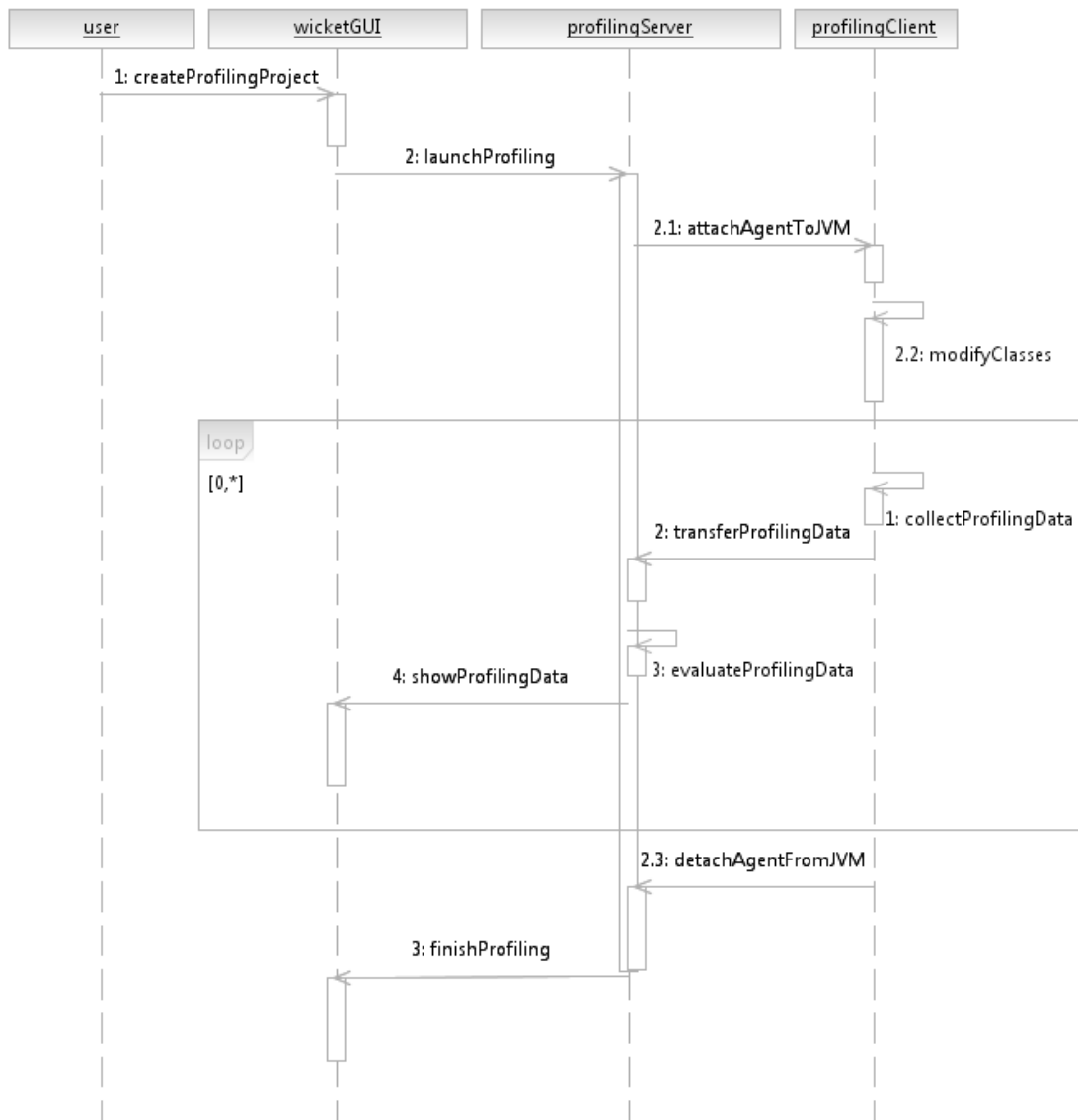
Powered by yFiles

**Figure 26:** Detailed class diagram of package Types.



**Figure 27:** Basic class diagram of profiling agent.

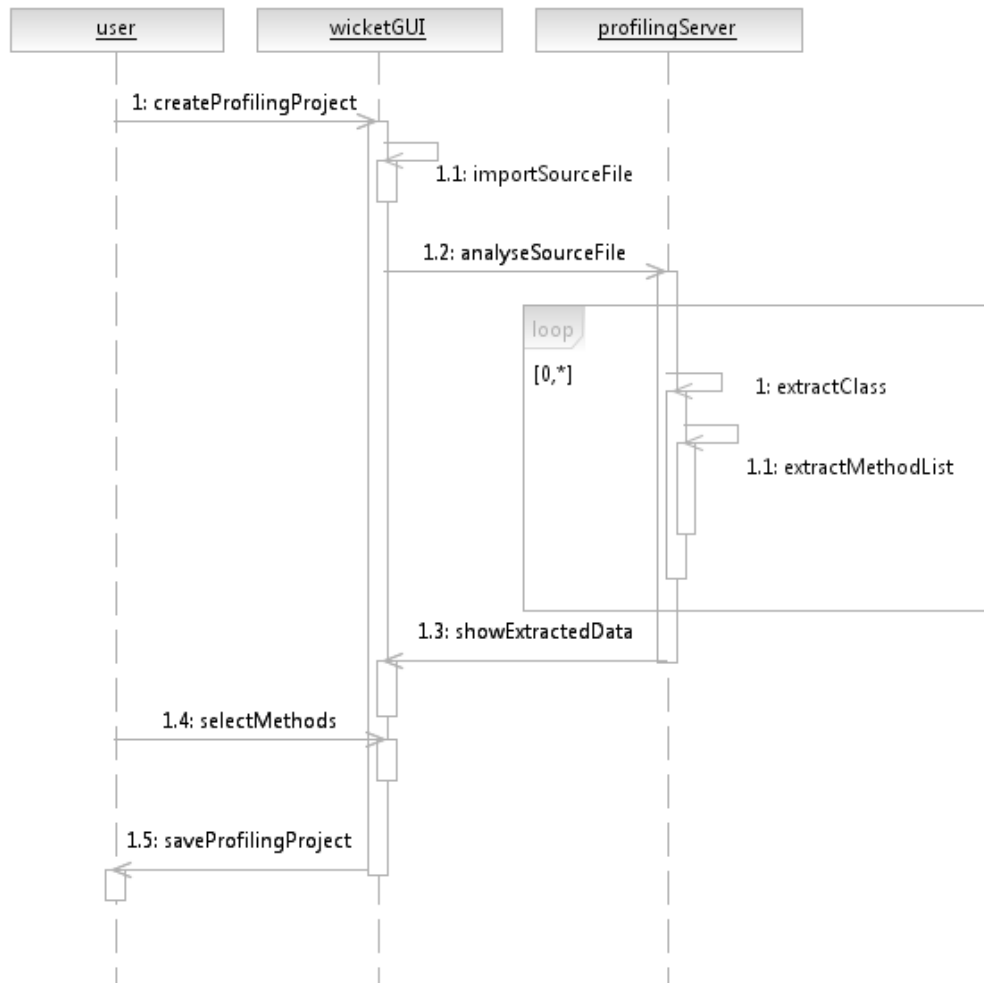
On Figure 27, we can see basic class diagram of implemented profiling agent that is used to perform analysis of target java applications. AgentMain is a main class, in which basic profiling functionality is implemented as well as dynamic bytecode instrumentation. ProfilingThread and CommunicationClientThread classes represent classes of threads used during profiling. MemoryMeasurement, ThreadMeasurement, ClassMeasurement, CPUmeasurement and TimeMeasurement are classes implementing functionality necessary to perform dynamic performance information gathering. Instances of these classes are used in Instrumentation class. ProfilingPacketType is a representation of network packet. Instances of this class are periodically sent to profiling server. They contain collected profiling data.



**Figure 28:** Abstract profiling concept described using sequence diagram. Same as Figure 10.

On Figure 28, we can see sequence diagram representing abstract profiling concept. This diagram can be also seen in section 3.1. After user created `ProfilingProject`, profiling is launched. The first step of profiling is agent attachment to target JVM. Next, compiled source code is dynamically modified by dynamic bytecode injection. Afterwards, the cycle of performance data collection is launched. During this cycle, profiling data are periodically collected, transferred to `ProdilingServer`. On `ProfilingServer` they are evaluated and shown user in appropriate form. After profiling is finished, agent is detached from analysed java application.





**Figure 29:** Profiling scenario concept.

On Figure 29, which can be also seen in section 3.2, we describe the concept of profiling scenarios. During the process of creation of `ProfilingProject`, user imports the source files (.jar/.war) containing source codes of analysed application. These source codes are then analysed. Analysis of provided source codes is composed of class and method extraction. After analysis is completed, results are shown to user in GUI and user is able to select methods and classes that are going to be analysed. After user has completed creation of `ProfilingProject`, this project is saved in repository. We would also like to introduce small part of profiling source code here. Next code is responsible for profiling thread activity. In this code example, we use `ThreadMXBean` object generated by `ManagementFactory`, `ThreadMXBean` is one of built-in performance beans in JMX API (Java Management eXtension, `java.lang.management`). Using this object, we are able to fetch information about current used threads, their identifiers, names and other usefull information, e.g. stack traces for each thread with user defined depth.

```

/* Attributes */

int threadCount;
long[] threadIds;
ThreadInfo[] threadsInfo;
String[] threadNames;

/**
 * Constructor
 */
public ThreadMeasurement() {

    ThreadMXBean threadBean = ManagementFactory.getThreadMXBean();
    threadIds = threadBean.getAllThreadIds();
    threadsInfo = threadBean.getThreadInfo(threadIds,
    THREAD_TRACE_DEPTH);
    threadCount = threadIds.length;

    threadNames = new String[threadCount];
    for(int i = 0; i < threadCount; i++){
        if(threadsInfo[i] != null)
            threadNames[i] = threadsInfo[i].getThreadName();
    }
} //Constructor end

```

## Appendix B – User Guide

---

In this appendix, we will provide simple user guide to implemented profiling solution. The purpose of this appendix is not to describe every single possibility and functionality of our tool but to provide extended image about profiling tools usage.

### B.1 Site Map

GUI of implemented solution is composed from these web pages:

- Home
  - My Last Actions
- Actual Profiling
  - Overview
  - Methods
  - Memory
  - CPU
  - Threads
  - Classes
  - VM Summary
- Start Profiling
  - Attach to JVM
  - New Project
- Projects
  - Actual profiling project
  - All profiling projects
- Scenarios
  - Actual profiling scenario
  - Create profiling scenario
  - List profiling scenarios
- Options
  - Overall
  - Method Filters
  - Add method filter
  - Profiling levels
  - Add profiling level

## B.2 Guides

In this section, you can see basic guidelines to usage of implemented profiling tool.

- To create profiling scenario, follow these steps:
  1. On Start Profiling tab, click on New Project label.
  2. Fill Project Name attribute, this attributes is required.
  3. Fill description field to help you identify the purpose of creating project.
  4. Upload projects source file by clicking Upload button next to Inset JAR/WAR field. This is required field.
  5. Select attach option, if you want to attach to running JVM. If you want to start analysed application yourself, do not select this option. Start script will be generated.
  6. Pid field is required as well (if you selected attach option). Press select button located next to this field. You will be redirect to Attach to JVM page, where you can select target JVM.
  7. If needed, specify profiling level and profiling scenario. You can also create one of these objects by clicking Create button below them. This will redirect you to Create profiling scenario/ Add profiling level pages
  8. If everything is filled, press Submit button to start actual profiling and you will be redirected to profiling overview page.
- To create profiling scenario, select Create profiling scenario label in Scenarios tab or you can invoke this page by clicking Create button during profiling projects creation. Then follow these steps:
  1. Fill the name of a profiling scenario. This field is required
  2. Fill description to identify the purpose of created scenario.
  3. Select method from generated menu. If you want to hide methods extended from Object class, uncheck 'Include methods from class Object'.
  4. If you are done, press Submit button. You will be redirected to scenarios project page. Created profiling scenario is now selected as active.
- To create profiling level, follow these steps:
  1. On Options tab, select Add profiling level label.
  2. Fill level name. This field is required.
  3. Fill description to identify the purpose of created profiling level.
  4. Choose profiling aspects checking multiple generated checkboxes.

5. If you want to specify profiling intervals of selected aspects, fill their values into prepared fields. Values are in milliseconds. If values are not inserted, default values will be used.
  6. If you are done, press Submit button.
- To create method call filter, follow these steps:
    1. Select label Add method filter on options tab. You can invoke this page from create profiling scenario page by clicking add filter button next to each selected method.
    2. Fill filter name. This field is required.
    3. Fill method call description. This field is not required but it may help you to identify created filter in the future.
    4. Select parameter attribute from prepared drop down choice.
    5. Insert value of method parameter.
    6. If you are done, press Submit button.
  - To list existing profiling projects, select All profiling projects label in Projects tab. On this page, you can delete, view/edit or launch any of shown profiling projects.
  - To list existing profiling scenarios, select List profiling scenarios label in Scenarios tab. On this page, you can view/edit or delete any of shown profiling scenarios.
  - To list existing profiling levels, select Profiling levels label in Options tab. On this page, you can view/edit or delete any of shown profiling levels
  - To list existing method call filters, select Method filters label in Options tab. On this page, you can edit/view any of shown method call filters.
  - To edit basic profiling options, select Overall label in Options tab.

While profiling is running, you can perform these actions:

- You can view actual profiling project by selecting Actual profiling project label on Projects tab.
- You can view actual profiling scenario (is it is used) by selecting Actual profiling scenario label on Scenarios tab.
- Overview general profiling results by selecting Overview label on Actual profiling tab.
- See method call statistics by selecting Methods label on Actual profiling tab.
- See CPU, Memory, Threads, Classes profiling results by clicking on CPU, Memory, Threads and Classes labels on Actual profiling tab.
- See general information about target JVM by selecting VM Summary label on Actual profiling tab.



## Appendix C – Test Results

In this appendix, we will show manual test results as well as detailed results of profiling overhead testing and measurement.

### C.1 Manual Test Results

#	Test Scenario description	Expected Result	Result
<b>Tests with profiling projects</b>			
1	Create profiling project: 1. Select Start Profiling, then New Project, 2. Fill project name and choose source .war/.jar file and select JVM pid by choosing from attach to JVM dialog, 3. Click submit button	1. Profiling scenario should be created in repository 2. Profiling with this scenario should be launched.	✓
2	List profiling projects: 1. Click projects, then all profiling projects	List containing all existing profiling projects should be displayed.	✓
3	Delete profiling project: 1. Click projects, then all profiling scenarios. 2. Click 'X' (delete) icon in Actions row	Selected profiling project should be deleted and actualised list of profiling scenarios without this project should be displayed.	✓
4	View/Edit profiling project: 1. From All profiling projects page, click on View/Edit icon.	Selected profiling project should be displayed on new page with View/Edit functionality.	✓
5	Error project creation: 1. On New Project page, fill project name and insert invalid source file (not .jar/.war file)	Error message should be displayed.	✓
<b>Tests with profiling scenarios</b>			
6	Create profiling scenario: 1. On New Project page, click on Create button above Profiling Scenario drop down choice. 2. Fill scenario name and select arbitrary displayed methods. 3. Press submit button	1. User should be redirected to Create New Profiling Scenario Page 2. Profiling scenario should be created and user should be redirected back to New Project Page.	✓
7	List profiling scenarios: 1. Click Scenarios, then List profiling scenarios	List of existing scenarios should be displayed.	✓
8	Delete profiling scenario: 1. On List profiling scenarios page, click 'X' (delete) icon.	Selected scenario should be deleted and actualised list should be displayed.	✓

9	View/Edit profiling scenario: 1. On List profiling scenarios page, click View/Edit icon	User should be redirected to Actual profiling scenario page.	✓
<b>Profiling mechanisms tests</b> (profiling project with selected scenario and all profiling aspects in profiling level)			
10	Profiling Overview: 1. Click Actual profiling, then Overview.	Overall profiling information should be displayed.	✓
11	Method profiling: 1. Click Actual profiling, then methods tab	Information about selected method profiling should be displayed.	✓
12	Memory profiling: 1. Click Actual profiling, then memory tab.	Information about current memory usage should be displayed.	✓
13	CPU profiling: 1. Click Actual profiling, then CPU tab.	Information about current CPU usage should be displayed.	✓
14	Thread profiling: 1. Click Actual profiling, then Thread tab.	Information about current Thread activity should be displayed.	✓
15	Class profiling: 1. Click Actual profiling, then Class tab.	Information about current Class load statistics should be displayed.	✓
16	JVM overview: 1. Click Actual profiling, then VM summary tab.	Information about target applications HW and SW configuration should be displayed.	✓

...

Many more similar test scenarios were tested, but we believe, that reader already has clear image about manual testing of our profiling solution.

## C.2 Profiling Overhead Measurement

#	Method	Memory [ms]	CPU [ms]	Thread [ms]	Class [ms]	#1[s]	#2[s]	#3[s]	Overhead [s]	Overhead [%]
1	-	-	-	-	-	2,735	4,770	170,10	-	-
2	Y	1000	1000	1000	1000	2,812	4,982	173,40	3,30	1,941
3	Y	100	100	100	100	2,923	5,074	175,86	5,76	3,386
4	Y	10	10	10	10	3,072	5,103	179,61	9,51	5,591
5	Y	-	-	-	-	2,745	4,898	172,89	2,79	1,650
6	-	1000	-	-	-	2,736	4,841	171,57	1,47	0,864
7	-	100	-	-	-	2,737	4,866	172,09	1,99	1,169
8	-	10	-	-	-	2,739	4,916	173,12	3,02	1,775



9	-	-	1000	-	-	2,746	4,867	172,28	2,18	1,281
10	-	-	100	-	-	2,744	4,870	172,30	2,20	1,293
11	-	-	10	-	-	2,776	5,041	176,36	6,26	3,681
12	-	-	-	1000	-	2,746	4,907	173,09	2,99	1,757
13	-	-	-	100	-	2,742	4,932	173,49	3,39	1,992
14	-	-	-	10	-	2,757	5,079	176,75	6,65	3,909
15	-	-	-	-	1000	2,736	4,784	170,32	0,22	0,129
16	-	-	-	-	100	2,746	4,907	173,09	2,99	1,757
17	-	-	-	-	10	2,757	5,079	176,57	6,47	3,803

#### Explanation:

- In first row, we can see values for raw execution of tested application, in other words, without our profiling agent attached,
- values in Memory, CPU, Thread and Class columns means profiling interval,
- value in Method column means, if method run times were monitored,
- #1 – doSomething() method average execution time,
- #2 – memoryTest() method average execution time,
- #3 – tested application average execution time,
- overhead – average overhead, slowdown in comparison of raw execution represented in seconds and percents.

Results of profiling overhead measurement overcome even our optimistic assumptions. High profiling overhead is crucial problem when using many open-source profiling tools, but we have managed to implement solution with nearly negligible average profiling overhead.



## Appendix D – Glossary

---

API	Application Programming Interface
CLI	Command Line Interface
GUI	Graphical User Interface
HW	Hardware
JDK	Java Development Kit
JIT	Just in Time compilation
JMX	Java Management Extensions
JRE	Java Runtime Environment
JVM	Java Virtual Machine
JVM DI	Java Virtual Machine Debugging Interface
JVM PI	Java Virtual Machine Profiling Interface
JVM TI	Java Virtual Machine Tool Interface
SDK	Software Development Kit
SW	Software
UML	Unified Modelling Language
Bytecode	Java symbolic instruction language
C	programming language
C++	programming language
GB	Garbage Collector
HotSpot	JVM developed by Oracle
MySQL	Database
OpenDJ	Directory Service
MidPoint	A specific open-source identity management solution



## Appendix E - Figure List

---

Figure 1:	write once, run everywhere	4
Figure 2:	JVM work scheme	5
Figure 3:	Example of java source code analysed by HPROF tool	10
Figure 4:	Example used to demonstrate memory usage analysis using HPROF	11
Figure 5:	Example of modified source code used during memory...	12
Figure 6:	CPU usage profiling using YourKit profiler	15
Figure 7:	Memory profiling using YourKit java profiler	16
Figure 8, 23:	Use case diagram of implemented profiling solution	22, 57
Figure 9:	Basic profiling architecture scheme	26
Figure 10, 28:	Abstract profiling concept described using sequence diagram	29, 62
Figure 11, 29:	Profiling scenario concept	30, 63
Figure 12:	Wicket Model – View – Controller implementation	35
Figure 13:	Class loading transformation technique	38
Figure 14:	Profiling agent ↔ profiling server communication scheme	39
Figure 15:	Profiling agent thread activity scheme during application profiling	41
Figure 16:	Wicket GUI functionality explained	42
Figure 17:	GUI appearance of our profiling tool	43
Figure 18:	Chart representation of gathered profiling data	44
Figure 19:	User defined profiling scenario creation example	45
Figure 20:	Creation of profiling level using profilers graphical user interface	47
Figure 21:	Example of test scenarios	49
Figure 22:	Profiling overhead testing and measurement results	50
Figure 24:	Component diagram representing architecture of ...	58
Figure 25:	Package diagram containing main packages	59
Figure 26:	Detailed class diagram of package Types	60
Figure 27:	class diagram of profiling agent	61



## Appendix F – Source Code (CD medium)

---

Attached CD medium contains:

- xsuta\_bachelor\_thesis.pdf
- xsuta\_bachelor\_thesis.docx
- Annotations – directory containing all version of annotations
- project – directory containing source files
  - src – source files
  - pom.xml – project build configuration
- uml – uml diagrams and pictures
- pics – pictures from gui
- various – other relevant attachments

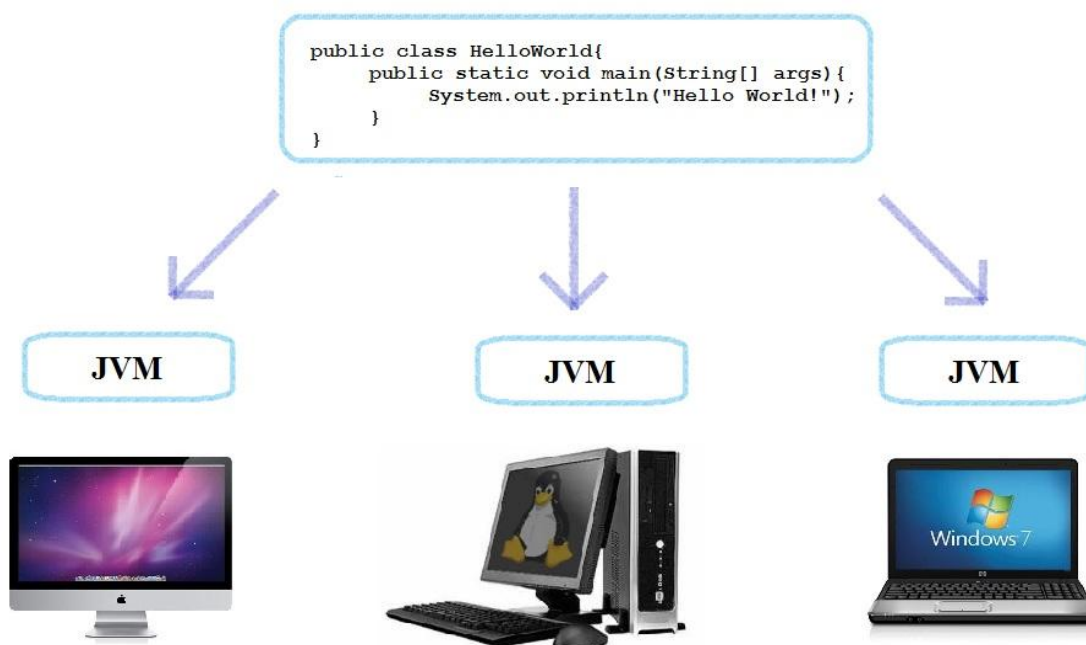




## Appendix G – Resume

Profilovanie je oblasť softvérového inžinierstva zaoberajúca sa dynamicou analýzou softvérových produktov počas ich vykonávania. Jeho cieľom je monitorovanie výkonnostných atribútov týchto aplikácií. Táto činnosť má za cieľ zvyšovanie výkonu a efektivity a znižovanie spotreby cenných systémových prostriedkov. V dnešnej dobe pracujú veľké softvérové aplikácie simultánne so státisícmi používateľov, pričom každý jeden očakáva okamžité reakcie na interakciu so softvérovým systémom. Čas sú peniaze a neefektivita používaných softvérových systémov môže v prípade veľkých softvérových spoločností znamenať výrazné straty.

Za účelom implementácie profilovacieho nástroja pre java aplikácie bolo najprv potrebné naštudovať problematiku a architektúru platformi java. Na rozdiel od klasických kompilovaných programovacích jazykov, kde je programátorom napísaný zdrojový kód priamo prekladaný do jazyka symbolických inštrukcií assembler a následne vykonávaný na hardvérovej vrstve počítača, v prípade platformi java vstupuje do hry ďalší medzikrok. Tým medzikrokom je Java Virtual Machine (ďalej JVM). V jednoduchosti povedané ide o klasický program, ktorý pracuje v operačnej pamäti počítača. Na rozdiel od bežných programov však JVM simuluje prácu počítača pomocou implementovanej funkcionality zásobníkov, registrov a dokonca vlastnej inštrukčnej sady. Ide teda o virtuálny počítač. Zdrojové kódy napísané v jazyku java sú prekladané do jazyka bytecode, ktorý je následne vykonávaný na bežiacej JVM. Týmto prístupom je možné dosiahnuť princíp multiplatformovosti.



Obr. 1: „write once, run everywhere“

Následná hĺbková analýza problematiky profilovania java aplikácií ukázala, že existuje viacero možných prístupov, konkrétne možno rozdeliť profilovacie nástroje do niekoľkých kategórií:

- Profilovacie nástroje založené na udalostiach,
- štatistické profilovacie nástroje,
- profilovacie nástroje využívajúce inštrumentáciu zdrojového kódu.

V prípade profilovacích nástrojoch založených na udalostiach dochádza k skúmaniu vykonávania programu práve pomocou implementovaných udalostí bežiacej JVM. Medzi tieto udalosti patrí napríklad vstup a výstup z metódy, načítanie a uvoľnenie triedy či spustenie cyklu garbage collector. Štatistické profilovacie nástroje používajú úplne odlišný prístup. Chod analyzovaného programu je pravidelne prerušovaný špeciálnymi systémovými inštrukciami, pričom je následne spustený cyklus zberu výkonnostných parametrov analyzovanej aplikácie. Tretí druh nástrojov využíva dynamickú inštrumentáciu zdrojových súborov. Táto technika ešte pred samotným spustením vykonávania programu vloží profilovací kód na určené miesta v programe. Vložený kód počas vykonávania programu vykonáva zber informácií o výkonnosti programu.

Prebehla detailná analýza existujúcich profilovacích riešení. Konkrétne sme analyzovali podporu profilovania zabudovanú priamo do štandardného java API, čo sme demonštrovali na príklade práce s nástrojom HPROF. Následne sme analyzovali riešenia s otvoreným zdrojovým kódom, konkrétne ide o nástroje HAT (Heap Analysis Tool), JProbe, JConsole či VisualVM. Analyzovali sme aj platformu Eclipse TPTP či Netbeans profiler. Z komerčných profilovacích riešení sme preverili program YourKit java profiler.

Následne prebehla špecifikácia a analýza požiadaviek. Zistili sme, že žiaden existujúci profilovací nástroj presne nevyhovuje potrebám kladeným zadaním tejto práce, pretože neposkytujú dostatočné možnosti špecifikácie profilovania. Boli určené nasledovné funkčné požiadavky:

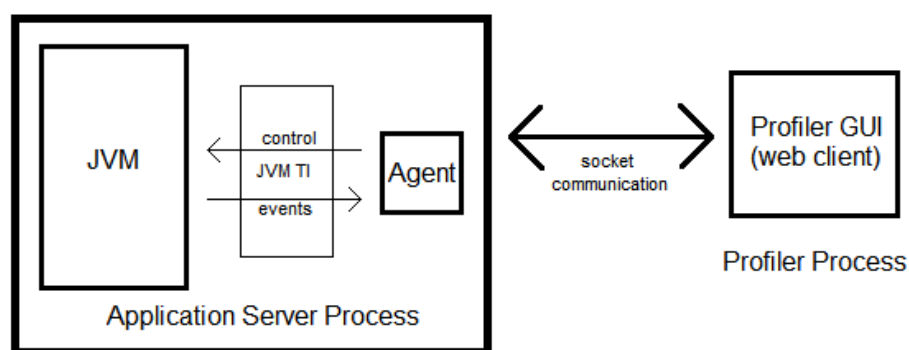
- Aktívny zber informácií výkonnostných parametrov použitím vhodných techník,
- vykonávanie zberu informácií o činnosti procesora, operačnej pamäte, volania metód a aktivity vlákien počas prevádzky softvérového systému,

Taktiež sme identifikovali niektoré nefunkcionálne požiadavky:

- Implementácia profilovacieho nástroje vo forme webaplikácie,
- intuitívne grafické používateľské rozhranie,
- čo najmenší negatívny dopad profilovania na výkonnosť analyzovanej aplikácie,
- jednoduchá rozšíriteľnosť,
- integrácia s väčším systémom s otvoreným zdrojovým kódom.

Architektúra typu klient-server nie je často vídaná v kontexte profilovania softvérových systémov, avšak lepšie vyhovuje stanoveným požiadavkám.

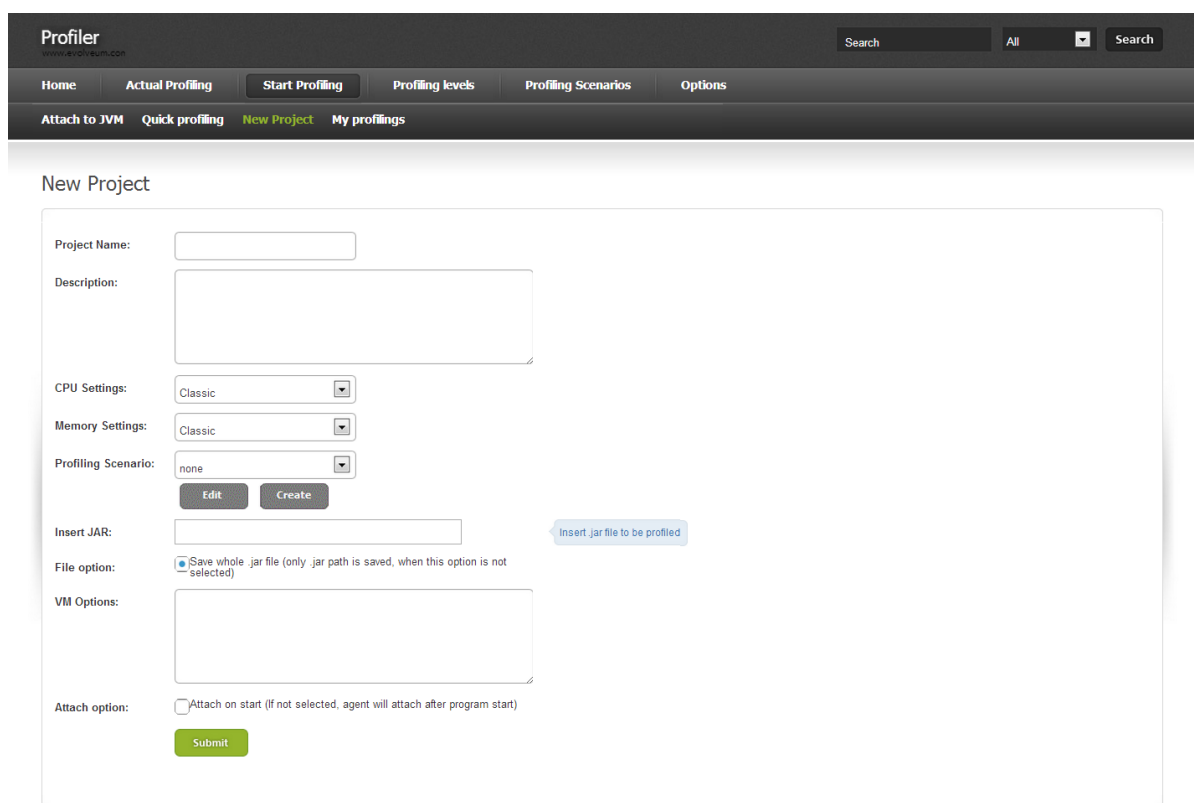
Následne bol vykonaný návrh aplikácie. V prvom rade sme navrhli mechanizmus profilovania. Samotný zber výkonnostných údajov je vykonávaný samostatnou jednotkou, tzv. agentom, ktorý je pred začatím vykonávania skúmanej aplikácie vložený do cieľovej JVM. Agent pomocou vstavaného rozhrania vykonáva dopyty ohľadom stavu a výkonnosti JVM, tieto informácie následne posielajú profilovaciemu serveru prostredníctvom komunikačného rozhrania. Na strane profilovacieho servera sú dáta spracované a zobrazené používateľovi v zrozumiteľnej forme. Tento proces možno detailnejšie vidieť na nasledujúcom obrázku.



**Obr. 10:** Základná schema profilovania. JVM TI je skratka z Java Virtual Machine Tool Interface.

Mechanizmus profilovania sme doplnili návrhom troch nových techník. Prvou z nich je technika definovaných používateľských scenárov. V grafickom používateľskom rozhraní si používateľ môže zvoliť metódy, ktorých výkonnosť ho zaujíma. Tieto informácie sú následne poslané agentovi, ktorý ignoruje volania ostatných ako používateľom zvolených metód. Ďalšou technikou sú tzv. úrovne profilovania. Nie vždy vývojára java aplikácií zaujímajú všetky aspekty výkonnosti. Niekedy chceme skúmať len prácu s operačnou pamäťou, popriprade záťaž procesora. Používateľ teda má možnosť zvoliť aspekty, ktorých skúmanie je relevantné. Túto techniku možno rozšíriť o tzv. profilovacie intervaly. Ide o časové úseky medzi jednotlivými cyklami zberu výkonnostných dát. Niekedy, hlavne pri rozsiahlych softvérových systémoch, ktorých architektúra je navrhnutá za účelom jednoduchej rozšíriteľnosti, sa stáva, že je široká paleta operácií vykonávaná jedinou metódou, pričom smerovanie funkcionality tejto metódy je určené vstupnými parametrami. Vývojára vykonávajúceho profilovanie môžu zaujímať len špecifické behy skúmanej metódy, preto sme navrhli techniku filtrov volania metód, ktorá umožňuje používateľom špecifikovať, ktoré behy vykonávania metód ich zaujímajú práve na základe hodnôt a typov vstupných parametrov.

Implementácia profilovacieho nástroja prebehla na základe stanovených požiadaviek a vypracovanom návrhu riešenia. Použili sme techniku dynamickej inštrumentácie zdrojových súborov použitím technológie Javassist a štandardných vstavaných knižníc v java API (`java.lang.instrument`). Zber dát týkajúcich sa činnosti operačnej pamäte, záťaže procesora či aktivity vlákien bol implementovaný použitím techniky vzorkovania a technológie JMX (Java Management eXtensions). Aplikácia bola implementovaná použitím klient-server architektúry s využitím tenkého klienta. Grafické používateľské rozhranie sme implementovali pomocou webovej šablóny s otvoreným zdrojovým kódom. Táto šablóna bola prepojená s profilovacou funkcionalitou použitím webového frameworku Apache Wicket, ktorý je založený na princípoch návrhového vzoru MVC (Model-View-Controller). Na implementáciu grafického zobrazovania zozbieraných výkonnostných parametrov sme použili knižnicu Wicket-Charts. Vzhľad aplikácie možno vidieť na nasledujúcom obrázku.



The screenshot shows the 'Profiler' application interface. At the top, there is a navigation bar with links: Home, Actual Profiling, Start Profiling, Profiling levels, Profiling Scenarios, and Options. Below this is a secondary bar with links: Attach to JVM, Quick profiling, New Project (highlighted), and My profilings. The main content area is titled 'New Project' and contains a form with the following fields and controls:

- Project Name:** A text input field.
- Description:** A larger text area.
- CPU Settings:** A dropdown menu with 'Classic' selected.
- Memory Settings:** A dropdown menu with 'Classic' selected.
- Profiling Scenario:** A dropdown menu with 'none' selected.
- Edit** and **Create** buttons.
- Insert JAR:** A text input field with a blue button labeled 'Insert jar file to be profiled'.
- File option:** A checkbox labeled 'Save whole jar file (only jar path is saved, when this option is not selected)' which is checked.
- VM Options:** A text area.
- Attach option:** A checkbox labeled 'Attach on start (if not selected, agent will attach after program start)' which is unchecked.
- Submit** button.

**Obr. 17:** Vzhľad grafického používateľského rozhrania implementovanej aplikácie.

Boli taktiež implementované techniky spomenuté v návrhu riešenia, konkrétne sme implementovali používateľom definované profilovacie scenáre, úrovne profilovanie ale aj filtre volania metód.

Implementované riešenie sme následne overili sériou testov. Vytvorili sme množstvo manuálnych testovacích scenárov, ktorých výsledky možno vidieť v prílohe C. Pre testovacie účely sme taktiež vytvorili aj sériu automatických testov pomocou frameworku TestNG, ktorý

ponúka funkcionality podobnú ako framework JUnit. Dôraz sme kládli na testovanie a meranie všeobecnej záťaže vykonávania profilovania na skúmanú aplikáciu. Časť výsledkov možno vidieť v nasledujúcej tabuľke. Vysvetlenie spoločne s detailnejšími výsledkami testovania možno vidieť v prílohe C.

#	Method	Memory [ms]	CPU [ms]	Thread [ms]	Class [ms]	#1[s]	#2[s]	#3[s]	Overhead [s]	Overhead [%]
1	-	-	-	-	-	2,735	4,770	170,10	-	-
2	Y	1000	1000	1000	1000	2,737	4,832	171,40	1,3	1,007
3	Y	100	100	100	100	2,745	4,847	171,86	1,76	1,010
4	Y	10	10	10	10	2,750	4,979	174,61	4,51	1,026

**Obr. 22:** Výsledky merania negatívneho výkonnostného dopadu profilovacieho nástroja na analyzovanú aplikáciu.

Implementované riešenie sme taktiež testovali v spolupráci s midPointom. MidPoint je softvérový produkt pôsobiaci v oblasti manažmentu identít a taktiež ide o produkt, do ktorého sa bude implementované riešenie integrovať.

V tejto bakalárskej práci sme sa zamerali na oblasť profilovania java aplikácií. Postupne sme vykonali všetky fázy štandardného cyklu vývoja softvéru. Konkrétne šlo o analýzu problémovej oblasti, špecifikáciu požiadaviek a stanovenie cieľov, návrh aplikácie a jej mechanizmov a samotná implementácia nasledovaná vyčerpávajúcim testovaním. Do oblasti profilovania sme uviedli niekoľko nových, respektíve upravených mechanizmov, konkrétne ide o používateľmi definované profilovacie scenáre, úrovne profilovania a filtre volaní metód. Práca na tejto aplikácii nie je ani zďaleka dokončená, keďže existuje množstvo vylepšení, ktorým sa budeme ďalej venovať, konkrétne:

- Väčšia dynamickosť používateľského rozhrania a integrácia technológie AJAX,
- XML forma pre objekty repozitára,
- rozšírenie filtrov volaní metód o možnosť definovania filtrov pomocou výrazového jazyka a mnohé iné.

Veríme však, že projektom sme splnili cieľ bakalárskej práce. Najbližší osud implementovaného riešenia je integrácia so softvérovým systémom midPoint, konkrétne bude vykonaná v niektorej z nasledujúcich stabilných verzií (2.3).