Institut für Softwaretechnologie

Abteilung Zuverlässige Softwaresysteme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Fachstudie Nr. 184

Evaluation von Java Profiler Werkzeugen

Albert Flaig, Daniel Hertl, Florian Krüger

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Lars Grunske
Betreuer:	Dipl. Inf. André van Hoorn
begonnen am:	31.05.2013
beendet am:	30.11.2013
CR-Klassifikation:	D.4.8

Abstract

The purpose of this study is to evaluate Java profilers and compare them with each other. As profilers differ in various aspects the evaluation has to cover many functional and non-functional scopes like the user interface and license properties, states of development, range of support, and the given underlying conditions. Choosing the right profiler is not an easy task as there is a wide variety each with their own pros and cons. This study aids in decision making by providing a comparison and enabling the user to easily weight up each individual aspect to ones personal needs. Through this study we try to publish a detailed comparison of current profilers as well as a personal recommendation based on objective, well-defined criteria. In this context different kinds of profilers — commercial as well as open source — will be looked at in an attempt to find the best-fitting tool for specific usage scenarios. In order to achieve accurate results each profiler is tested with a similar set of examinees. As a result a table based ranking will be established by the help of the perceived results. The result of this analysis can further be used to choose one profiler out of the tested ones to perfectly fit to the user needs. Secondarily it also provides a detailed overview of current profilers and its functional scopes.

Contents

Chapter 1

Introduction

1.1 Motivation

Software has seamlessly integrated into our everyday lives in the recent decades. Through the ever increasing processing power of today's computers, software has to solve tasks more complex than ever before. Therefore the demands of the software have risen and thus increased the responsibility of the programmers. Even in the rarest exception, software has to function correctly or else will cause high costs. In some extreme situations software errors can even endanger human lives. In order to identify flaws or inefficiency in programs, specific techniques have been developed that help the developer seek problematic code spots. One technique is described through the use of profilers. Profilers are tools which analyze run time behavior on various aspects like the efficiency of specific routines, the memory usage or the concurrency. They are used especially in development environments to help developers pin down memory leaks, demanding methods, and critical code segments. Recently new aspects have appeared like measuring the amounts of database accesses or analyzing internet interfaces. The goal of this study is to use and compare commercial and non-commercial profilers and highlight their similarities and differences.

Collaboration with NovaTec GmbH

This study is executed in cooperation with NovaTec Consulting GmbH[1] located in Leinfelden-Echterdingen.

1.2 Goals

The goal of this study is to compare current Java profilers and acquire meaningful results to allow to carry out result analysis. Open source profiler will be considered as well as commercial products. Through research the aspects of profiler and the possibilities of the Java VM should be summarized. All profilers under test will be inspected with a similar set of examinees to achieve results in order to create relative comparisons. A detailed report will be created by the help of the study results to gain deeper insight into the functional behavior of each profiler. Besides the main functionality like observing

[1] http://www.novatecgmbh.de/

CPU processing, memory usage or thread activity special features like coverage analysis, workflow integration into other tools and continuous integration or support of further JVM based languages are to be assessed individually. In addition to the functional aspects, the usability of the profiler itself will also get a rating. To create a user-friendly overview, a catalog is to be created by the help of the detailed reports. The catalog should show a short summary of each profiler that includes a ranking of its functionality that offers the possibility to filter a small set of profilers to fit perfectly to the users needs.

Tasks in Detail

▻ Illustration of the JVM profiler concept through the use of (JVM TI)

▻ Categorizing of profiler tools by license, features, risks, and effort

▻ Setup of a unified feature catalog of the tools including a term glossary

▻ Crucial illustration of the quality of the tools based on personal experiments and extern reviews especially regarding developer workflow

▻ Creation of a decision-making support based on use cases to help the selection of profiler tools.

Chapter 2

Foundations

The term profiling describes a dynamic analysis that covers issues of the resource management of a program during execution. Main issues like CPU utilization, memory usage or thread activity could be considered as well as special functions like network activity or garbage collection. However this study considers Memory Profiling, Thread Profiling and CPU Profiling as main aspects and are described in Section 2.1. Most Java profiler tools rest upon the Java Virtual Machine Tooling Interface which has been introduced in Java 5 JVM [2004]. Therefore they rather differ in presentation and workflow than the amount of available profiling information. See Section 2.2 for a overview of the JVM TI. A small set of available profilers use different methods to collect information than through the usage of JVM TI. To gather execution data profilers use the technique of Bytecode Instrumentation, which is described in Section 2.4. Also some profilers use Method Sampling to reduce overhead. This technique is described in Section 2.3. In Section 2.5 remote Java Management Extensions (JMX) connections are described, which are used to connect to a remote JVM, which is mostly useful for monitoring but also used during profiling.

2.1 Profiling Aspects

The purpose of this section is to define the borders of profiling aspects. For example when talking about CPU profiling one needs to know what aspects are included in this aspect. This section is going to clarify each term.

2.1.1 Memory Profiling

In this study the term Memory Profiling describes the ability to inspect heap, memory management, allocated objects, allocation stack trace

2.1.2 Thread Profiling

In this study the term Thread Profiling involves the ability to see the states of each thread, the code place a specific thread is running, the stack trace of the thread and the time spent on each method by the thread, and analyze possible dead lock scenarios. Thread Profiling transitions smoothly into CPU Profiling.

2.1.3 CPU Profiling

In this study the term CPU Profiling describes the ability to inspect timers, CPU usage in general and the percentage used by specific packages, classes, and methods.

2.2 Java Virtual Machine Tooling Interface

The Java Virtual Machine Tooling Interface is a programming interface for use by tools. It provides functions to inspect the state of the Application under Test (AuT) running in a JVM. The functions of JVM TI include profiling, debugging, monitoring, thread analysis, coverage analysis, and more. Most profilers rely on the JVM TI to gather information on the AuT. The abilities of the JVM TI are too vast to be covered here JVM. To retrieve data through the JVM TI one needs to provide a JVM TI-agent. Agents have to be written in a native programming language. Each profiler that relies on the JVM TI has got its own agent which tells the JVM TI what kind of data has to be gathered on the AuT.

2.3 Method Sampling

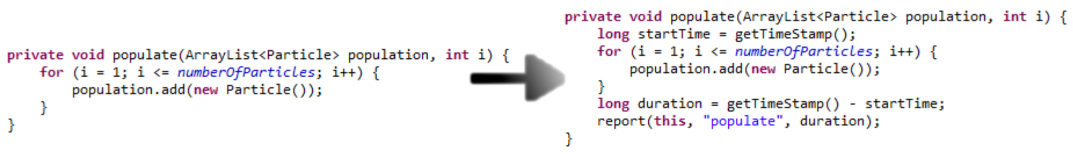
Method Sampling can be used to reduce overhead, however this technique greatly reduces the level of detail of gathered data. When using Method Sampling one profiles the application periodically after a set amount of time has passed JPM [2011]. On some implementations of this technique the missed information on execution data is extrapolated. Execution time of methods is thus estimated. Different profilers use a different term when referring to this technique. In Chapter 7 only the term Method Sampling is used, regardless what term the profiler uses.

2.4 Bytecode Instrumentation

Bytecode Instrumentation (BCI) describes the technique of injecting custom code in Java classes during runtime. The purpose is to inject code which calls a custom event trigger on different occasions like at class load time, method start and end time, custom expression evaluation and many more. See figure 2.1 for a simplified example of Bytecode Instrumentation (BCI).

There are different methods of using BCI JVM:

- **Static Instrumentation**
  The class file is instrumented before it is loaded into the JVM. Usually by providing a duplicate class file with injected code.
- **Load-Time Instrumentation**
  The class file is injected with custom code during load time in memory only.



```
private void populate(ArrayList<Particle> population, int i) {
    for (i = 1; i <= numberOfParticles; i++) {
        population.add(new Particle());
    }
}
```

```
private void populate(ArrayList<Particle> population, int i) {
    long startTime = getTimeStamp();
    for (i = 1; i <= numberOfParticles; i++) {
        population.add(new Particle());
    }
    long duration = getTimeStamp() - startTime;
    report(this, "populate", duration);
}
```

Figure 2.1. The left method without BCI; the right method with BCI

▻ **Dynamic Instrumentation**
Class files which are already loaded and possibly even during execution are injected with custom code in memory only.

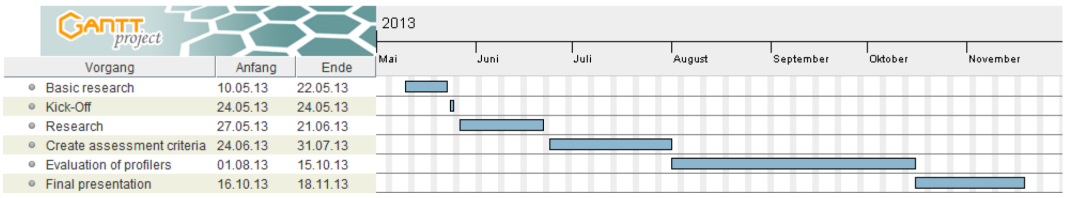
2.5 Remote JMX Connection

The Java Remote JMX Connection is an interface to connect to remote Java Applications running in a different JVM on a remote or local host. This provides profiling and monitoring capabilities on remote Java applications. The JMX connection supports SSL authentication. This is a very important feature, especially for monitoring purposes, but is also useful for profiling. One can monitor this way either a local or a remote JVM. In contrast to the Java Virtual Machine Tooling Interface, the Java Management Extensions do not provide in-depth information of the AuT and thus there is a need to provide additional interfaces within the AuT for in-depth information mining. The JMX provides basic data like the number of allocated objects, the CPU usage, general information of the system and the JVM, etc.

Chapter 3

Research Procedure

This study is conducted within six months and takes place in the period from August 2, 2013 to November 18, 2013. A meeting with the adviser was held periodically every second week to discuss the recent results. In a team of three members, we performed this study. We introduced milestones to split our goals into small steps and improve the cooperation with NovaTec. (3.1) visualizes our schedule as a Gantt-diagram.



Vorgang	Anfang	Ende
Basic research	10.05.13	22.05.13
Kick-Off	24.05.13	24.05.13
Research	27.05.13	21.06.13
Create assessment criteria	24.06.13	31.07.13
Evaluation of profilers	01.08.13	15.10.13
Final presentation	16.10.13	18.11.13

Figure 3.1. schedule of study

3.1 Preparation

To prepare us for a presentation and a well-founded discussion we had to make us familiar with the basics of the issue profiler. We took a short look at current profilers and read some articles of the theoretical basics. The outcome has been included as Foundations in Chapter 2.

3.2 Kick-Off

After the preparation phase in order to make ourselves familiar with the matter and to prepare to an analysis discussion, our study started with the kick-off meeting in the office of NovaTec GmbH in Leinfelden-Echterdingen. After a short introduction of the company, two representatives depicted their concern regarding the main focus of the study and we discussed the demands of an expressive study.

The early evaluation criteria of profiler tools have already been established in the kick-off meeting.

3.3 Research

After the kick-off meeting, a 4-week-lasting research phase followed. During this phase, existing profilers have been searched and categorized. Several interesting profilers have been found through this research. In order to establish a detailed as possible list of profilers, we started to gather information about the profilers by the use of forums and data of the manufacturer.

3.4 Create Assessment Criteria

To compare the profilers against each other we had to create a few assessment criteria that characterize a profiler at all important aspects. We laid down important points like CPU utilization, memory usage, or thread activity. Beside that we wanted to take a further look at the individual profilers and added a detailed list of criteria to our catalog. The result of the gathered criteria can be looked up at the created table that offers an overview of the current profilers.

3.5 Evaluation of Profilers

To establish well-founded results, we installed each listed profiler and created a small set of test cases to take a closer look at the functionality. To challenge the specific functionality of all profilers we chose two applications that were especially suited for these test cases. Furthermore, we developed code snippets and injected them into the test applications. By the help of these snippets we were able to compare the overhead which was produced by the profilers during the profile process. During the evaluation of a single profiler a detailed report of its functional behavior was written.

3.6 Final Presentation

After all profilers were tested and the study report was finished, a final presentation was held at NovaTec GmbH.

Chapter 4

Market Overview

The following list represents our selection of current profilers. Refer also to Chapter A to see a short summary of each individual tool and the complete list of our initial profiler lineup. As time was limited we were not able to consider every tool. Therefore in Section 4.1 each tool is shown which has been removed from this study with a short reasonable explanation. The following list in Section 4.1 represents the profilers being tested in this study.

4.1 Considered Profilers

▻ JVM Monitor (Section A.12)
JVM Monitor is a Java profiler to monitor CPU, threads, and memory usage of Java applications.

▻ Java VisualVM (Section A.18)
VisualVM is a tool which utilizes various available technologies like jvmstat, JMX, the Serviceability Agent, and the Attach API to profile applications.

▻ NetBeans Profiler (Section A.16)
This profiler comes along with functions including CPU, memory and thread profiling as well as basic JVM monitoring. It is integrated into the Netbeans IDE.

▻ Eclipse TPTP (Section A.5)
Eclipse Test & Performance Tools Platform (TPTP) includes frameworks to analyze the run time behavior of desktop and enterprise applications written in Java. However it is no longer developed and support for the newer JVM versions has been dropped.

▻ InspectIt (Section A.6)
Analyzing tool developed by NovaTec GmbH. Collects data through manually integrated measuring points at Java classes.

▻ JProfiler (Section A.11)
A commercial profiler that comes along with a wide range of functionality. It runs on various operating systems and development environments.

▻ YourKit Java Profiler (Section A.19)
A commercial profiler that comes along with a wide range of functionality. It is free to use for open source projects and runs on various operating systems.

▻ dynaTrace (Section A.2)
Commercial profiler with a huge range of features.

▻ JRockit Mission Control (Section A.7)
Profiler that includes tools to monitor, manage, profile, and eliminate memory leaks in Java applications with little performance overhead.

▻ Memory Analyzer (Section A.15)
The Eclipse Memory Analyzer (MAT) helps to get Heap dumps.

Excluded Profilers

▻ Hyades Eclipse Plugin (Section A.3)
The Hyades Eclipse Plugin has been integrated into Eclipse TPTP. Instead Eclipse TPTP has been tested as mentioned in Section 4.1.

▻ Thermostat (Section A.17)
Thermostat has only monitoring capability for now. Profiling functions are planned but not yet implemented.

▻ Eclipse Profiler Plugin (Section A.4)
The developer itself recommends the use of other profiler tools and does not support Eclipse Profiler Plugin anymore.

▻ jMechanic (Section A.9)
The last official version was v0.6 and has been released in 2004. Due to this inactivity jMechanic is not considered.

▻ JRat (Section A.14)
The development of JRat is already inactive since several years. Thus it has been excluded.

▻ AppDynamics (Section A.1)
AppDynamics is a monitoring tool that is specialized for production environments rather than for profiling.

▻ JMap (Section A.8)
Unsupported small tool, which isn't stable and only runs on a few Linux distributions.

▻ JProbe (Section A.10)
A powerful commercial tool. However, we were not able to get a hold of a test license for the latest version.

▻ jvmstat (Section A.13)
A small tool without GUI whose only functionality lies in the profiling of the memory heap. It has too little functionality to be considered for an extensive test.

Chapter 5

Evaluation Criteria

To compare individual profiler tools with each other and to get a detailed list of the functionality the profiler have been evaluated by specific criteria and categorized in scales. Thereby a criteria catalog has been created to rate each profiler in each category. However at this point the ratings are already filled in with the use of the scale definitions in section 5.2. See Chapter 7 on page 25 for the actual evaluation.

5.1 Criteria Catalog

5.2 Scale Definitions

In this caption you find the scales you need to evaluate the tested tools. Every scale is well defined so it is possible to evaluate all tools.
The scale types used are ordinal and nominal scales. The ordinal scale types are described in a table like grades in school. In the particular columns you will find the corresponding grade definition. The nominal scales only describes if a profiler has a particular property.

5.2.1 Nominal Scales

- **Development:**
  - Latest version
    Date : The date of the latest version of the profiler
  - Future-proof
    Is the profiler future-proofed? : Yes/No
- **License:**
  - Name
    Under which license is the profiler? : Open Source / Eclipse Public License v1.0 / commercial [test version available]
- **Support:**

  - ▻ Forum
    Is a forum available? : Yes/No Forum available
  - ▻ Documentation
    Is a documentation available? : Yes/No documentation available
  - ▻ Active Support
    Is a active support available? : Yes/No active support available
  - ▻ Active Community
    Is a active community available? : Yes/No active community available

- ▻ **Basic conditions:**
  - ▻ IDE-integration
    Is it possible to integrate the profiler and if possible for what IDE? : [Yes],Eclipse / [Yes],Netbeans / No
  - ▻ OS-compatibility
    For what OS is the profiler provided for? : Windows / MacOS / Linux / Solaris / AIX / FreeBSD / HP-UX
  - ▻ extensibility
    Is the profiler extendable? : Yes/No is (not) extendable

5.2.2 Ordinal Scales

Memory Scale*

Memory Type	Scale value				
	0	1	2	3	4
Database	With this profiler it is not possible to observe any database objects	The profiler can connect to a Database and show the data sets of an application	The profiler is able to do save trends of data set changes during the profile time	The profiler is able to understand where the instructions of statements come from and show it	The profiler make execution of statements visible. You can follow the exchange of data. You can save the results of database connections and you can compare it to earlier results
Heap	The profiler isn't able to display the Heap	The profiler is able to take snapshots of the Heap	The profiler is able to show the heap and the live changes during execution	The profiler is able to show detailed Heap and the corresponding classes, packages, objects and arrays.	The Profiler is able to analyze the heap with diagrams and create and save heap dumps to compare them to earlier records.

*Each scale value includes the aspects of the prior ones.

Thread Scale*

Thread profiling	Scale value				
	0	1	2	3	4
Functionality	The profiler don't show any running thread	The profiler shows all running threads in a timeline or table	The profiler shows all threads and can visualize them in different graphical method. You can research the threads at self started recorded time.	The profiler shows all threads of the running system. You can record the threads at a particular time and go more detailed into that record. You can create a thread dump	You have the possibility to visualize, filter and go more detailed into the data of the running threads. The feature to follow call hierarchies and to go into the source code is possible

*Each scale value includes the aspects of the prior ones.

CPU Scale*

CPU profiling	Scale value				
	0	1	2	3	4
Functionality	The profiler don't show CPU action	The profiler show snapshots of CPU activity	The profiler shows a time line which displays the CPU percentage	The profiler shows a visualization of running threads and their percentage CPU load	The profiler shows detailed computing percentage of each packages or more exactly of each class.

*Each scale value includes the aspects of the prior ones.

Overhead Scale*

Overhead	Scale value				
	0	1	2	3	4
Overhead influence and settings	The overhead have to much influence of executing the application under test. You're not able to work with the application while the profiler is running	The overhead is perceptible but doesn't disable the execution of the application.	There are possibilities to adjust the overhead of single actions you want to do	There are settings which adjust the overhead. You can choose several settings to set up the expected overhead.	The overhead doesn't affect the execution of the application.

*Each scale value includes the aspects of the prior ones.

Additional functions*

Additional functions	Scale value				
	0	1	2	3	4
Count and quality of additional functions	There are no additional functions	Additional functions only support the main function	The additional functions are very helpful in case of profiling	The profiler comes along with a huge amount of professional functions beside the main functionality	The profiler is extendable for example with plug ins

*Each scale value includes the aspects of the prior ones.

GUI scale*

GUI Quality Properties	Scale value				
	0	1	2	3	4
Usability, Utility, Likability, Workflow, 8 golden rules of Ben Shneiderman	The profiler has no GUI	The profiler has a visualization part	The profiler has some settings and options to adjust the profilers functions	You are able to personalize the profiler	User Interface design is observed the 8 golden rules of Ben Shneidermans

*Each scale value includes the aspects of the prior ones.

Chapter 6

Evaluation Procedure

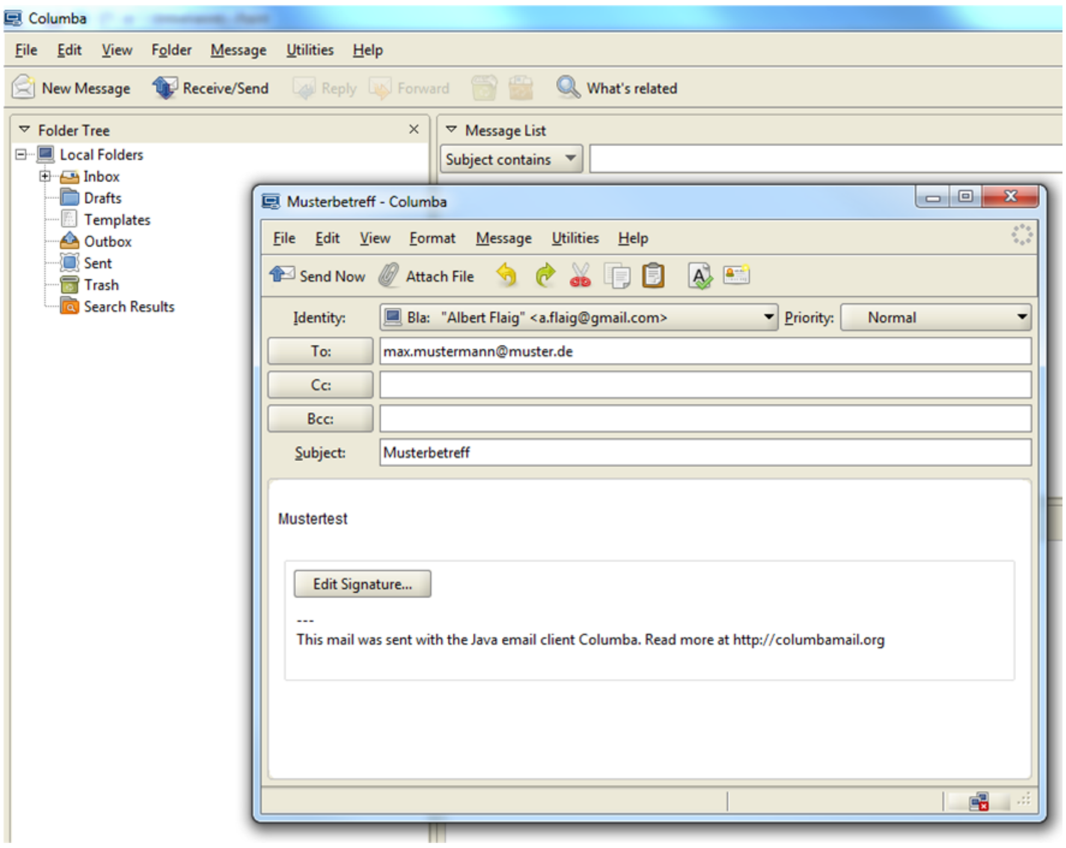
To execute the evaluation on profilers, a test object has to be chosen which exhausts as many as possible features of the profiler under test, based on the evaluation criteria on page 15. The test object has to be applicable to as many profiler as possible in order to make a uniform statement. Therefore several applications are used to cover specific tasks of profilers. These two applications are introduced in Section 6.1. Additionally, use cases have to be defined in order to establish an objective and uniform evaluation procedure. In Section 6.2 use cases are defined which are to be executed for each test of each profiler for both introduced applications.

6.1 Applications under Test

Columba

Columba[1] is an open source email client [Col]. This application has been chosen to monitor its network activity and examine the features of the profiler in this field. Also this application is relative big. Beside a high concurrency it demands a lot of functions of the profiler. Thus the profiler being tested can be measured qualitatively.

[1]http://sourceforge.net/projects/columba/



Identity: Bla: "Albert Flaig" <a.flaig@gmail.com>
Priority: Normal
To: max.mustermann@muster.de
Cc:
Bcc:
Subject: Musterbetreff

Mustertest

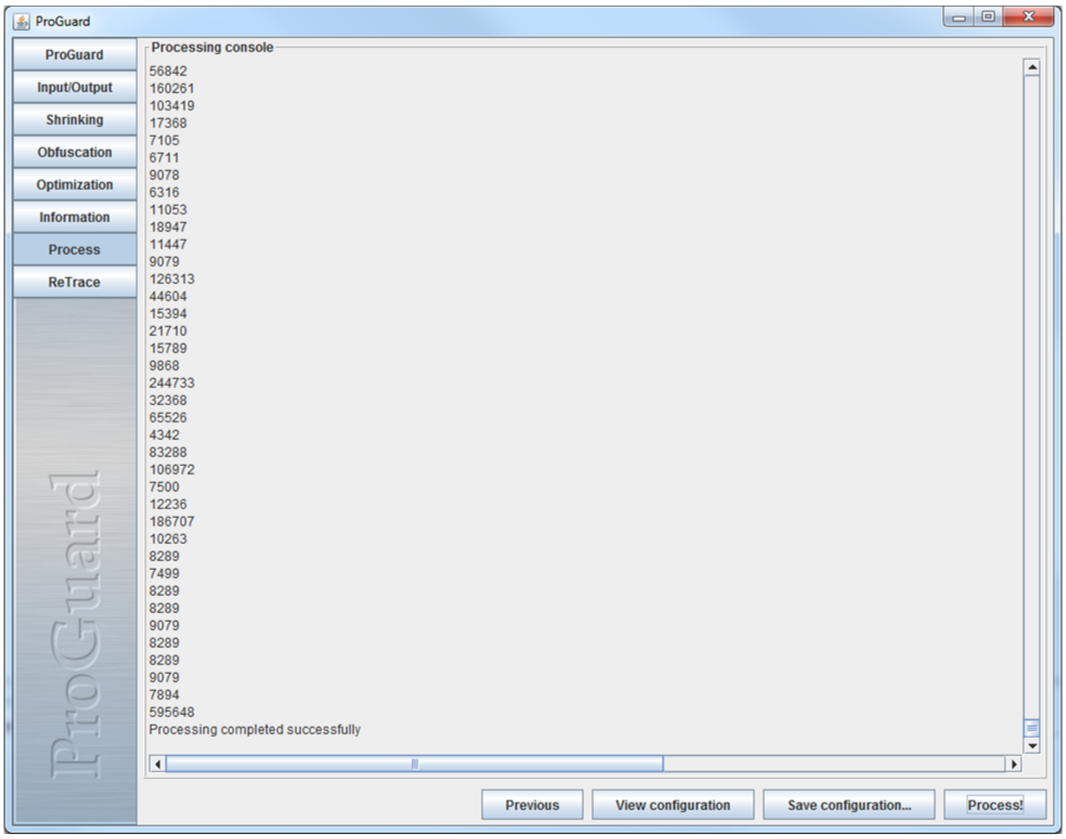
This mail was sent with the Java email client Columba. Read more at http://columbamail.org

Figure 6.1. Columba, a versatile email client

Proguard

Through the use of Proguard[2] one can optimize and obfuscate Java code in order to remove unused code and make the decompilation process of Java applications more difficult [Pro]. This process is memory and CPU heavy and demands the corresponding monitoring functions of the profiler being tested. Also during obfuscating one can easily measure the overhead, as this process claims a lot of processing time. Therefore the Proguard class shrinker component is injected with an execution time measuring code.

[2]http://proguard.sourceforge.net/



```
56842
160261
103419
17368
7105
6711
9078
6316
11053
18947
11447
9079
126313
44604
15394
21710
15789
9868
244733
32368
65526
4342
83288
106972
7500
12236
186707
10263
8289
7499
8289
8289
9079
8289
8289
9079
7894
595648
Processing completed successfully
```

Figure 6.2. ProGuard, a simple java obfuscator; The output during obfuscating procedure is shown at this point

6.2 Use Cases

We evaluate the profilers with predefined use cases to reach comparable results. The use case for each application under test is now described.

Columba

Columba is an email management program, therefore it is apparent to research the sending and receiving of emails.
The test is started with a request to the added email provider, to check whether there are new messages. The next step is to write a new mail with at least five receivers and content of a short inserted text.

Proguard

In Proguard we start our testing scenario with optimizing some code. We choose the Columba executable jar as a uniform test object. The configuration is left in default state except for *Ignore warnings about possibly erroneous input* which is getting checked.

Chapter 7

Experimental Evaluation

7.1 JVM Monitor

JVM Monitor makes a solid impression on first sight. The plugin flawlessly integrates into Eclipse. The Java Monitor perspective is easy to figure out. The JVM Explorer view shows active java applications that can be profiled with a simple mouse click. There is no need to mess around with run configurations which leaves a positive mark. The usual features are included: Thread, memory and CPU profiling. Some important functions like starting and stopping the CPU profiling are represented as small buttons on the top bar of the view and are thus easy to miss. But still JVM Monitor is pretty easy to find out how to use it. The time line shows various data like memory usage, loaded classes and more over the course of run time.

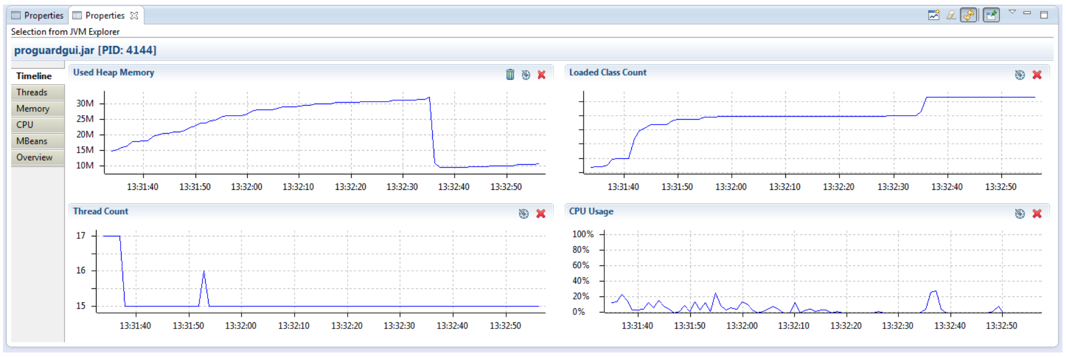


Figure 7.1. Screenshot of the JVMMonitor tool

However the time line takes getting used to as the scales are not perfectly clear and not always shown. Hovering your mouse over one graph tells its exact value though. Also the more advanced user can add more graphs to the time line by inputting specific java beans which are mostly already provided by the JVM. These beans can also be watched in the MBeans tab.

CPU Profiling

BCI while CPU profiling puts quite a heavy toll on the overhead. In experiment the measured method took around 5 to 10 times more time to execute. However you can easily turn this feature off/on and specify the refresh precision which relieves overhead to a negligible amount.

Thread Profiling

Each thread is shown with its claimed CPU power and its state along with its stack trace. A very pleasant feature of JVM Monitor is to jump to the source code location of a stacktrace entry while profiling threads. This can be done without any additional configuration.

Memory Profiling

Nothing extraordinary here. The needed information is displayed nicely in the Eclipse view and the memory can be dumped with a simple click.

7.2 VisualVm

VisualVM impressed with a nice clear GUI. It can be used as Eclipse plugin. But this only start the stand alone program. At first view the Profiler locks really good visualized and gives a lot of information about the application.

The GUI

The GUI has a menu with six sections. Below the menu there is a little toolbar with some options. The selection of the toolbar icons can be change by the user to personalize your own tool. Further more the profiler got a tree view of the applications that are possible to observe. The Remote observable applications are also shown in the tree. The last item in the tree is the recorded snapshots that had been taken by the user. In the main window in the middle, you can see the proper profiler.

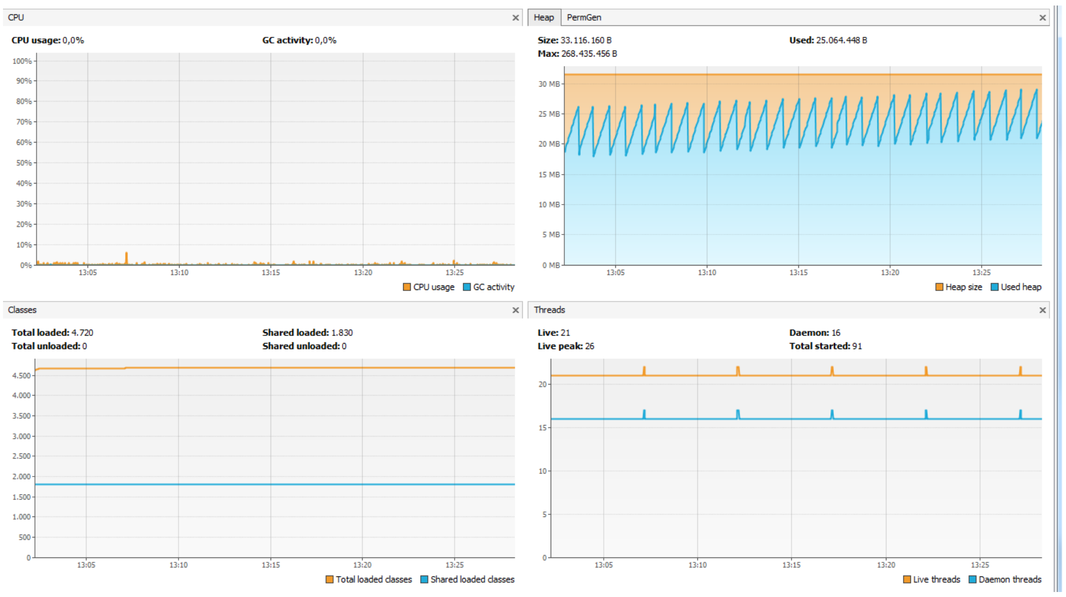


Figure 7.2. Overview of the VisualVm timelines

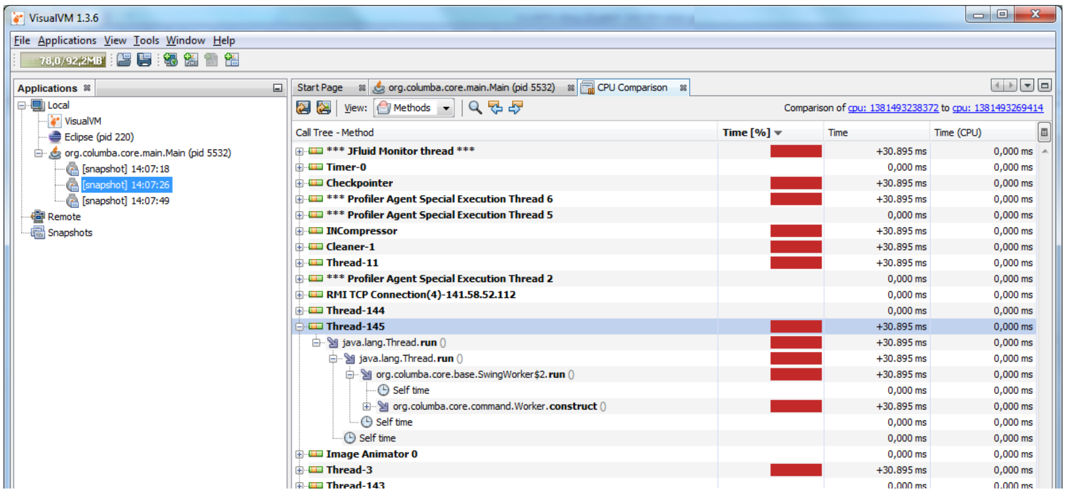
Main functions

Application manager

In the tree view of the profiler you can choose one of the observable applications which are temporary running on your system. You can select them with an double click to go into the detailed view or you can choose the short way with a right click on the application you want to research. Direct executable functions in the Application manager are creating a "Heap Dump", "Thread Dump", taking an application snapshot or adding a remote connection to a host where an application is running you want to research.

CPU observation

VisualVM provides several possibilities to observe the CPU activities. There is an live diagram which shows the CPU process while the application is running. In this diagram you can point a particular time you want to see the CPU load exactly. Further more CPU activities in a Sampler and in a profiler. In the Sampler you got details about the CPU since the start of the application under test. Now you have the possibility to take copy of the actual status of the CPU. Later you can compare this copy to earlier copies. You will see the difference between the snapshots as you can see in Figure 2.



Comparison of cpu: 1381493238372 to cpu: 1381493269414

Call Tree - Method	Time [%]	Time	Time (CPU)
*** JFluid Monitor thread ***	🟥	+30.895 ms	0,000 ms
Timer-0		0,000 ms	0,000 ms
Checkpointer	🟥	+30.895 ms	0,000 ms
*** Profiler Agent Special Execution Thread 6	🟥	+30.895 ms	0,000 ms
*** Profiler Agent Special Execution Thread 5		0,000 ms	0,000 ms
INCompressor	🟥	+30.895 ms	0,000 ms
Cleaner-1	🟥	+30.895 ms	0,000 ms
Thread-11	🟥	+30.895 ms	0,000 ms
*** Profiler Agent Special Execution Thread 2		0,000 ms	0,000 ms
RMI TCP Connection(4)-141.58.52.112		0,000 ms	0,000 ms
Thread-144		0,000 ms	0,000 ms
Thread-145	🟥	+30.895 ms	0,000 ms
java.lang.Thread.run ()	🟥	+30.895 ms	0,000 ms
java.lang.Thread.run ()	🟥	+30.895 ms	0,000 ms
org.columba.core.base.SwingWorker$2.run ()	🟥	+30.895 ms	0,000 ms
Self time		0,000 ms	0,000 ms
org.columba.core.command.Worker.construct ()	🟥	+30.895 ms	0,000 ms
Self time		0,000 ms	0,000 ms
Self time		0,000 ms	0,000 ms
Image Animator 0		0,000 ms	0,000 ms
Thread-3	🟥	+30.895 ms	0,000 ms
Thread-143		0,000 ms	0,000 ms

Figure 7.3. Comparison of the snapshots

And the last option to check the CPU is the most interest. If you go to the profiler you got nearly the same function as in the Sampler with the big difference that you can start recording of the CPU at a particular time. Than you can do the same actions like in the Sampler. You can got CPU snapshots and compare them to earlier versions.

Heap observation

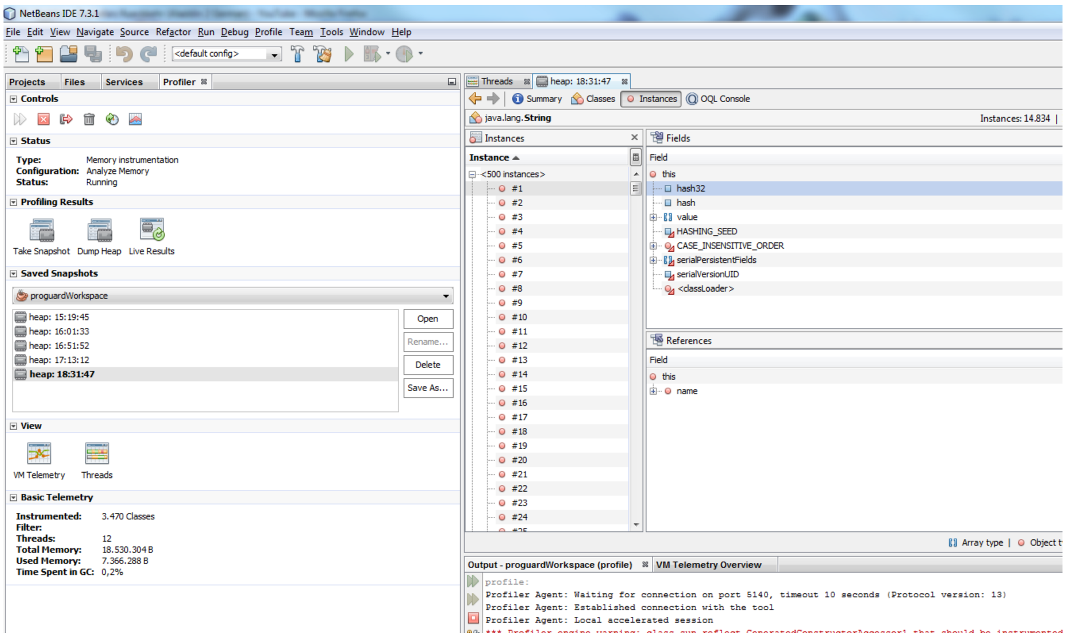
The Heap observation is the same case like the observation of the CPU. You got several options of researching the memory. A diagram, the Sampler and the profiler. The function to observe the heap is analogue to the CPU profiling function. You see the instances and the live objects at a particular time. Again with the difference that in the Sampler you got it since the start and in the profiler started at a particular time.

7.3 NetBeans Profiler

The Netbeans Profiler is a feature of Netbeans. On the first view it is almost conspicuously that it is similarity to the VisualVM profiler. This may be because VisualVm is actually equal profiler than the Netbeans Profiler. But there are some differences between the two profilers. The plugin is more powerful than the stand alone VisualVM. The main advantage is that you can directly go into the code and check the points you are profiling.

The GUI

The individual screens are nearly the same like in the VisualVM profiler. The navigator on the left side is the only difference, but a really nice and helpful add-on. If you double click one of the classes in the memory view you will go to a view that shows detailed the corresponding instances of this class and the actual states.



Type: Memory instrumentation
Configuration: Analyze Memory
Status: Running

Instrumented:	3.470 Classes
Filter:	
Threads:	12
Total Memory:	18.530.304 B
Used Memory:	7.366.288 B
Time Spent in GC:	0,2%

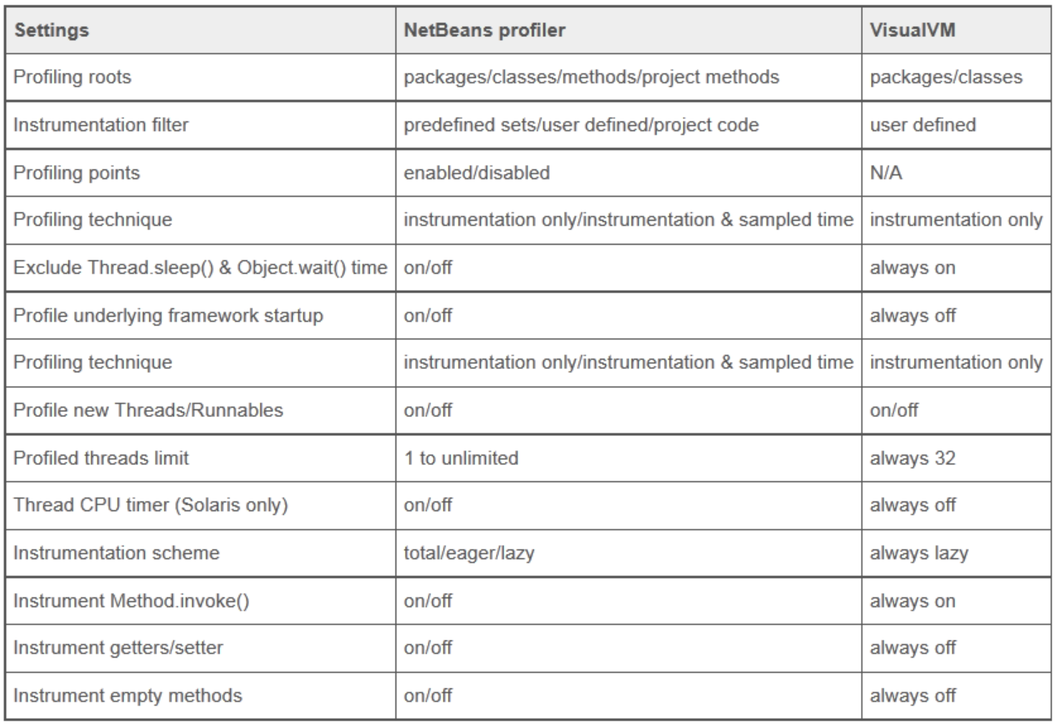
```
profile:
Profiler Agent: Waiting for connection on port 5140, timeout 10 seconds (Protocol version: 13)
Profiler Agent: Established connection with the tool
Profiler Agent: Local accelerated session
*** Profiler engine warning: class sun.reflect.GeneratedConstructorAccessor1 that should be instrumented
```

Figure 7.4. NetBeans Profiler Overview

Main funcions

The main functions of the Netbeans profiler are basically the same as they are in VisualVM. But the advantage of the plugin is that you are directly connected to your code you want to profile.

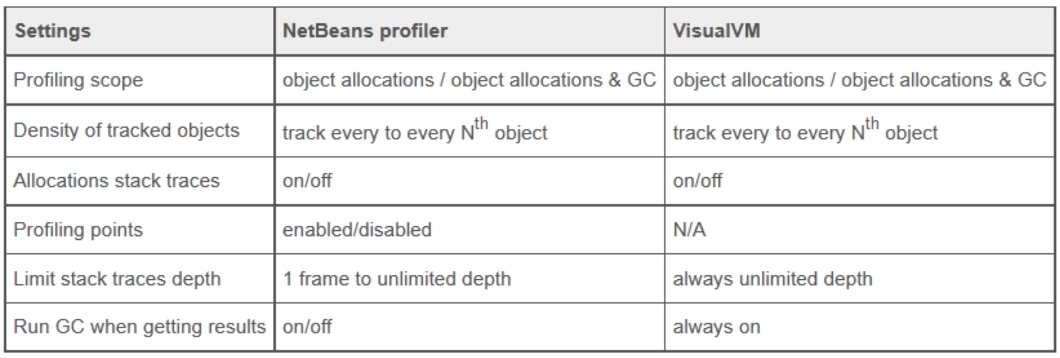
The differences to VisualVM in case of CPU profiling you will see in the following table.



Settings	NetBeans profiler	VisualVM
Profiling roots	packages/classes/methods/project methods	packages/classes
Instrumentation filter	predefined sets/user defined/project code	user defined
Profiling points	enabled/disabled	N/A
Profiling technique	instrumentation only/instrumentation & sampled time	instrumentation only
Exclude Thread.sleep() & Object.wait() time	on/off	always on
Profile underlying framework startup	on/off	always off
Profiling technique	instrumentation only/instrumentation & sampled time	instrumentation only
Profile new Threads/Runnables	on/off	on/off
Profiled threads limit	1 to unlimited	always 32
Thread CPU timer (Solaris only)	on/off	always off
Instrumentation scheme	total/eager/lazy	always lazy
Instrument Method.invoke()	on/off	always on
Instrument getters/setter	on/off	always off
Instrument empty methods	on/off	always off

Figure 7.5. Comparison of the NetbeansProfiler and VisualVm in case of CPU profilingJiri [July 28 ,2008]

The differences to VisualVM in case of Memory profiling you will see in the following table.



Settings	NetBeans profiler	VisualVM
Profiling scope	object allocations / object allocations & GC	object allocations / object allocations & GC
Density of tracked objects	track every to every N^{th} object	track every to every N^{th} object
Allocations stack traces	on/off	on/off
Profiling points	enabled/disabled	N/A
Limit stack traces depth	1 frame to unlimited depth	always unlimited depth
Run GC when getting results	on/off	always on

Figure 7.6. Comparison of the NetbeansProfiler and VisualVm in case of CPU profilingJiri [July 28 ,2008]

7.4 Eclipse Test & Performance Tools Platform

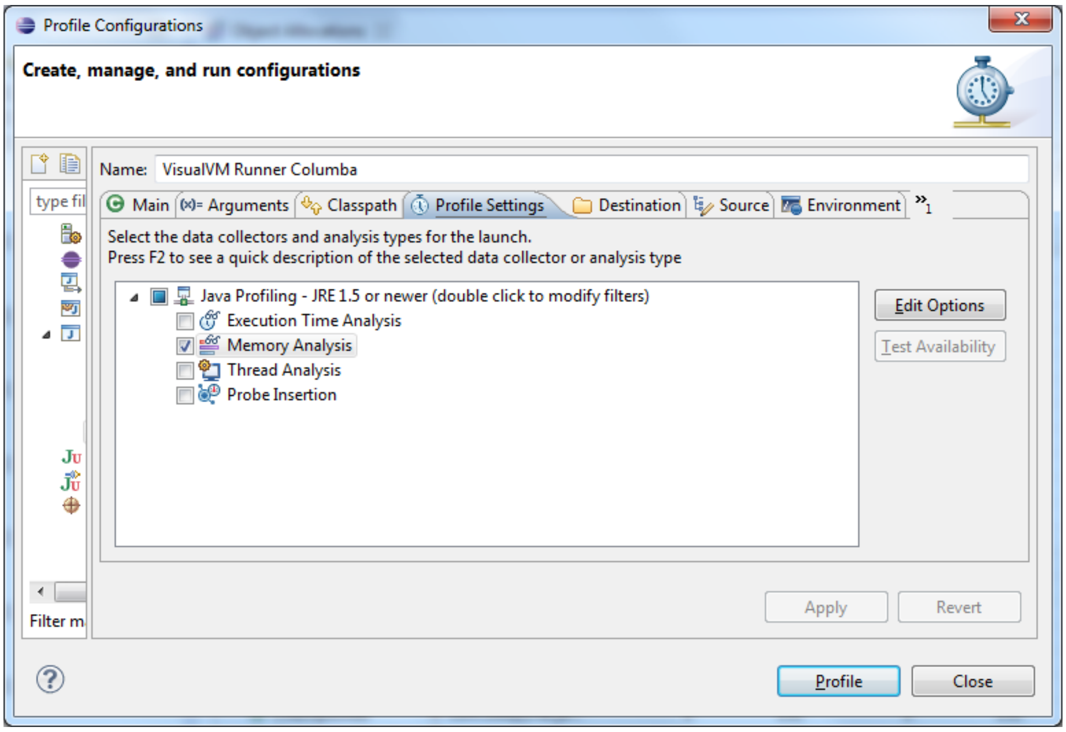
The Eclipse Test & Performance Tools Platform is a powerful yet hard to use profiler. There is no standalone version and therefore it is very conservative in its user interface. Unfortunately due to its development stop in February 2011 the profiler is left in an unstable state. It is incompatible with Java 1.7 as it causes errors through its byte code instrumentation making the application unable to execute. The plugin seems to slowdown the workbench when at work and the usability seems unpolished. Certain buttons have unexpected behavior and you can't move the uml diagrammes around. Also you can only profile one category at a time which is a blow to its workflow rating.

Additional to its profiling capabilities Eclipse TPTP provides Probekit, a powerful feature which allows developers to inject Java code fragments at specific points in code on certain events. This allows for lightweight profiling by printing out only the information needed by developers.

CPU Profiling

The execution statistics are neatly presented with just the right amount of information. Each package is shown along its classes which in turn have its methods.

However there is a huge overhead. In experiment Proguard was at times frozen during computation. The experiment had to be aborted. So the conclusion is to use CPU profiling only on specific packages, classes or methods. There is also an execution flow feature which shows a graphical representation of method calls along a vertical axis representing execution time. The graph is flooded with method calls and cannot be moved around, only zoomed in and out.



Name: VisualVM Runner Columba

Select the data collectors and analysis types for the launch.
Press F2 to see a quick description of the selected data collector or analysis type

- Java Profiling - JRE 1.5 or newer (double click to modify filters)
  - [ ] Execution Time Analysis
  - [✓] Memory Analysis
  - [ ] Thread Analysis
  - [ ] Probe Insertion

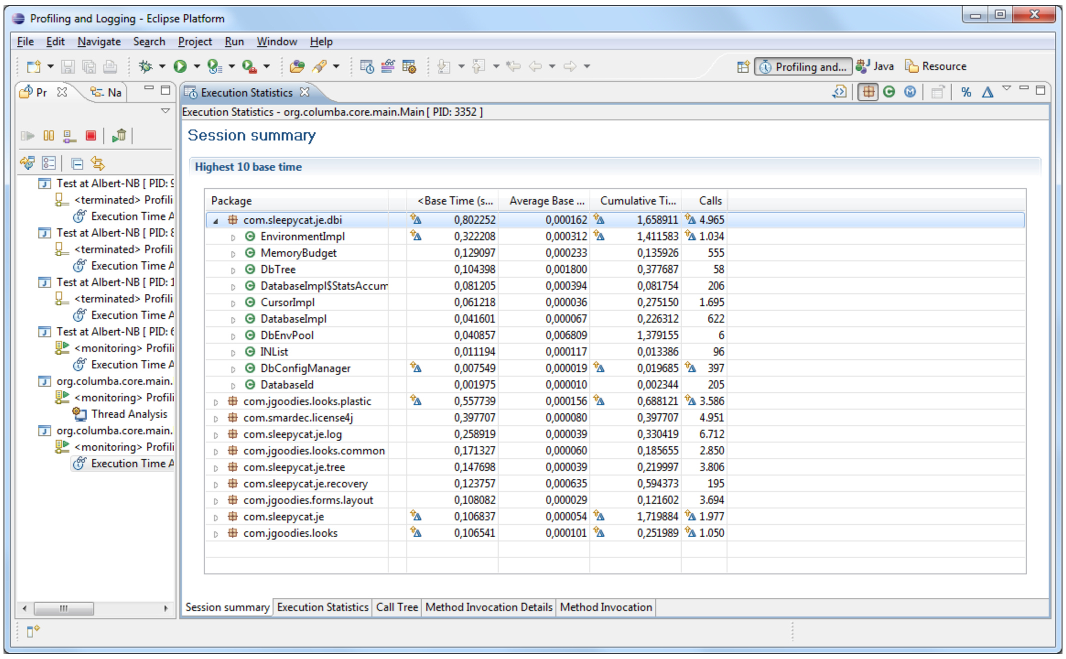
Figure 7.7. The TPTP profile configuration

Thread Profiling

Eclipse TPTP has good thread profiling capabilities. The usual thread overview is included and presented in a structured way. However the jump to source code location is missing at this point when trying to jump to a specific threads implementation. The statistics show what class uses which thread and further in which method. Also there is a thread visualizer that gives the developer an overview of thread states at given instants.

Memory Profiling

Memory profiling has the usual information included which is presented in a good structured way. Allocation details can be shown to see at which point a specific class has been allocated.


Profiling and Logging - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Profiling and... Java Resource
Execution Statistics

Execution Statistics - org.columba.core.main.Main [PID: 3352]

Session summary

Highest 10 base time

Package	<Base Time (s...	Average Base ...	Cumulative Ti...	Calls
com.sleepycat.je.dbi	0,802252	0,000162	1,658911	4.965
EnvironmentImpl	0,322208	0,000312	1,411583	1.034
MemoryBudget	0,129097	0,000233	0,135926	555
DbTree	0,104398	0,001800	0,377687	58
DatabaseImpl$StatsAccum	0,081205	0,000394	0,081754	206
CursorImpl	0,061218	0,000036	0,275150	1.695
DatabaseImpl	0,041601	0,000067	0,226312	622
DbEnvPool	0,040857	0,006809	1,379155	6
INList	0,011194	0,000117	0,013386	96
DbConfigManager	0,007549	0,000019	0,019685	397
DatabaseId	0,001975	0,000010	0,002344	205
com.jgoodies.looks.plastic	0,557739	0,000156	0,688121	3.586
com.smardec.license4j	0,397707	0,000080	0,397707	4.951
com.sleepycat.je.log	0,258919	0,000039	0,330419	6.712
com.jgoodies.looks.common	0,171327	0,000060	0,185655	2.850
com.sleepycat.je.tree	0,147698	0,000039	0,219997	3.806
com.sleepycat.je.recovery	0,123757	0,000635	0,594373	195
com.jgoodies.forms.layout	0,108082	0,000029	0,121602	3.694
com.sleepycat.je	0,106837	0,000054	1,719884	1.977
com.jgoodies.looks	0,106541	0,000101	0,251989	1.050

Session summary | Execution Statistics | Call Tree | Method Invocation Details | Method Invocation

Figure 7.8. CPU Analyze View

Rating

▻ Need to tamper with run configurations to start profiling (Workflow -1)

▻ Allows only to profile one category at a time (Workflow -2)

▻ Slows down Eclipse in general (Usability -2)

▻ Not compatible with newest Java version (Future-Proof -5)

▻ Sometimes the tab within a view won't switch correctly (Usability -1)

▻ Good presentation of execution statistics (CPU +1)

▻ Huge overhead (Overhead -4)

▻ Thread visualization (Threads +1)

▻ Memory allocation trace (Memory +1)

▻ ProbeKit for custom Java code injection (Features +1)

▻ Can be extended by various plugins (Features +1)

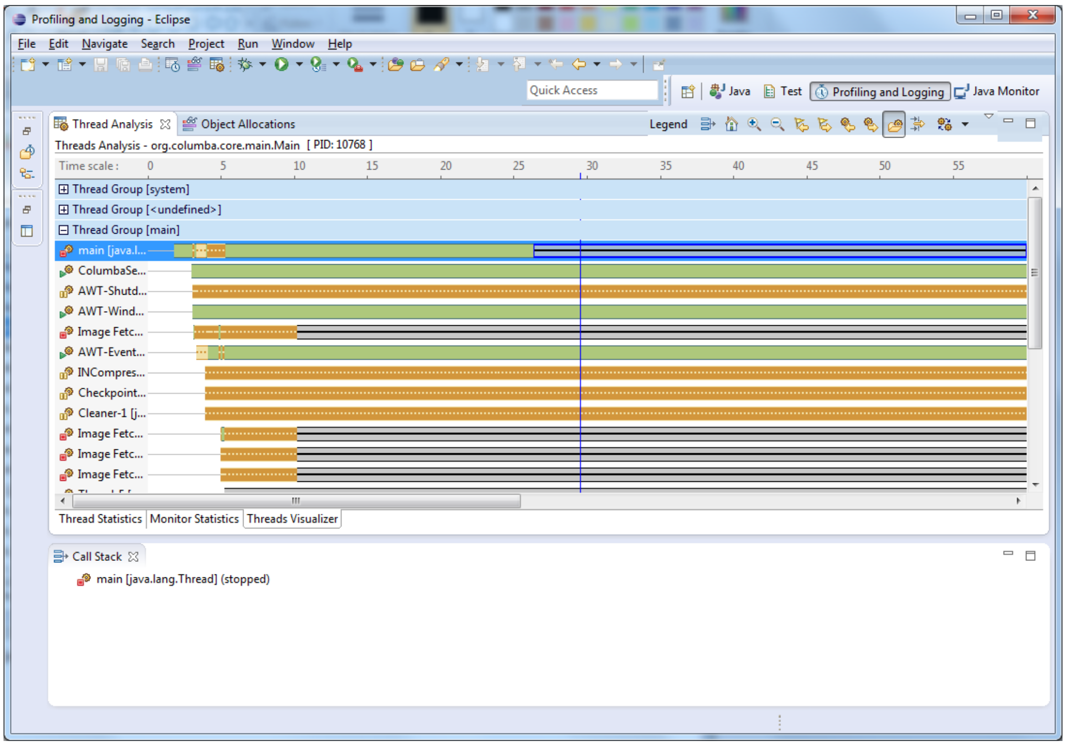
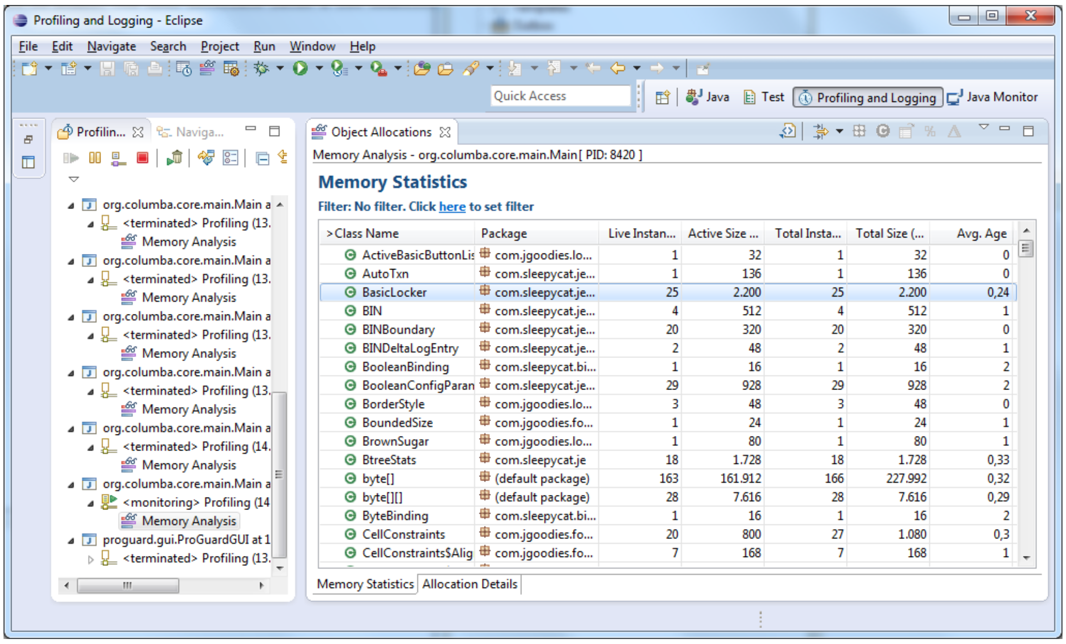


Figure 7.9. Detailed Thread Analysis

Summary

One point of criticism is the need to tamper with run configurations to start profiling. Also one can only profile one category at a time. This has quite an impact on workflow. It noticeably slows down the Eclipse IDE and tabs will not switch reliably. It is clear that Eclipse TPTP is left in an unstable state. Eclipse TPTP is not future-proof at all, as development has been halted and there is no support for newer Java versions. Also there was a huge overhead tested in experiment. Nevertheless it has a few powerful features like the ProbeKit and it can be extended by various plugins. The Thread and CPU visualization is done nicely and during memory testing one can take a look at the allocation trace. Eclipse TPTP has had its place in the profiler race but not it is definitely left behind.



org.columba.core.main.Main a
<terminated> Profiling (13.
Memory Analysis
org.columba.core.main.Main a
<terminated> Profiling (13.
Memory Analysis
org.columba.core.main.Main a
<terminated> Profiling (13.
Memory Analysis
org.columba.core.main.Main a
<terminated> Profiling (13.
Memory Analysis
org.columba.core.main.Main a
<terminated> Profiling (14.
Memory Analysis
org.columba.core.main.Main a
<monitoring> Profiling (14
Memory Analysis
proguard.gui.ProGuardGUI at 1
<terminated> Profiling (13.

Memory Analysis - org.columba.core.main.Main[PID: 8420]

Memory Statistics

Filter: No filter. Click here to set filter

>Class Name	Package	Live Instan...	Active Size ...	Total Insta...	Total Size (...	Avg. Age
ActiveBasicButtonLis	com.jgoodies.lo...	1	32	1	32	0
AutoTxn	com.sleepycat.je...	1	136	1	136	0
BasicLocker	com.sleepycat.je...	25	2.200	25	2.200	0,24
BIN	com.sleepycat.je...	4	512	4	512	1
BINBoundary	com.sleepycat.je...	20	320	20	320	0
BINDeltaLogEntry	com.sleepycat.je...	2	48	2	48	1
BooleanBinding	com.sleepycat.bi...	1	16	1	16	2
BooleanConfigParan	com.sleepycat.je...	29	928	29	928	2
BorderStyle	com.jgoodies.lo...	3	48	3	48	0
BoundedSize	com.jgoodies.fo...	1	24	1	24	1
BrownSugar	com.jgoodies.lo...	1	80	1	80	1
BtreeStats	com.sleepycat.je	18	1.728	18	1.728	0,33
byte[]	(default package)	163	161.912	166	227.992	0,32
byte[][]	(default package)	28	7.616	28	7.616	0,29
ByteBinding	com.sleepycat.bi...	1	16	1	16	2
CellConstraints	com.jgoodies.fo...	20	800	27	1.080	0,3
CellConstraints$Alig	com.jgoodies.fo...	7	168	7	168	1

Figure 7.10. Detailed Memory Analysis

7.5 inspectIT

InspectIT is a tool which was developed by NovaTec and belongs to the group of monitoring tools more then to the group of profiling tools. We want to take a closer look at this application anyway. With inspectIT its possible to include several sensors within the sourcecode of an application. This sensors collects data through a connection to the instrumented hooks of methods. Which methods gets equipped with such a hook will be predetermined by a configuration file. The following figure shall describe each component and its functionality.

The application is based on three components.

▻ User Interface
This component visualizes all created data by the sensors. Besides the profiling functionality like CPU activity, loaded classes, needed memory or thread activity it's possible to analyze sql statements, http activity and exceptions.

▻ Centralized Measurement Repository
The Centralized Measurement Repository(CMR) saves all information from each sensor given by the agents. The user interface has access to this information and visualize.

▻ inspectIT Agent

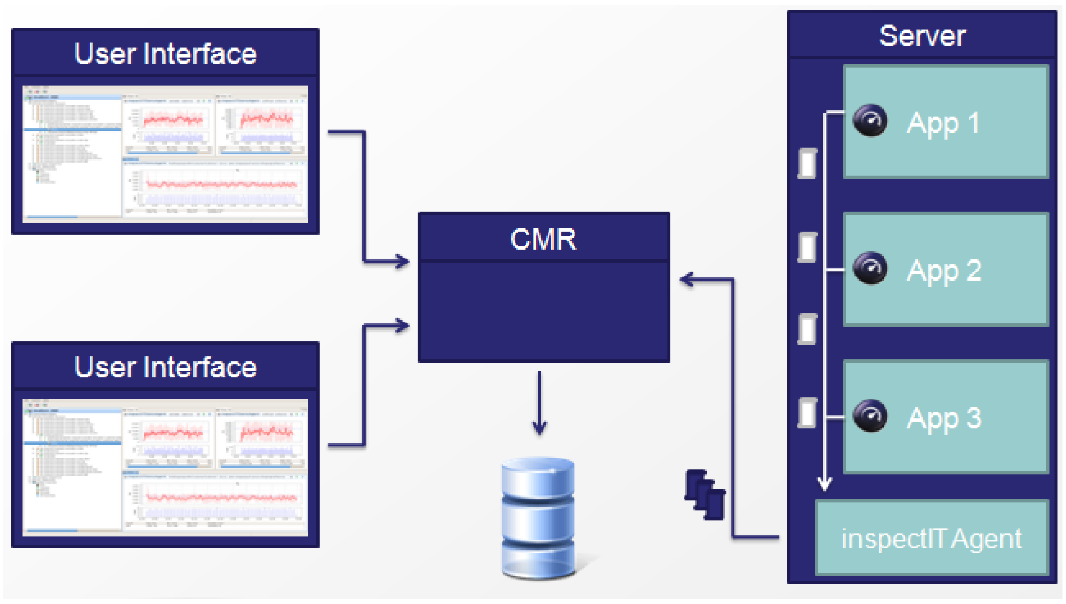


Figure 7.11. Environment structure Siegl [May 17, 2013]

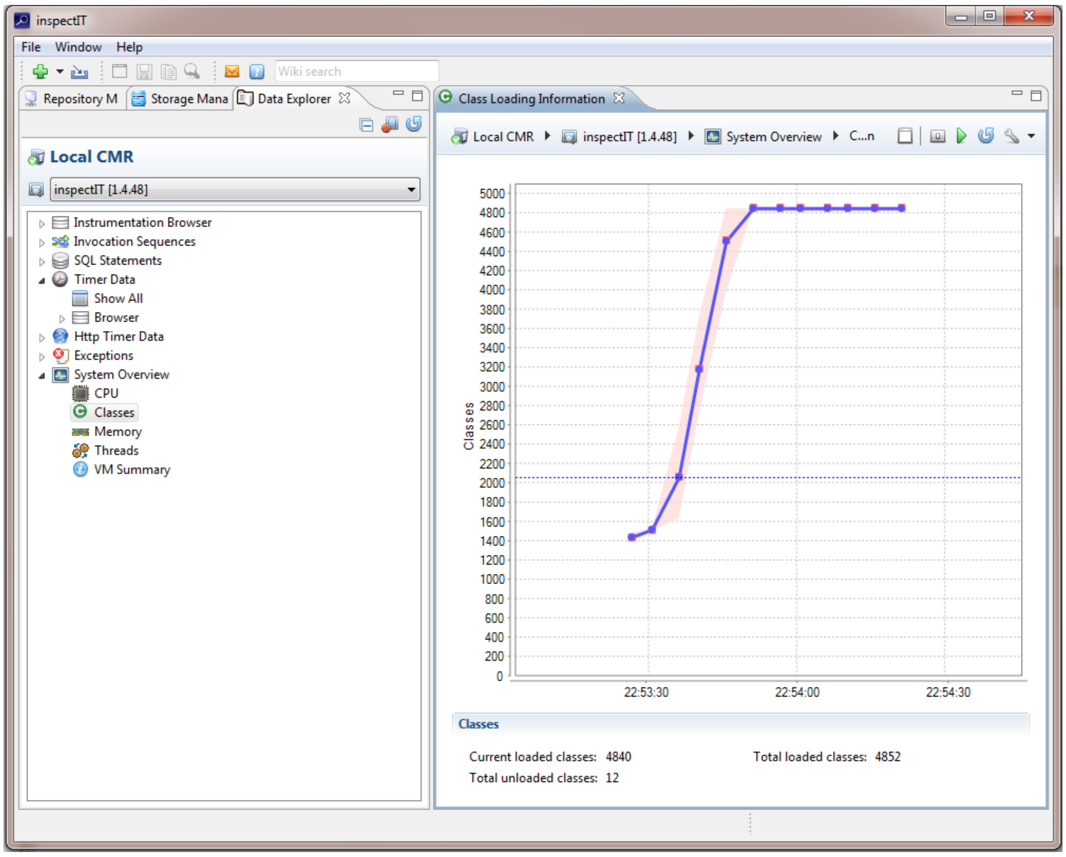
Agents are made to receive the data from an application. To collect data from an application it has to be started with an agent. Each agent needs a configuration file which defines which sensors shall be activated and which methods should be observated.

The GUI and its functions

The application comes as standalone version. The application to visualize the data has an graphical user interface. To set up the CMR or an Agent the command line is needed. The User Interface is made with Eclipse RCP and looks a bit like the Eclipse IDE.

Summary

InspectIT has a few abilities we already known from profilers. But this program isn't really made to profile an application. Its profiling functions like analyzing current loaded classes, memory usage, thread activity or CPU usage are in comparison to other profilers very simple. But the aim of the application wasn't to collect all information you can get it was the aim to produce less overhead. It wouldn't be fair to compare its functions with other profilers. But the application could be a nice monitor.



Classes

Current loaded classes: 4840
Total loaded classes: 4852
Total unloaded classes: 12

Figure 7.12. View over the GUI of the InspectIt profiler

7.6 jProfiler

JProfiler is a powerful tool with many features and nice visualizations of the measurement results you want to research. It is impressively because of the intuitive GUI. There are many on the fly hints and navigation instructions that are very helpful to mastering your tasks you want to do. This 'instruction pilot' is not only helpful, you need it at many points of the program, because it is possible that you loose the view in the variety of features.

The GUI

The program is distinguished by the clear GUI and the big intuitive symbols in the navigator and the toolbar. The GUI is separated in 3 main parts. The menu bar with the toolbar for the particular action, the navigator and the big main view. In the menu bar you have many options to easy control the system. The toolbar supports to do the right actions you have chosen in the navigator. The last part, the view part, shows you all results of your profiling. It is important that the visualization is very clear to do a right interpretation of the measured data. In the case of jProfiler the visualization is definitely clear.

The toolbar

The toolbar is separated in three parts. The Session part in which you have some options for the whole profiling session you temporary run, the Profiling part in which you can control your profiling and the last is the individual part called View specific. This last part changes every time you change the function you want to execute. Every function you can choose in the navigator includes other specific options you can use.

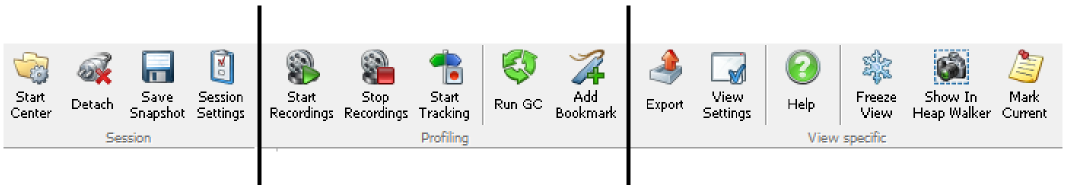


Figure 7.13. The toolbar with the particular sections marked by the black lines

The navigator

The navigator shows all the variety of actions you have with jProfiler. So in the figure below 7.14 you see a short overview of the navigator with section and subsections of the main actions.

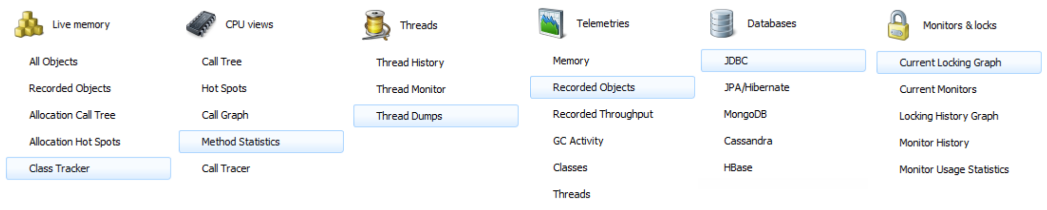


Figure 7.14. The Navigator to chose the particular action of the profiler

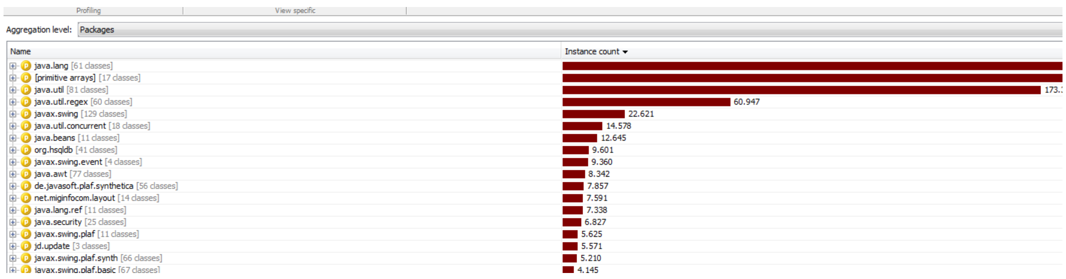
The exact functionality will be described in a later section.
The main view is the visualization part as mentioned. It works with tables, maps, call hierarchies, time lines and many more. Further more you have always the option to view the visualizations in a filtered version. For example in the memory profiling view you have the possibility to show a specific package you want to observe. In this package you can only show the corresponding methods which are running.

Main functions

Live Memory

In all sub-functions of the Live memory feature you can choose an so called aggregation level. This aggregation level is subdivided in classes, packages and Java EE Components. This should give you the opportunity to have different views on the memory usage of your profiled program.

▻ All Objects and Recorded Objects
The view of the All Objects Section and the Recorded Section is the same. If you take a look at Figure 7.15 you see the extendable list of packages and the includes class counts to each package. Now you can open the packages. If you now double click own of the classes you will get to a source view of the class.

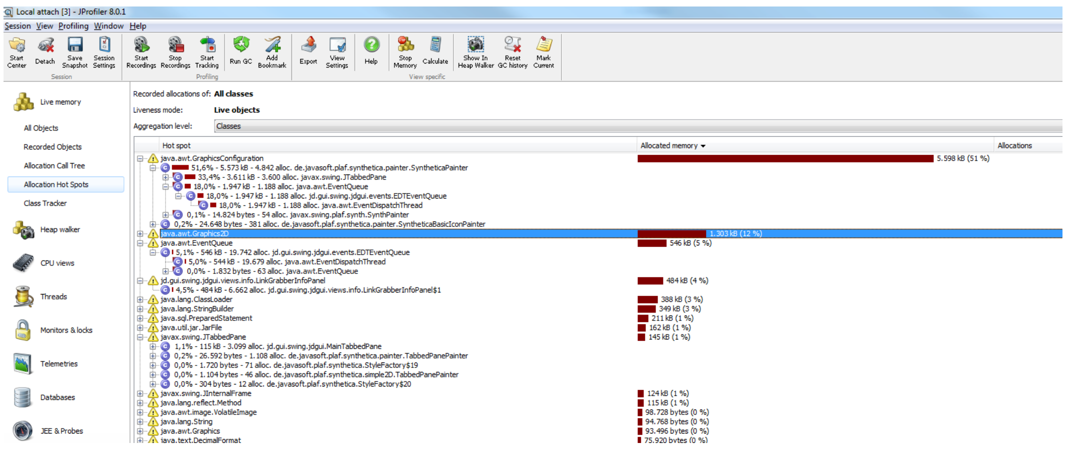


Aggregation level: Packages

Name	Instance count
java.lang [61 classes]	
[primitive arrays] [17 classes]	
java.util [81 classes]	173.:
java.util.regex [60 classes]	60.947
javax.swing [129 classes]	22.621
java.util.concurrent [18 classes]	14.578
java.beans [11 classes]	12.645
org.hsqldb [41 classes]	9.601
javax.swing.event [4 classes]	9.360
java.awt [77 classes]	8.342
de.javasoft.plaf.synthetica [56 classes]	7.857
net.miginfocom.layout [14 classes]	7.591
java.lang.ref [11 classes]	7.338
java.security [25 classes]	6.827
javax.swing.plaf [11 classes]	5.625
jd.update [3 classes]	5.571
javax.swing.plaf.synth [66 classes]	5.210
javax.swing.plaf.basic [67 classes]	4.145

Figure 7.15. Memory usage view

▻ Allocation Call Tree and Allocation Hot Spots
In this section you have the same choice of selecting an aggregation level, but in this case you have the levels methods, classes, packages and Java EE Components. The allocation call tree shows a top-down call tree accumulated for all threads and filtered according. The allocation hot spots view shows a list of methods where objects of a selected class have been allocated. Only methods which contribute at least 0.1 percent of the total number of allocations are included.



Recorded allocations of: **All classes**
Liveness mode: **Live objects**
Aggregation level: Classes

```
Hot spot                                                                                      Allocated memory        Allocations
java.awt.GraphicsConfiguration                                                                5.598 kB (51 %)
  51,6% - 5.573 kB - 4.842 alloc. de.javasoft.plaf.synthetica.painter.SyntheticaPainter
    33,4% - 3.611 kB - 3.600 alloc. javax.swing.JTabbedPane
    18,0% - 1.947 kB - 1.188 alloc. java.awt.EventQueue
      18,0% - 1.947 kB - 1.188 alloc. jd.gui.swing.jdgui.events.EDTEventQueue
        18,0% - 1.947 kB - 1.188 alloc. java.awt.EventDispatchThread
    0,1% - 14.824 bytes - 54 alloc. javax.swing.plaf.synth.SynthPainter
  0,2% - 24.648 bytes - 381 alloc. de.javasoft.plaf.synthetica.painter.SyntheticaBasicIconPainter
java.awt.Graphics2D                                                                           1.303 kB (12 %)
java.awt.EventQueue                                                                           546 kB (5 %)
  5,1% - 546 kB - 19.742 alloc. jd.gui.swing.jdgui.events.EDTEventQueue
    5,0% - 544 kB - 19.679 alloc. java.awt.EventDispatchThread
    0,0% - 1.832 bytes - 63 alloc. java.awt.EventQueue
jd.gui.swing.jdgui.views.info.LinkGrabberInfoPanel                                            484 kB (4 %)
  4,5% - 484 kB - 6.662 alloc. jd.gui.swing.jdgui.views.info.LinkGrabberInfoPanel$1
java.lang.ClassLoader                                                                         388 kB (3 %)
java.lang.StringBuilder                                                                       349 kB (3 %)
java.sql.PreparedStatement                                                                    211 kB (1 %)
java.util.jar.JarFile                                                                         162 kB (1 %)
javax.swing.JTabbedPane                                                                       145 kB (1 %)
  1,1% - 115 kB - 3.099 alloc. jd.gui.swing.jdgui.MainTabbedPane
  0,2% - 26.592 bytes - 1.108 alloc. de.javasoft.plaf.synthetica.painter.TabbedPanePainter
  0,0% - 1.720 bytes - 71 alloc. de.javasoft.plaf.synthetica.StyleFactory$19
  0,0% - 1.104 bytes - 46 alloc. de.javasoft.plaf.synthetica.simple2D.TabbedPanePainter
  0,0% - 304 bytes - 12 alloc. de.javasoft.plaf.synthetica.StyleFactory$20
javax.swing.JInternalFrame                                                                    124 kB (1 %)
java.lang.reflect.Method                                                                      115 kB (1 %)
java.awt.image.VolatileImage                                                                  98.728 bytes (0 %)
java.lang.String                                                                              94.768 bytes (0 %)
java.awt.Graphics                                                                             93.496 bytes (0 %)
java.text.DecimalFormat                                                                       75.920 bytes (0 %)
```

Figure 7.16. Allocation overview of the tested application

▻ Class Tracker
In the Class Tracker you can observe whole packages or only arrays. You get a time line that shows the progress of class counts in package or the value trend of an selected array.

Heap Walker

The Heap Walker gives you the opportunity to take a snapshot of the actual state of the memory. Now you can do a walkthrough to your memory. In Figure 7.17 you see the Heap Walker toolbar.

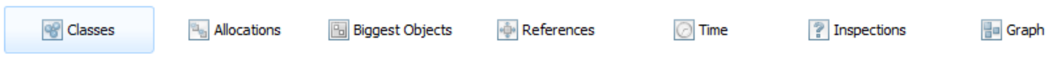


Figure 7.17. In the Heapwalker you can have a walk through the memory

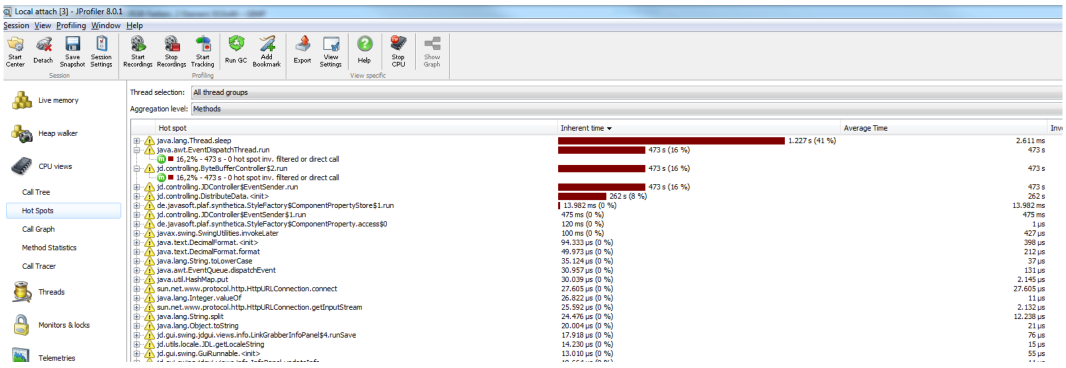
With the left three symbols of the toolbar you can chose at a time either a class, allocation or one of the biggest objects. Now you can research this selection in detail. For example you have the opportunity to explore the references or the time the selection has impact in the program. An additional feature is that you can show selected references in a graph.

CPU views

The This section you also have the aggregation level choice. Methods, classes, packages and Java EE Components are available.

▷ Call Tree
Analogue to the call tree for the memory it shows a tree of filtered or all threads with CPU percentage.

▷ Hot Spots
The hot spots view shows a list of calls of a selected type. The list is truncated at the point where calls use less than 0.1 percent of the total time of all calls.



Thread selection: All thread groups
Aggregation level: Methods

Hot spot	Inherent time	Average Time
java.lang.Thread.sleep	1.227 s (41 %)	2.611 ms
java.awt.EventDispatchThread.run	473 s (16 %)	473 s
16,2% - 473 s - 0 hot spot inv. filtered or direct call		
jd.controlling.ByteBufferController$2.run	473 s (16 %)	473 s
16,2% - 473 s - 0 hot spot inv. filtered or direct call		
jd.controlling.JDController$EventSender.run	473 s (16 %)	473 s
jd.controlling.DistributeData.<init>	262 s (8 %)	262 s
de.javasoft.plaf.synthetica.StyleFactory$ComponentPropertyStore$1.run	13.982 ms (0 %)	13.982 ms
jd.controlling.JDController$EventSender$1.run	475 ms (0 %)	475 ms
de.javasoft.plaf.synthetica.StyleFactory$ComponentProperty.access$0	120 ms (0 %)	1 µs
javax.swing.SwingUtilities.invokeLater	100 ms (0 %)	427 µs
java.text.DecimalFormat.<init>	94.333 µs (0 %)	398 µs
java.text.DecimalFormat.format	49.973 µs (0 %)	212 µs
java.lang.String.toLowerCase	35.124 µs (0 %)	37 µs
java.awt.EventQueue.dispatchEvent	30.957 µs (0 %)	131 µs
java.util.HashMap.put	30.039 µs (0 %)	2.145 µs
sun.net.www.protocol.http.HttpURLConnection.connect	27.605 µs (0 %)	27.605 µs
java.lang.Integer.valueOf	26.822 µs (0 %)	11 µs
sun.net.www.protocol.http.HttpURLConnection.getInputStream	25.592 µs (0 %)	2.132 µs
java.lang.String.split	24.476 µs (0 %)	12.238 µs
java.lang.Object.toString	20.004 µs (0 %)	21 µs
jd.gui.swing.jdgui.views.info.LinkGrabberInfoPanel$4.runSave	17.918 µs (0 %)	76 µs
jd.utils.locale.JDL.getLocaleString	14.230 µs (0 %)	15 µs
jd.gui.swing.GuiRunnable.<init>	13.010 µs (0 %)	55 µs

Figure 7.18. The CPU Hotspots. Here you see the main active threads in focus

▻ Call Graph
This feature generates call graph. You can choose threads that you want to display in this graph. In this graph you see the threads an fields which are in relation to the selected threads. It should illustrate how the thread call circle was executed. Additional to that there are some facts to each thread and field like the total calls and the total compute time.

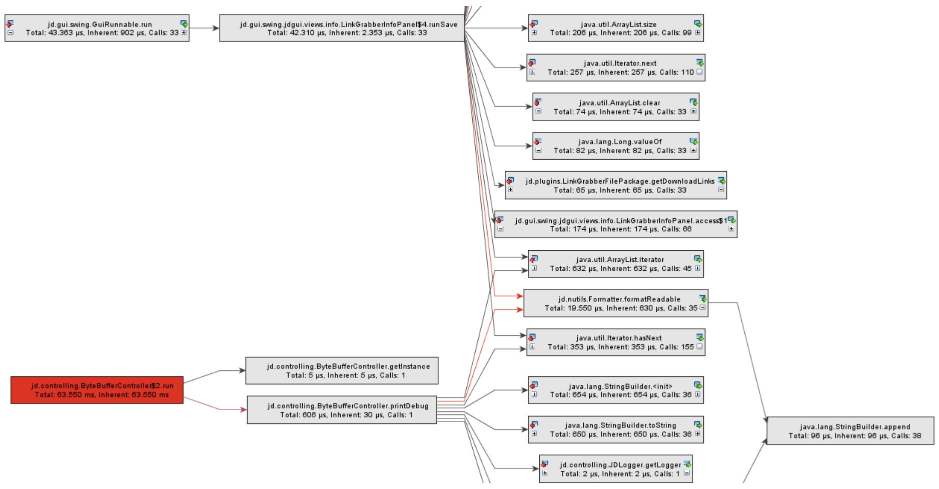

jd.gui.swing.GuiRunnable.run
Total: 43.363 µs, Inherent: 902 µs, Calls: 33
jd.gui.swing.jdgui.views.info.LinkGrabberInfoPanel$4.runSave
Total: 42.310 µs, Inherent: 2.353 µs, Calls: 33
java.util.ArrayList.size
Total: 206 µs, Inherent: 206 µs, Calls: 99
java.util.Iterator.next
Total: 257 µs, Inherent: 257 µs, Calls: 110
java.util.ArrayList.clear
Total: 74 µs, Inherent: 74 µs, Calls: 33
java.lang.Long.valueOf
Total: 82 µs, Inherent: 82 µs, Calls: 33
jd.plugins.LinkGrabberFilePackage.getDownloadLinks
Total: 65 µs, Inherent: 65 µs, Calls: 33
jd.gui.swing.jdgui.views.info.LinkGrabberInfoPanel.access$1
Total: 174 µs, Inherent: 174 µs, Calls: 66
java.util.ArrayList.iterator
Total: 632 µs, Inherent: 632 µs, Calls: 45
jd.nutils.Formatter.formatReadable
Total: 19.550 µs, Inherent: 630 µs, Calls: 35
java.util.Iterator.hasNext
Total: 353 µs, Inherent: 353 µs, Calls: 155
jd.controlling.ByteBufferController$2.run
Total: 63.550 ms, Inherent: 63.550 ms
jd.controlling.ByteBufferController.getInstance
Total: 5 µs, Inherent: 5 µs, Calls: 1
jd.controlling.ByteBufferController.printDebug
Total: 606 µs, Inherent: 30 µs, Calls: 1
java.lang.StringBuilder.<init>
Total: 654 µs, Inherent: 654 µs, Calls: 36
java.lang.StringBuilder.toString
Total: 650 µs, Inherent: 650 µs, Calls: 36
jd.controlling.JDLogger.getLogger
Total: 2 µs, Inherent: 2 µs, Calls: 1
java.lang.StringBuilder.append
Total: 96 µs, Inherent: 96 µs, Calls: 38

Figure 7.19. In the Call Graph you have the possibility to generate call graphs of selected threads

▻ Method Statistics
In this section you can start a recording. In this interval of start and stop recording you get a list of methods which were running in this time. Now you can choose one of them and go into the details.

▻ Call Tracer
The same recording action you got in the Method Statistics you have in the Call Tracer. Here you will get a extendable list of packages which have methods that were called at the recording interval.

Threads

▻ Thread History and Thread Monitor
This two sections visualizing the running threads in the program. In the History all running threads are displayed in a time line. By right clicking one of them you can go to the CPU view and check it in the call tree. In the Monitor you got a list of all threads. If you select one of them you get a list of methods involved in this thread. This methods you can double click and go straight to a source code viewer if it is possible to look into the classes.

- ▻ Thread Dump
  This function only creates a summary of all threads. At this point you also have the possibility to go into the source code.

Telemetries

In all time line diagrams you have the same set of options you can use to go more detailed into the set of data. You can set bookmarks, you can zoom in and zoom out and you can change the graph type from area to line graph view. Additional to that you have an export view function and a set of settings to personalize your time line diagrams.

- ▻ Memory
  - ▻ Heap
    This time line diagram shows the total heap usage. The used Heap and the free space of heap
  - ▻ Non Heap
    This time line diagram shows the code cache usage and the perm gen usage.
- ▻ Recorded Objects
  This time line shows what size of memory is used by arrays and what part used by non array typed objects.
- ▻ Recorded Throughput
  This diagram shows how many objects are collected of the GC and how many are created.
- ▻ GC Activity
  This diagram shows the live action of the garbage collector in percent
- ▻ Classes
  Shows the CPU computed and non CPU computed class counts
- ▻ Threads
  In this time line diagram you can see all Threads with the particular state in different colors.
- ▻ CPU Load
  Last but not least the time line diagram of the CPU load. Here you see the CPU computing percentage which the application needs.

7.7 YourKit Profiler

YourKit profiler is a commercial great tool to analyze applications. The application comes as standalone version with plugins for several IDEs [YourKit GmbH]. The plugins just starts the applications out of the IDE's. The standalone version visualizes the information in real time. The profiler is very powerful and comes with a huge set of functions. In comparison to other profilers with same range of functionality its very easy to install. Just execute the installer downloaded at the Homepage and that's it. After starting an application out of the IDE the profiler starts to visualize the information. To the functional scope belongs well known functions like CPU and memory profiling as well as innovative ones like inspections of leaks and memory wastes.

The plugin

The profiler comes with a standalone version and a corresponding plugin for several IDEs like Eclipse, IntelliJ IDEA, NetBeans and JDeveloper. The plugin is very simple and only starts the application with the YourKit Profiler

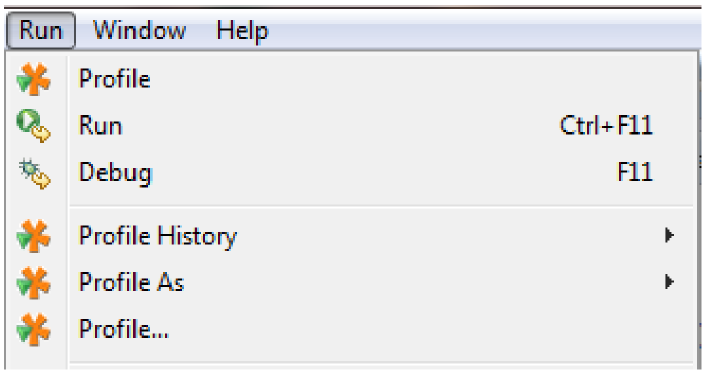
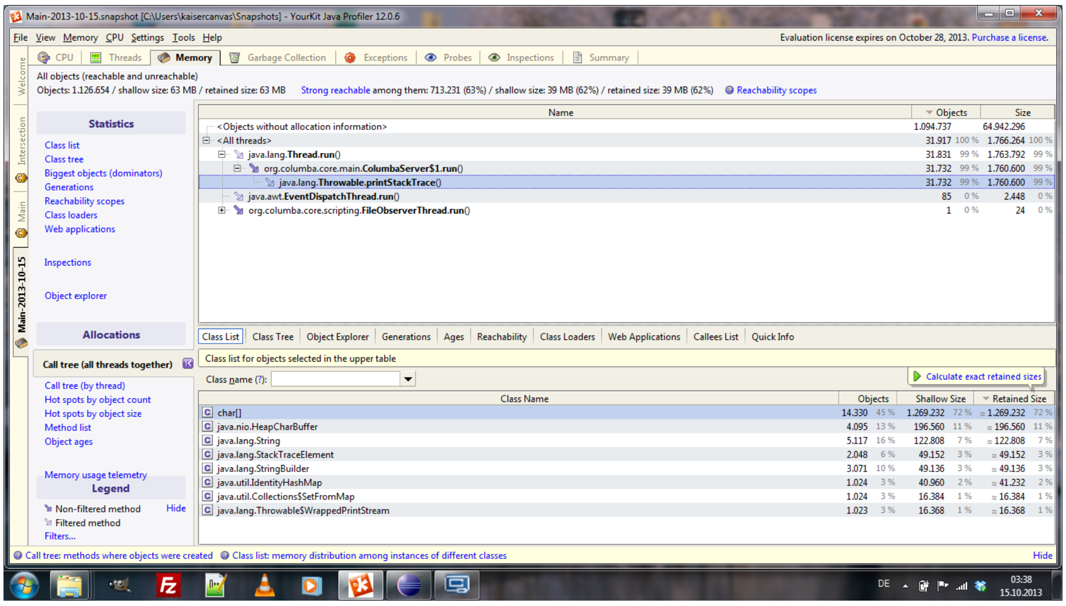


Figure 7.20. Overview of the Eclipse Yourkit profiler

The figure shows the menu item *run* at Eclipse Juno. Besides the menu entry *profile* , which starts the standalone program and a selected project at eclipse, the plugin appends a profile history and a *profile as*function, which is similar to the well known function *run as* unless it also starts the profiler.

The standalone application

The standalone version starts with a welcome screen. The welcome screen shows a few important functions and helps to come familiar with the program. Here the user can find the documentation, an example program which can be started out of the application to test the profiler for the first time or a list of the current detected java application running on the system. Furthermore there are an integration wizard to install the plugin into the used IDE, a function to open an existing snapshot to analyze older recorded data or even a function to connect to a remote application.


File View Memory CPU Settings Tools Help
CPU Threads Memory Garbage Collection Exceptions Probes Inspections Summary
Welcome Intersection Main Main-2013-10-15
Statistics
Class list
Class tree
Biggest objects (dominators)
Generations
Reachability scopes
Class loaders
Web applications
Inspections
Object explorer
Allocations
Call tree (all threads together)
Call tree (by thread)
Hot spots by object count
Hot spots by object size
Method list
Object ages
Memory usage telemetry
Legend
Non-filtered method
Hide
Filtered method
Filters...
Class List Class Tree Object Explorer Generations Ages Reachability Class Loaders Web Applications Callees List Quick Info
Calculate exact retained sizes
Hide
DE
03:38
15.10.2013

Main-2013-10-15.snapshot [C:\Users\kaisercanvas\Snapshots] - YourKit Java Profiler 12.0.6

Evaluation license expires on October 28, 2013. Purchase a license.

All objects (reachable and unreachable)
Objects: 1.126.654 / shallow size: 63 MB / retained size: 63 MB   Strong reachable among them: 713.231 (63%) / shallow size: 39 MB (62%) / retained size: 39 MB (62%)   Reachability scopes

Name	Objects		Size	
<Objects without allocation information>	1.094.737		64.942.296	
<All threads>	31.917	100 %	1.766.264	100 %
java.lang.Thread.run()	31.831	99 %	1.763.792	99 %
org.columba.core.main.ColumbaServer$1.run()	31.732	99 %	1.760.600	99 %
java.lang.Throwable.printStackTrace()	31.732	99 %	1.760.600	99 %
java.awt.EventDispatchThread.run()	85	0 %	2.448	0 %
org.columba.core.scripting.FileObserverThread.run()	1	0 %	24	0 %

Class list for objects selected in the upper table

Class name (?):

Class Name	Objects		Shallow Size		Retained Size	
char[]	14.330	45 %	1.269.232	72 %	≈ 1.269.232	72 %
java.nio.HeapCharBuffer	4.095	13 %	196.560	11 %	≈ 196.560	11 %
java.lang.String	5.117	16 %	122.808	7 %	≈ 122.808	7 %
java.lang.StackTraceElement	2.048	6 %	49.152	3 %	≈ 49.152	3 %
java.lang.StringBuilder	3.071	10 %	49.136	3 %	≈ 49.136	3 %
java.util.IdentityHashMap	1.024	3 %	40.960	2 %	≈ 41.232	2 %
java.util.Collections$SetFromMap	1.024	3 %	16.384	1 %	≈ 16.384	1 %
java.lang.Throwable$WrappedPrintStream	1.023	3 %	16.368	1 %	≈ 16.368	1 %

Call tree: methods where objects were created   Class list: memory distribution among instances of different classes

Figure 7.21. Yourkit memory usage analysis

By the help of the list of the current running java applications we can come to the analyze view of an application. Here we have a toolbar to record, clear or stop data of the profiled application. At the bottom of the toolbar the program shows a tab-bar which offers access to all common functions. A detailed list of all tabs and its functions was made bellow.

Functions

▷ **CPU**
This view shows two diagrams at the top of the panel. One illustrates an overview of the current used CPU in percent, the other one the amount of current sleeping and running threads. The sampling of the CPU can be adjust at the settings menu. A detailed list can be found at the bottom of the panel. A *Call Tree* offers the function to navigate though the methods of the application and *Stack Traces* can be found there too.

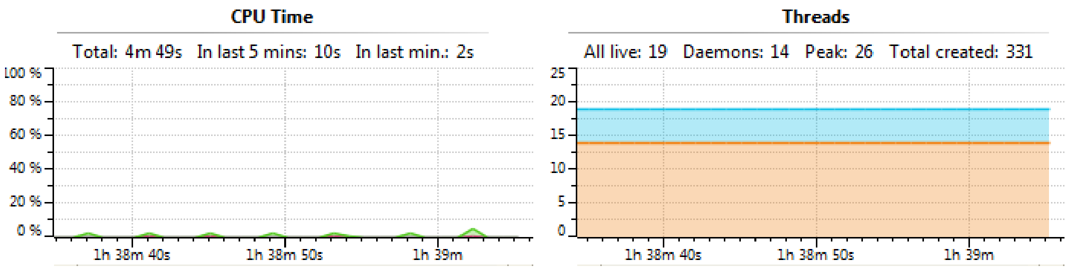

CPU Time
Total: 4m 49s In last 5 mins: 10s In last min.: 2s
100 %
80 %
60 %
40 %
20 %
0 %
1h 38m 40s
1h 38m 50s
1h 39m
Threads
All live: 19 Daemons: 14 Peak: 26 Total created: 331
25
20
15
10
5
0
1h 38m 40s
1h 38m 50s
1h 39m

Figure 7.22. Yourkit CPU usage analysis

▷ **Threads**
This view shows the current running threads. It shows a small table that visualizes the states of the threads in real time. The states could be *runnable*, *blocked*, *sleeping* and *waiting*. Beneath the table a small figure shows the CPU usage.

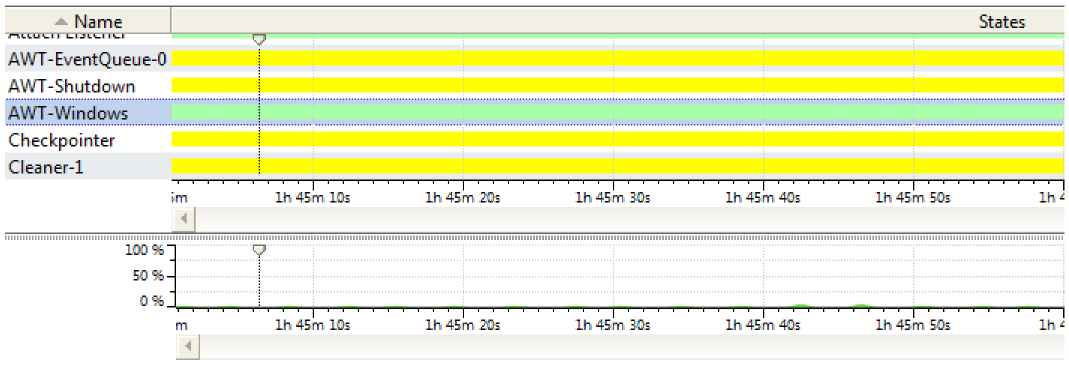
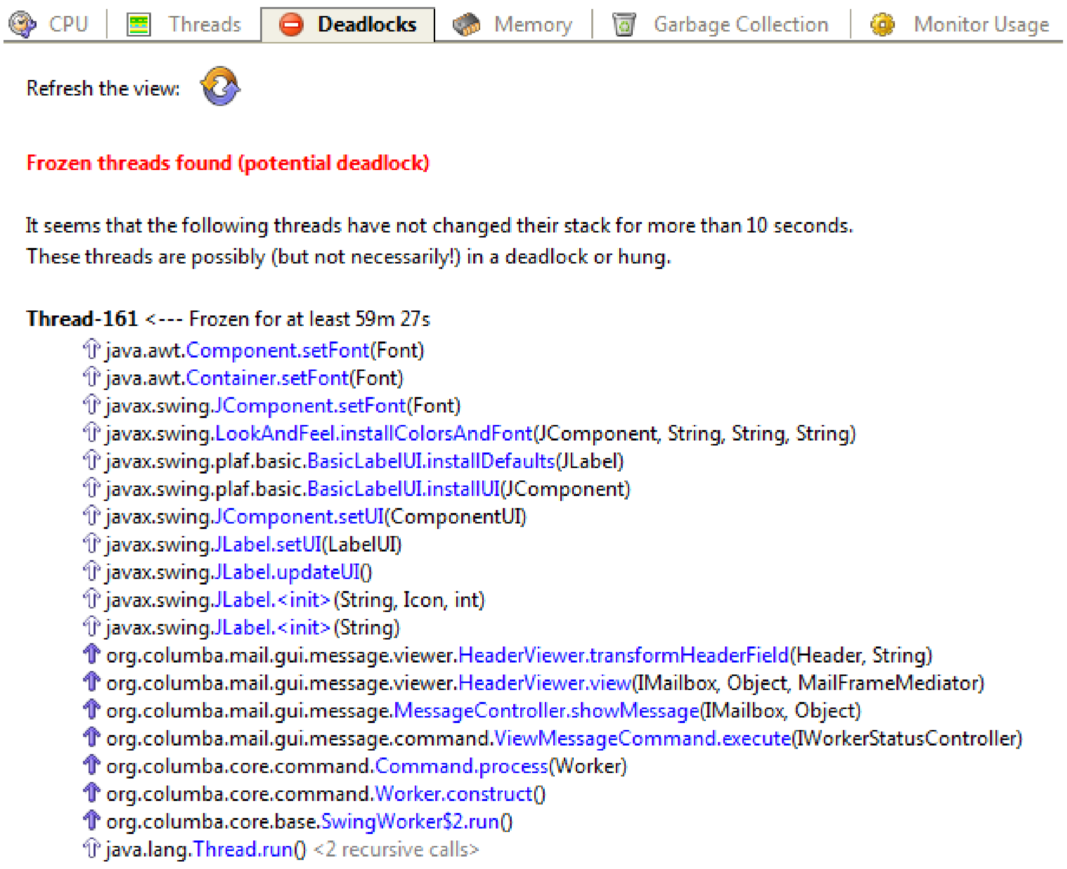


Figure 7.23. Thread overview of the observed application

To get detailed information of a thread and its stack trace at a specific time a click at the table appends a slider. A table with more information is shown at the bottom of the panel when the slider was added.

▻ **Deadlock**
A helpful function can be reached through the tab *Deadlock*. The profiler recognizes Threads that seems to be in a deadlock. That means the profiler takes a look at the stack of the threads. If a thread doesn't change its stack for more than ten seconds it is probable a deadlock.



Frozen threads found (potential deadlock)

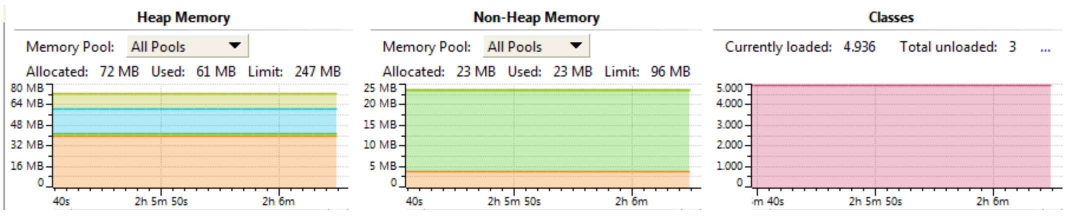
It seems that the following threads have not changed their stack for more than 10 seconds.
These threads are possibly (but not necessarily!) in a deadlock or hung.

```
Thread-161 <--- Frozen for at least 59m 27s
    java.awt.Component.setFont(Font)
    java.awt.Container.setFont(Font)
    javax.swing.JComponent.setFont(Font)
    javax.swing.LookAndFeel.installColorsAndFont(JComponent, String, String, String)
    javax.swing.plaf.basic.BasicLabelUI.installDefaults(JLabel)
    javax.swing.plaf.basic.BasicLabelUI.installUI(JComponent)
    javax.swing.JComponent.setUI(ComponentUI)
    javax.swing.JLabel.setUI(LabelUI)
    javax.swing.JLabel.updateUI()
    javax.swing.JLabel.<init>(String, Icon, int)
    javax.swing.JLabel.<init>(String)
    org.columba.mail.gui.message.viewer.HeaderViewer.transformHeaderField(Header, String)
    org.columba.mail.gui.message.viewer.HeaderViewer.view(IMailbox, Object, MailFrameMediator)
    org.columba.mail.gui.message.MessageController.showMessage(IMailbox, Object)
    org.columba.mail.gui.message.command.ViewMessageCommand.execute(IWorkerStatusController)
    org.columba.core.command.Command.process(Worker)
    org.columba.core.command.Worker.construct()
    org.columba.core.base.SwingWorker$2.run()
    java.lang.Thread.run() <2 recursive calls>
```

Figure 7.24. List of dead threads that seemed to be dead

The view shows all deadlocks and its stack trace.

▻ **Memory**
One of the main functions which is usually for a profiler is to show current memory usage. This profiler offers three diagrams that show heap- and non-heap-usage of the current application as well as the amount of current loaded classes. The table below offers some functions to search for specific classes to get an amount of its instances and memory usage as well as memory allocations.


Heap Memory
Memory Pool: All Pools
Allocated: 72 MB Used: 61 MB Limit: 247 MB

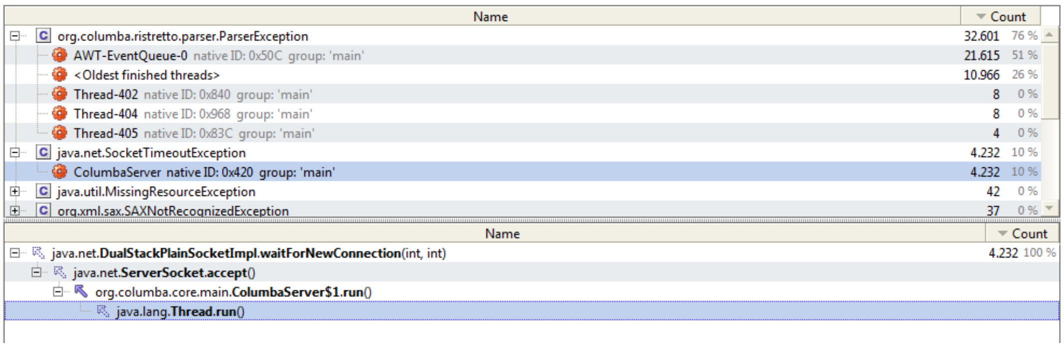
Non-Heap Memory
Memory Pool: All Pools
Allocated: 23 MB Used: 23 MB Limit: 96 MB

Classes
Currently loaded: 4.936 Total unloaded: 3 ...

Figure 7.25. Yourkit Memory observation

▻ **Garbage Collection**
The next function visualizes the garbage collector activity. It will help to estimate garbage collection load.

▻ **Exceptions**
Another helpful view can be reached over the *Exceptions* tab. The exceptions may be grouped by their exception class or by thread. The second table shows the stack trace of the current selected exception at the first table. It is possible to filter for specific exceptions.

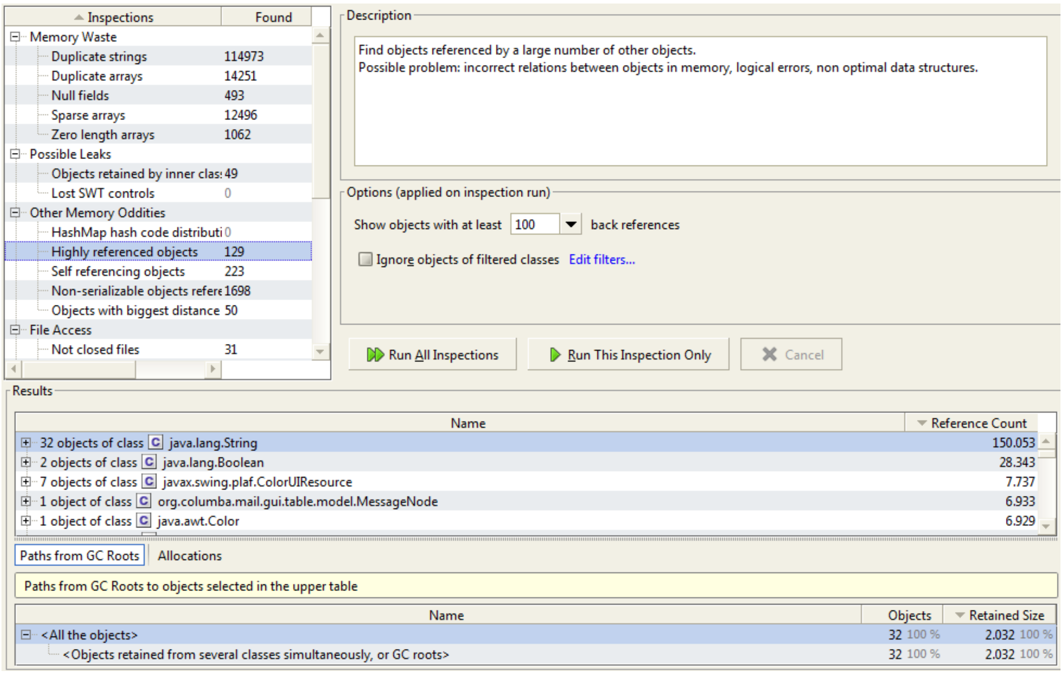


Name	Count	
org.columba.ristretto.parser.ParserException	32.601	76 %
AWT-EventQueue-0 native ID: 0x50C group: 'main'	21.615	51 %
<Oldest finished threads>	10.966	26 %
Thread-402 native ID: 0x840 group: 'main'	8	0 %
Thread-404 native ID: 0x968 group: 'main'	8	0 %
Thread-405 native ID: 0x83C group: 'main'	4	0 %
java.net.SocketTimeoutException	4.232	10 %
ColumbaServer native ID: 0x420 group: 'main'	4.232	10 %
java.util.MissingResourceException	42	0 %
org.xml.sax.SAXNotRecognizedException	37	0 %

Name	Count	
java.net.**DualStackPlainSocketImpl.waitForNewConnection**(int, int)	4.232	100 %
java.net.**ServerSocket.accept**()		
org.columba.core.main.**ColumbaServer$1.run**()		
java.lang.**Thread.run**()		

Figure 7.26. Recognized Exceptions of the application under test

▻ **Inspections**
Possibly, the most innovative function this profiler offers. This function is only enabled after profiling an application and scans the application snapshots that could be saved after profiling. It inspects the application and tries to find code-segments that could be improved. It tests various possible faults and problems.



Inspections	Found
Memory Waste	
Duplicate strings	114973
Duplicate arrays	14251
Null fields	493
Sparse arrays	12496
Zero length arrays	1062
Possible Leaks	
Objects retained by inner clas	49
Lost SWT controls	0
Other Memory Oddities	
HashMap hash code distributi	0
Highly referenced objects	129
Self referencing objects	223
Non-serializable objects refere	1698
Objects with biggest distance	50
File Access	
Not closed files	31

Description
Find objects referenced by a large number of other objects.
Possible problem: incorrect relations between objects in memory, logical errors, non optimal data structures.

Options (applied on inspection run)
Show objects with at least 100 back references
Ignore objects of filtered classes Edit filters...

Run All Inspections | Run This Inspection Only | Cancel

Results

Name	Reference Count
32 objects of class C java.lang.String	150.053
2 objects of class C java.lang.Boolean	28.343
7 objects of class C javax.swing.plaf.ColorUIResource	7.737
1 object of class C org.columba.mail.gui.table.model.MessageNode	6.933
1 object of class C java.awt.Color	6.929

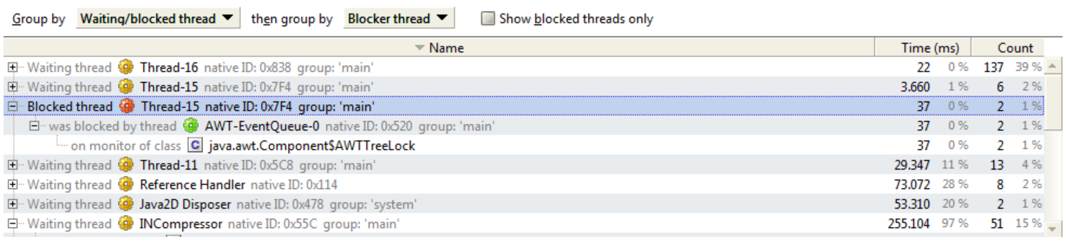
Paths from GC Roots | Allocations

Paths from GC Roots to objects selected in the upper table

Name	Objects		Retained Size	
<All the objects>	32	100 %	2.032	100 %
<Objects retained from several classes simultaneously, or GC roots>	32	100 %	2.032	100 %

Figure 7.27. Yourkit Inspection view

▻ **Monitoring**
The view *Monitor Usage* helps to find synchronization issues at the application. It visualizes dependencies between the threads. That means it shows which thread calling the wait function and blocks other threads.



Group by Waiting/blocked thread then group by Blocker thread Show blocked threads only

Name	Time (ms)		Count	
Waiting thread Thread-16 native ID: 0x838 group: 'main'	22	0 %	137	39 %
Waiting thread Thread-15 native ID: 0x7F4 group: 'main'	3.660	1 %	6	2 %
Blocked thread Thread-15 native ID: 0x7F4 group: 'main'	37	0 %	2	1 %
was blocked by thread AWT-EventQueue-0 native ID: 0x520 group: 'main'	37	0 %	2	1 %
on monitor of class C java.awt.Component$AWTTreeLock	37	0 %	2	1 %
Waiting thread Thread-11 native ID: 0x5C8 group: 'main'	29.347	11 %	13	4 %
Waiting thread Reference Handler native ID: 0x114	73.072	28 %	8	2 %
Waiting thread Java2D Disposer native ID: 0x478 group: 'system'	53.310	20 %	2	1 %
Waiting thread INCompressor native ID: 0x55C group: 'main'	255.104	97 %	51	15 %

Figure 7.28. Yourkit Monitoring features

▻ **Other functions** Other functions are Probes, where specific profile tests can be enabled or disabled and Summary which gives a short overview of all collected information.

Summary

YourKit is a very useful profiler which comes along with a lot of well developed functions. All well-known basic functions are included and work well. The profiler is very easy to install and easy to understand after short practice. Sometimes the visualization is a bit ordinary so that a better visualization would be desirable. A plugin can be included in various IDE's and works suitable. The function *Inspections* is very innovative and can help a lot to improve the performance of an application. All in all a very nice commercial product.

7.8 Dynatrace

Dynatrace is a commercial, very powerful profiler tool.

The environment

The Dynatrace environment consists of several components

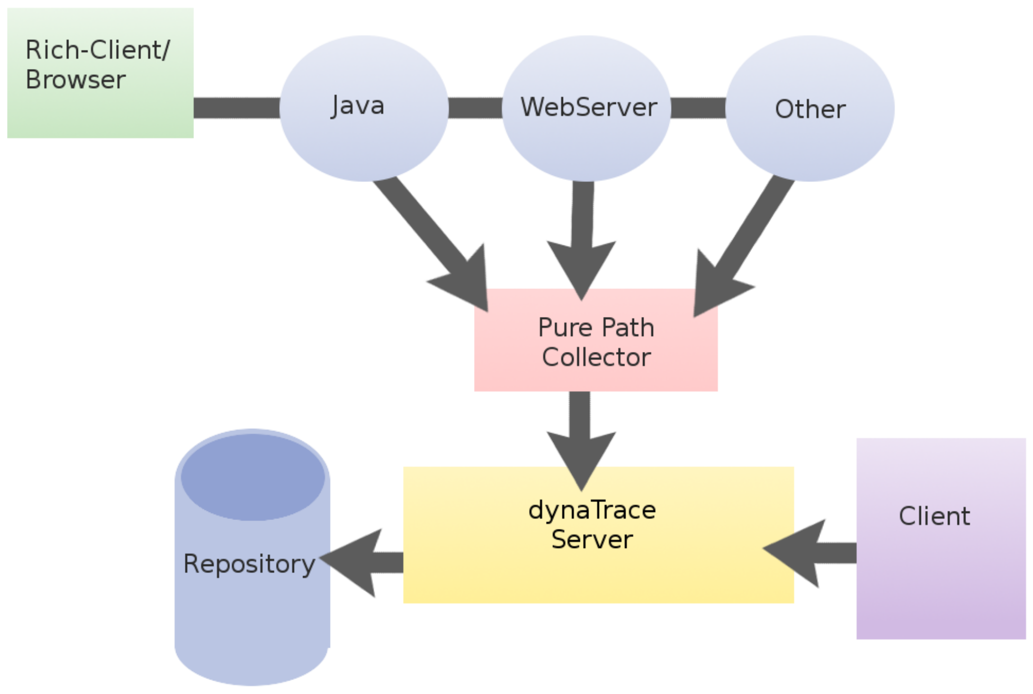


Figure 7.29. The DynaTrace environment

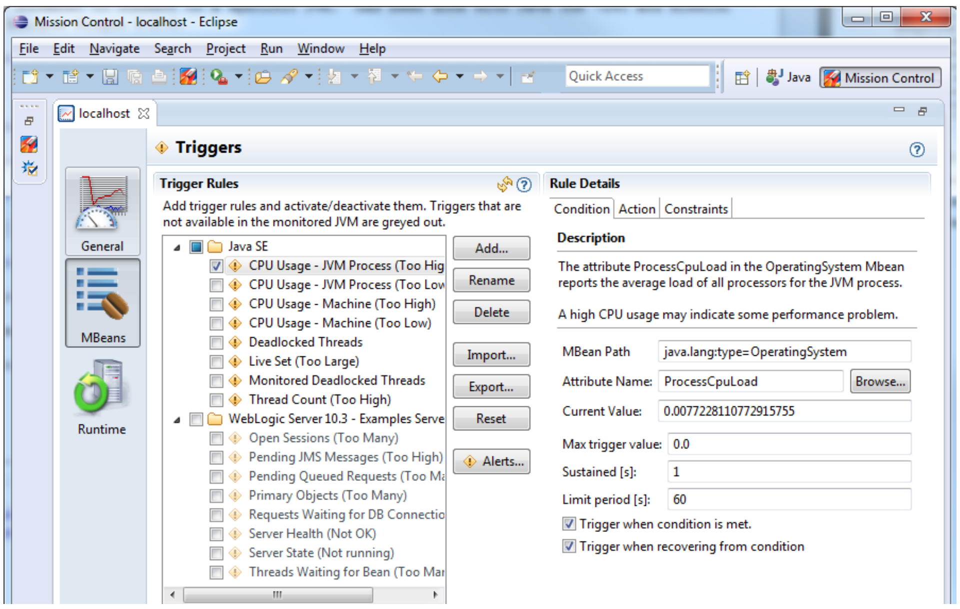
The main component is visualized in the figure above as the Dynatrace Server. This component holds the central configuration and administration. The Server saves the information it receives from the pure path controller into the repository. Beside getting information from the pure path controller the server is allowed to receive information from a monitoring collector too (not shown in the figure). The Dynatrace client is responsible for the presentation layer and visualizes the collected information. Dynatrace works with sensors to collect various data from the applications.

Summary

DynaTrace is one of the most powerful profiler/monitoring tools. But we didn't get any license to test this tool. Without testing its not possible to make serious statement. Anyway if you want to profile a big software system you definitely should take a closer look at this software.

7.9 Java Mission Control

As of recently, JRockit Mission Control and the JRockit JVM have been both merged into Hotspot JVM JMC. Mission Control is now included in Java starting Java 7u40 so there is no need to download additional tools. One can optionally get the eclipse plugin which additionally brings the feature to jump to source code locations. The oracle website has the download for Mission Control Eclipse Plugin split at two different places, so you have to make sure to pick the newer version of it. JRockit Mission Control looks very good on startup. The user interface is not overloaded and has a limited amount of actions at startup, foremost to connect to a specific JVM. When starting the Flight Recorder, Mission Control shows its full potential. One can set very specific the level of detail and thus the overhead of the record. The Flight Recorder records a wide variety of information starting from class loading statistics over method profiling till system processes. After a recording has finished or even during the recording, one can take a look at the generated report which - additionally to its recorded information - has statistics generated using various kind of graphs. There is also an interesting Trigger-feature, which allows the developer to make certain actions happen when certain conditions are met. For example one can set to automatically start deep level recording when CPU usage is above 50 percent. This is however more of a monitoring feature.
Testing has been done with Java Java Development Kit (JDK) 7u40 and Mission Control Eclipse Plugin 5.2.0.



Add trigger rules and activate/deactivate them. Triggers that are not available in the monitored JVM are greyed out.

- Java SE
  - [✓] CPU Usage - JVM Process (Too Hig
  - [ ] CPU Usage - JVM Process (Too Low
  - [ ] CPU Usage - Machine (Too High)
  - [ ] CPU Usage - Machine (Too Low)
  - [ ] Deadlocked Threads
  - [ ] Live Set (Too Large)
  - [ ] Monitored Deadlocked Threads
  - [ ] Thread Count (Too High)
- WebLogic Server 10.3 - Examples Serve
  - [ ] Open Sessions (Too Many)
  - [ ] Pending JMS Messages (Too High)
  - [ ] Pending Queued Requests (Too M
  - [ ] Primary Objects (Too Many)
  - [ ] Requests Waiting for DB Connectio
  - [ ] Server Health (Not OK)
  - [ ] Server State (Not running)
  - [ ] Threads Waiting for Bean (Too Ma

Description

The attribute ProcessCpuLoad in the OperatingSystem Mbean reports the average load of all processors for the JVM process.

A high CPU usage may indicate some performance problem.

MBean Path	java.lang:type=OperatingSystem
Attribute Name:	ProcessCpuLoad
Current Value:	0.0077228110772915755
Max trigger value:	0.0
Sustained [s]:	1
Limit period [s]:	60

[✓] Trigger when condition is met.
[✓] Trigger when recovering from condition

Figure 7.30. Overview of the triggers

CPU Profiling

The overview gives the user a very quick hint on the current state of the application. But the information shown here is pretty generic.

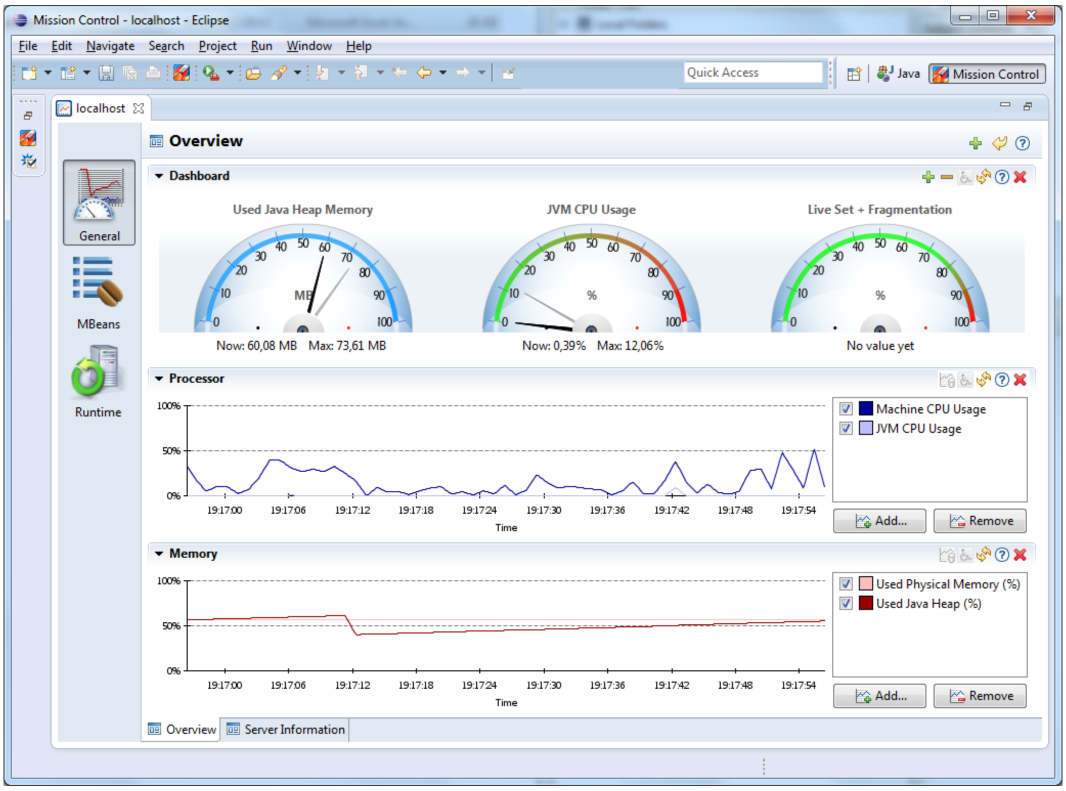
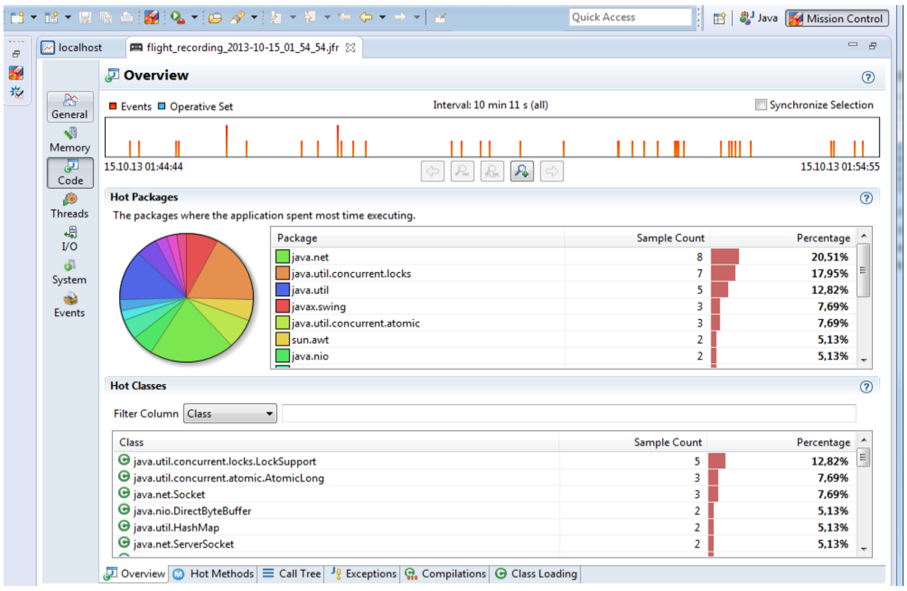


Figure 7.31. Overview of Java Mission Control

A report shows more detailed information regarding CPU: The amount of method calls, the method in question, the time the execution used up and more. Fortunately one can directly jump to the source code location when using the eclipse plugin.

Hot methods show methods which are called most frequently and take most time. So it basically shows the methods along the methods cumulative time being active. There is also the call tree showing the stack trace of methods.



Hot Packages

The packages where the application spent most time executing.

Package	Sample Count	Percentage
🟩 java.net	8	20,51%
🟧 java.util.concurrent.locks	7	17,95%
🟦 java.util	5	12,82%
🟥 javax.swing	3	7,69%
🟩 java.util.concurrent.atomic	3	7,69%
🟨 sun.awt	2	5,13%
🟩 java.nio	2	5,13%

Hot Classes

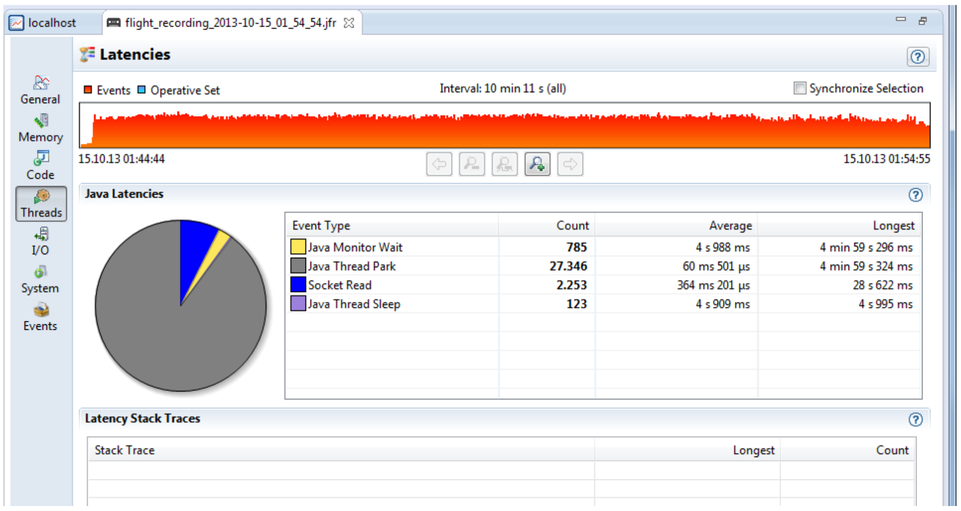
Filter Column Class

Class	Sample Count	Percentage
java.util.concurrent.locks.LockSupport	5	12,82%
java.util.concurrent.atomic.AtomicLong	3	7,69%
java.net.Socket	3	7,69%
java.nio.DirectByteBuffer	2	5,13%
java.util.HashMap	2	5,13%
java.net.ServerSocket	2	5,13%

Figure 7.32. Java Mission Control code report feature

Thread Profiling

Thread profiling in real time provides most information needed by developers. One can take a look at the stacktrace, the graph of active threads or the state of each individual thread whether it is running, blocked or waiting. The view pleases the user by being neatly structured. Even more information is available when opening a recording, which, sadly, has no live update feed.



Interval: 10 min 11 s (all)

15.10.13 01:44:44 — 15.10.13 01:54:55

Java Latencies

Event Type	Count	Average	Longest
🟨 Java Monitor Wait	785	4 s 988 ms	4 min 59 s 296 ms
⬛ Java Thread Park	27.346	60 ms 501 µs	4 min 59 s 324 ms
🟦 Socket Read	2.253	364 ms 201 µs	28 s 622 ms
🟪 Java Thread Sleep	123	4 s 909 ms	4 s 995 ms

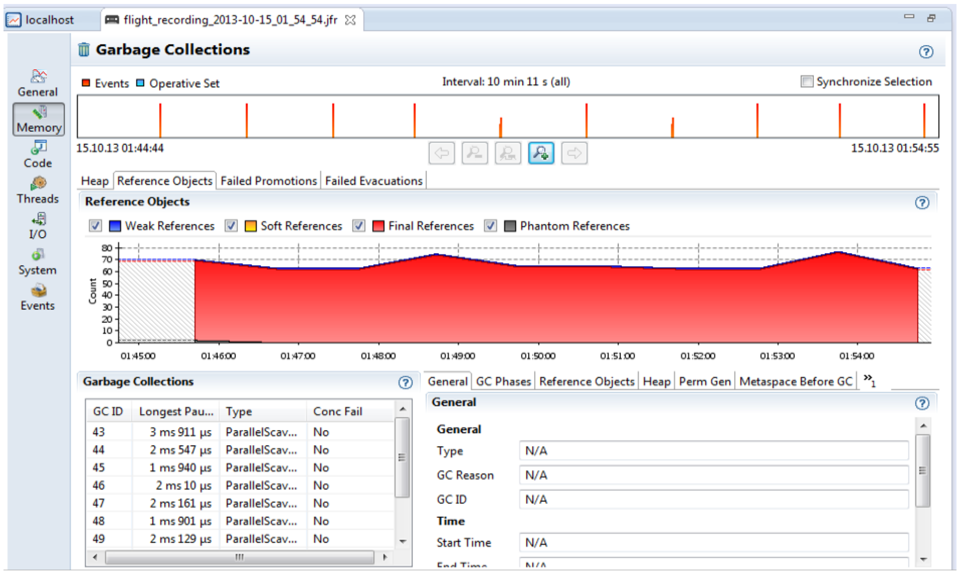
Latency Stack Traces

Stack Trace	Longest	Count

Figure 7.33. Java Mission Control Thread observation

Memory Profiling

Memory Profiling is another thing Mission Control presents greatly. The overview presents most needed information in one single graph, while the bottom tabs show more detailed information.



Interval: 10 min 11 s (all)

15.10.13 01:44:44 — 15.10.13 01:54:55

Garbage Collections

GC ID	Longest Pau...	Type	Conc Fail
43	3 ms 911 µs	ParallelScav...	No
44	2 ms 547 µs	ParallelScav...	No
45	1 ms 940 µs	ParallelScav...	No
46	2 ms 10 µs	ParallelScav...	No
47	2 ms 161 µs	ParallelScav...	No
48	1 ms 901 µs	ParallelScav...	No
49	2 ms 129 µs	ParallelScav...	No

General

General	
Type	N/A
GC Reason	N/A
GC ID	N/A
Time	
Start Time	N/A
End Time	N/A

Figure 7.34. Java Mission Control Memory Report

Rating

Mission Control is clearly more of a monitoring tool rather than a profiler. But still it does many profiling features better than some dedicated profiler tools.

- ▻ Need to tamper with JVM arguments to start profiling (Workflow -1)
- ▻ Only compatible starting Java 7u2. Earlier versions need specific JRockit VM (Ergonomy -1)
- ▻ Nice "speedometer" visualizations which give the user a quick hint on the state of the application (Usability +1)
- ▻ User interface is designed pretty well and complies with the eclipse user interface guideline (Likability +1)
- ▻ Very detailed information presented in a structured way (Usability +1)
- ▻ Need to start Flight Recorder in order to get detailed information which in turn is not updated in real time (Workflow -1)
- ▻ Overhead very low and can be set to ones needs (Overhead +5)

Summary

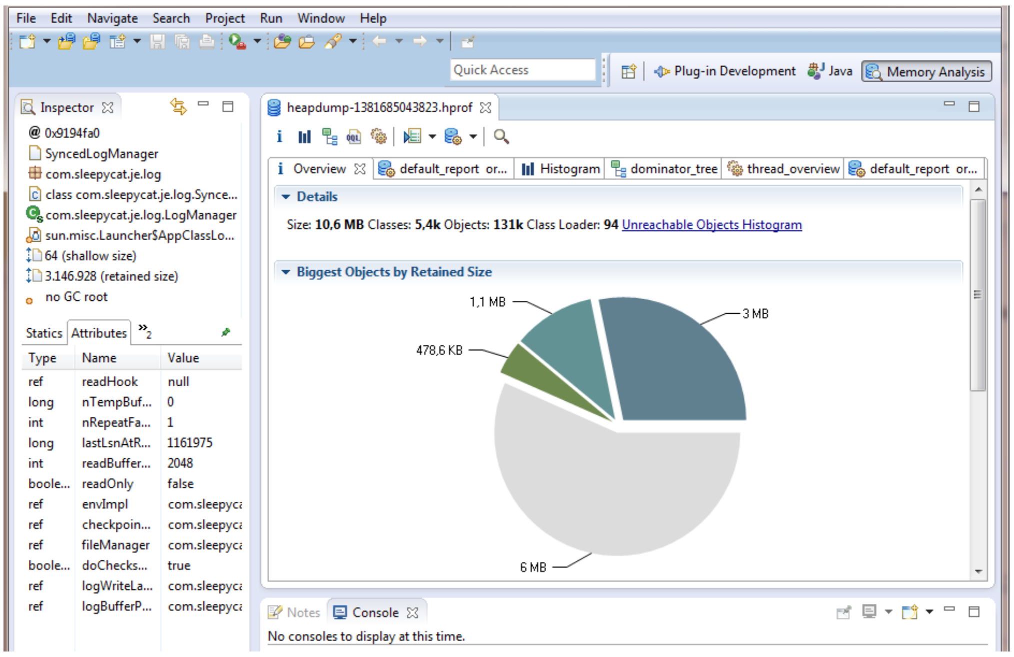
The need to tamper with JVM arguments to start profiling is inconvenient and has a negative impact on workflow. This feature would be more useful for the task of monitoring. Mission Control is only compatible with Java starting version 7u2, earlier versions need a specific JRockit JVM. Also one needs to start the Flight Recorder of Mission Control in order to start gaining detailed information about the AuT. And that data is not updated in real time on the user interface but rather saved in a log file. It is clear that Java Mission Control is developed in regard to monitoring rather than the task of profiling. However Java Mission Control has a very likable user interface. Especially the *speedometer* visualization is a good eye candy. It gives the user an instant impression on the current state of the application. The partly very detailed information is presented in a very structured way. Also the overhead can be set to any detail level, making the overhead range from practically negligible to huge depending on the user's needs and the amount of data needed. The overhead is also handled very efficiently, which gives Java Mission Control a perfect score at this aspect. However this study is about profilers so Java Mission Control is not its native environment.

7.10 Eclipse Memory Analyzer

Memory Analyzer (MAT) is a memory analyzing tool which helps to visualize heap dumps. Since Java 2 Platform Editon a simple command line profiling tool called HPROF have been added. The output files that hprof creates can be read by Eclipse MAT. By supplying HPROF options at startup, users can request various types of heap or CPU profiling features from HPROF. Besides HPROF it's also possible to read IBM dump files. A Java heap dump is a snapshot of the complete Java object graph at a certain point in time. It includes all objects, fields, primitive types and object references.

The GUI and its functions

Eclipse MAT comes of course as a plugin for Eclipse IDE. The plugin firstly looks very simple and the administration is a bit confusing.



@ 0x9194fa0
SyncedLogManager
com.sleepycat.je.log
class com.sleepycat.je.log.Synce...
com.sleepycat.je.log.LogManager
sun.misc.Launcher$AppClassLo...
64 (shallow size)
3.146.928 (retained size)
no GC root

Statics | Attributes | »2

Type	Name	Value
ref	readHook	null
long	nTempBuf...	0
int	nRepeatFa...	1
long	lastLsnAtR...	1161975
int	readBuffer...	2048
boole...	readOnly	false
ref	envImpl	com.sleepyca
ref	checkpoin...	com.sleepyca
ref	fileManager	com.sleepyca
boole...	doChecks...	true
ref	logWriteLa...	com.sleepyca
ref	logBufferP...	com.sleepyca

Size: **10,6 MB** Classes: **5,4k** Objects: **131k** Class Loader: **94** Unreachable Objects Histogram

No consoles to display at this time.

Figure 7.35. Overview of the Eclipse Memory Analyzer

The plugin comes with a own perspective view called Memory Analysis. This perspective view consists of a few views.

- ▻ Inspector
  This view shows detailed information of Java Objects when hovering parts of pie charts or clicking on items at the main view. Detailed information could be for example the memory address of an array, its class hierarchy or content or even the values of its attributes.

- ▻ Main View: Heap Dump analysis
  The main view visualizes all information covered by the heap dump. With the menu bar at the top of this view its possible to open different analysis.
  - ▻ Overview
    Shows general information of the heap dump like needed memory, amount of classes and objects or biggest objects of the corresponding application.
  - ▻ Histogram
    Lists number of instances per class.
  - ▻ Dominator Tree
    visualizes a tree which contains objects and its child objects. Its possible to walk trough the tree to analyze the shallow and retained heap of each object.
  - ▻ Object Query Language
    MAT offers the possibility to search for objects with a sql-like language called Object Query language (OQL). That allows to query the heap dump. OQL represents classes as tables, objects as rows, and fields as columns. As an example the following query would lists all Java objects that starts with "java.lang" "Select * from "java.lang.*" .
  - ▻ Threads
    This view visualizes information of all Threads at the time of the heap dump. Besides name, needed Heap space, context class loader and state of the Thread its possible to analyze more detailed information by the help of the Inspector View.

  Besides the basic functionality its also possible to create reports automatically. MAT offers a report to create a system overview, a report of the top components or even a report that tries to find leaks automatically which can be a nice feature.

Summary

The inspected tool "Eclipse Memory Analyzer" isn't a tool that belongs to the chapter "Profiler" because its not possible to observe an application. But it can be a huge help to analyze the result of a profiler which creates heap dumps and isn't that great in visualize the dumps. Furthermore there is a domain where this tool provides perfect information which helps a lot by profiling an application. One big problem is to detect the faults of an application that causes out-of-memory errors. This errors are very difficult to solve. But its possible to generate heap and thread dumps by adding the param -XX:+HeapDumpOnOutOfMemoryError at start up to the vm. If the application wants to throw an out-of-memory error the vm creates a heap dump first. By the help of Eclipse MAT this heap dump can be a big help to detect the faults at the source code.

Chapter 8

Conclusion

In the conclusion you will see the summary of this study. It starts with a result-table which shows the details of every aspect of the tested profilers. Refer to the scale definitions in section 5.2 to comprehend the final results of the evaluation. Note that some of the scales are ordinal and some are nominal. The last part of this section is the recommendation on the basis of this result table.

Scaletype		JVMMonitor	NetbeansProfiler	EclipseTPTP	Mission Control
Database		0	0	0	0
Heap		2	3	3	3
Thread Profiling Functionality		4	3	2	3
CPU Profiling Functionality		2	2	2	3
Overhead		3	2	1	4
Additional Functions		1	2	1	2
GUI		2	3	2	3
Development:	Latest Version	version 3.8.1, Feb 2, 2013	version 6.5.1,	version 4.7.2, 25 Feb 2011	version 4.1
	future-proof	yes	yes	no	yes
Licence:	Name	Eclipse Public License v1.0	CDDL and GPL v2	Eclipse Public License v1.0	Eclipse Public License v1.0
Support:	Forum	no	yes	no	yes
	Documentation	yes	yes	yes	yes
	Active support	yes	yes	no	yes
	Active community	yes	yes	no	yes
Basic conditions:	IDE Integration	yes/ Eclipse > 3.6	Netbeans	yes/Eclipse (only AUT < JAVA 1.6)	yes/Eclipse
	OS-compatibility	platform independent	platform independent	platform independent	platform independent
	extensibility	no	no	no	no

Scaletype		VisualVM	JProfiler	YourKit	Memory Analyzer
Database		0	3	0	0
Heap		2	3	3	1
Thread Profiling Functionality		3	3	3	1
CPU Profiling Functionality		3	4	4	0
Overhead		2	3	2	-
Additional Functions		2	4	4	0
GUI		3	4	4	2
Development:	Latest Version	12 Sep, 2013	31 July 2013	30 Oct 2013	version 1.3.0, 26 June 2013
	future-proof	yes	yes	yes	yes
Licence:	Name	GPL v.2+ce	Commercial, 10 days full testversion	Commercial, Testversion	Eclipse Public License v1.0
Support:	Forum	yes	no	yes	yes
	Documentation	yes	yes	yes	yes
	Active support	yes	yes	yes	no
	Active community	yes	yes	no	yes
Basic conditions:	IDE Integration	yes/Eclipse >3.6	yes/Eclipse >3.3	yes/Eclipse to startup	yes/Eclipse (stand-alone also available)
	OS-compatibility	platform independent	platform independent	platform independent	platform independent
	extensibility	yes	no	no	no

After a long period of evaluation three profilers stood out through their convincing capabilities. Thus it made it difficult to make a recommendation for a single profiler. Nowadays the demands of software engineering products are ever-increasing. Software could contain various technologies and principles like databases, interfaces to the underlying system, connecting to the internet, rendering real-time animations, and the usage of several protocols to communicate with peripheral devices. Every software demands various resources from its environment. A software which creates real-time animations, for instance, would require a lot of power of the graphics card. To calculate the rendering algorithms the software would need to interact with a rendering library like OpenGL or DirectX. These requirements require other testing techniques than — for example — a program for tax-calculation would need. A perfect profiler would need to offer the best solution for any imaginable scenario. This is an impossible feat to develop.

We extracted the main functionality of each profiler and analyzed them in their capability and functional diversity. Each functionality obtained a rating by the use of predefined scale definitions. As a result the commercial profilers have achieved higher ratings in overall. The JProfiler and the YourKit profiler are solid in the CPU profiling functionality, its handling, and its multiplicity of qualitative additional functions. If you develop a commercial software which would cost your company a huge amount of money, if crashed, you should consider the pros and the cons before you choose a profiler. One function of the YourKit profiler which was outstanding in comparison to similar functions was the ability to find possible improvements in the code. The profiler searches for memory waste like duplicate strings and arrays, null fields or zero length arrays, possible memory leaks, file accesses, and various other possible improvements. The other profiler which attracts positive attention is the JProfiler. It stood out with a beautiful way to present the details of the AuT. Furthermore this presentation was consistently present in all functionalities of the profiler. Additional functions like viewing a graph of the call tree of threads allows the analyst to find out which thread started another and how long it stood active. The beautiful and multifaceted visualization in every aspect is an additional plus point.
But the open source products don't lack heavily behind the commercial products. A lot of the free profilers come with plugins to integrate into common IDEs and provide full featured functionality as well developed as the commercial tools are.

In conclusion you have to compare the tools with the help of the given result table to find a profiler which fits your needs perfectly.

Appendix A

Detailed Profiler List

A.1 AppDynamics

Name:	AppDynamics
Manufacturer:	AppDynamics
Language:	
GUI:	yes
State / Activity:	active, stable
IDE Integration:	Eclipse / intellij / Oracle JDeveloper / Netbeans Win/Linux/Unix/Mac
Command line capability:	-
Supported OS:	no details
Time Measurement:	yes, Observation of all activity of the application under test
Memory Measurement:	yes,
Thread Measurement:	yes
Data Collection:	no detail
Overhead:	no detail
Special Features:	display the full application, observation of the full system (databases, server, application etc.) automated fail and error detection with AppDynamics. statistics and overviews of application usage
Support:	Support and Community
License:	Commercial Version

A.2 dynaTrace

Name:	dynaTrace
Manufacturer:	Compuware
Language:	Multilingual
GUI:	yes
State / Activity:	active, stable
IDE Integration:	Eclipse plugin / intellij plugin
Command line capability:	yes
Supported OS:	all common - inform at http://de.compuware.com/application-performance-management/platform-support-matrix.html
Time Measurement:	yes, observation of requests, Hot Spots, statistics of methods
Memory Measurement:	yes, very detailed (size of objects, instances, call patches, classes, heap-structure, Garbage Collector)
Thread Measurement:	yes
Data Collection:	no details
Overhead:	"<2% overhead"
Special Features:	locale and remote applications are supported, on-demand profiling,
Support:	Online documentation, Request Support / Bug Report
License:	commercial

A.3 Eclipse Hyades Plugin

Name:	Eclipse Hyades Plugin
Manufacturer:	Eclipse Foundation
Language:	English
GUI:	yes
State / Activity:	inactive, obsolete
IDE Integration:	Eclipse plugin
Command line capability:	no
Supported OS:	All OS supported by Eclipse
Time Measurement:	yes
Memory Measurement:	yes
Thread Measurement:	yes
Data Collection:	no details
Overhead:	no details
Special Features:	no details
Support:	Inactive community
License:	Eclipse Public License v1.0

A.4 Eclipse Profiler Plugin

Name:	Eclipse Profiler Plugin
Manufacturer:	Ricardo Inzaurra
Language:	English
GUI:	yes
State / Activity:	active, stable
IDE Integration:	Eclipse plugin
Command line capability:	no
Supported OS:	All OS supported by Eclipse
Time Measurement:	yes
Memory Measurement:	yes, heap profiling
Thread Measurement:	yes, call tree is shown
Data Collection:	no details
Overhead:	no details
Special Features:	Visualization of call graph
Support:	Get started guide
License:	Common Public License 1.0

A.5 Eclipse TPTP

Name:	Eclipse TPTP
Manufacturer:	Eclipse Foundation
Language:	English
GUI:	yes
State / Activity:	inactive, successor to Eclipse Hyades Plugin (Section A.3)
IDE Integration:	Eclipse plugin
Command line capability:	no
Supported OS:	All OS supported by Eclipse
Time Measurement:	yes
Memory Measurement:	yes
Thread Measurement:	yes
Data Collection:	no details
Overhead:	no details
Special Features:	locale and remote applications are supported
Support:	Online documentation, community
License:	Eclipse Public License v1.0

A.6 InspectIt

Name:	InspectIt
Manufacturer:	Novatec
Language:	Java, Spring, Eclipse RCP, PicoAgent.
GUI:	yes
State / Activity:	active, stable
IDE Integration:	-
Command line capability:	-
Supported OS:	Windows (32/64 bit), Linux (32/64 bit) Mac(partly 32/64bit)
Time Measurement:	yes, Observation of single methods in detail
Memory Measurement:	yes, cursorily
Thread Measurement:	yes, cursorily
Data Collection:	manipulating source code previous to compile process
Overhead:	middle
Special Features:	Observation of database-accesses, exceptions and Http-Requests
Support:	Kick-off presentation of NovaTec, online documentation
License:	License of NovaTec

A.7 Java Mission Control

Name:	Java Mission Control
Manufacturer:	Oracle
Language:	Multilingual
GUI:	yes
State / Activity:	active, stable
IDE Integration:	Eclipse plugin
Command line capability:	yes
Supported OS:	all common - inform at http://de.compuware.com/application-performance-management/platform-support-matrix.html
Time Measurement:	yes, really exact measurements of requests
Memory Measurement:	yes, memory leaks
Thread Measurement:	yes
Data Collection:	directly from the VM
Overhead:	no Overhead
Special Features:	single JVM; maybe the application have to be ported
Support:	Online documentation, community, tutorials
License:	free, closed source (special function have to be payed)

A.8 JMap

Name:	JMap
Manufacturer:	Oracle
Language:	English
GUI:	no
State / Activity:	inactive, finished
IDE Integration:	In JDK integrated
Command line capability:	yes
Supported OS:	In JDK integrated
Time Measurement:	no
Memory Measurement:	yes, Memory Dumps
Thread Measurement:	no
Data Collection:	directly from the VM
Overhead:	no overhead
Special Features:	absolute shell application, only memory dumps
Support:	Online documentation, community, tutorials
License:	open source

A.9 jMechanic

Name:	jMechanic
Manufacturer:	
Language:	java
GUI:	yes, swt
State / Activity:	Development suspended since 2004. Alpha state
IDE Integration:	Eclipse
Command line capability:	no
Supported OS:	java supporting OS
Time Measurement:	yes
Memory Measurement:	yes, Heap- and loaded classes observation
Thread Measurement:	no
Data Collection:	?
Overhead:	less
Special Features:	supports local and remote applications
Support:	sourceforge - But not active
License:	AFL

A.10 JProbe

Name:	JProbe
Manufacturer:	Quest Software
Language:	Java
GUI:	Eclipse Plug-In
State / Activity:	Production/Stable
IDE Integration:	Eclipse 3.4, 3.5 and 3.6
Command line capability:	-
Supported OS:	Systems that support Eclipse
Time Measurement:	yes, memory observation in real time, CPU observation
Memory Measurement:	yes, Garbage Collector-, Heap-, loaded classes observation
Thread Measurement:	yes
Data Collection:	-
Overhead:	less
Special Features:	only application on local systems, well designed GUI with many settings, record snapshots at particular time, many features
Support:	JProbe Documentation, JProbe Users Guide, JProbe Help https://support.quest.com/Default.aspx
License:	Commercial

A.11 JProfiler

Name:	JProfiler
Manufacturer:	ej-technologies
Language:	
GUI:	yes
State / Activity:	active, stable
IDE Integration:	Eclipse / intellij / Oracle JDeveloper / Netbeans Win/Linux/Unix/Mac
Command line capability:	-
Supported OS:	all common - inform at http://www.ej-technologies.com/products/jprofiler/featuresPlatforms.html
Time Measurement:	yes, observation of requests, Hot Spots, statistics of methods
Memory Measurement:	yes, very detailed (size of objects, instances, call paths, classes, heap-structure, Garbage Collector)
Thread Measurement:	yes
Data Collection:	no details
Overhead:	no details
Special Features:	locale and remote applications are supported, supports comparison of actual and previous executions, tracking of requests (AWT/Swing /Thread Requests) . It is possible to install on server like Apache/JBoss/Jetty/IBM to inspect EE applications
Support:	Online documentation, Request Support / Bug Report
License:	Comercial Version

A.12 JVM Monitor

Name:	JVM Monitor
Manufacturer:	http://www.jvmmonitor.org/
Language:	Java
GUI:	Yes, embedded into Eclipse. Also SWT GUI
State / Activity:	Active development (Release 3.8.1)
IDE Integration:	Eclipse PlugIn for: Helios 3.6.x, Indigo 3.7.x or Juno 3.8.x/4.2.x Java for Eclipse: Oracle JDK 6 or 7, OpenJDK 6 or 7, or Apple JDK 6
Command line capability:	No Statements
Supported OS:	Windows, Linux, or Mac OS X
Time Measurement:	Shows time depending diagrams (configurable) of main memory, number temporary loaded objects, number of threads, CPU usage
Memory Measurement:	Detailed illustration of memory usage
Thread Measurement:	Detailed illustration of running threads with CPU usage and CPU state
Data Collection:	-
Overhead:	little
Special Features:	automatic detection of JVM in localhost or via remote connection by stating of host and port
Support:	Issue Tracker to report bugs or make suggestions for new requirements
License:	Free open source software, Eclipse Public License v1.0

A.13 jvmstat

Name:	jvmstat
Manufacturer:	ORACLE
Language:	Java
GUI:	yes, Swing
State / Activity:	development suspended
IDE Integration:	stand alone tool
Command line capability:	yes
Supported OS:	Windows 98 and Windows ME
Time Measurement:	yes - compiling time/ Class Loader/ histogram of previous runtimes
Memory Measurement:	yes
Thread Measurement:	yes
Data Collection:	-
Overhead:	less
Special Features:	remote access
Support:	no support
License:	Open Source Software

A.14 JRat

Name:	JRat
Manufacturer:	
Language:	java
GUI:	yes, swing
State / Activity:	not activity since 2007 - never reached stable version
IDE Integration:	no
Command line capability:	no
Supported OS:	OS able to run Java
Time Measurement:	yes, single methods are observable
Memory Measurement:	no
Thread Measurement:	no
Data Collection:	JVMPI
Overhead:	less
Special Features:	-
Support:	sourceforge : Wiki / Mailing List / BugTracker
License:	GNU Library or Lesser General Public License version 2.0 (LGPLv2)

A.15 Memory Analyzer

Name:	Memory Analyzer
Manufacturer:	Eclipse Org
Language:	English
GUI:	yes
State / Activity:	active, stable
IDE Integration:	Eclipse plugin
Command line capability:	no
Supported OS:	all common which supports Eclipse
Time Measurement:	no
Memory Measurement:	yes, memory leaks
Thread Measurement:	no
Data Collection:	directly from the VM
Overhead:	less overhead
Special Features:	-
Support:	Online documentation, Community, Tutorials
License:	open source

A.16 NetBeans Profiler

Name:	NetBeans Profiler
Manufacturer:	NetBeans
Language:	-
GUI:	Embedded into Netbeans
State / Activity:	active, stable
IDE Integration:	Netbeans
Command line capability:	-
Supported OS:	Windows, Linux(x86/x64), Solaris(x86/x64), Solaris(sparc), Mac OS, OS independent Zip
Time Measurement:	yes, CPU-Activity at real time, Threads and storage usage at real time
Memory Measurement:	yes, Garbage Collector-, Heap-, loaded classes observation
Thread Measurement:	yes
Data Collection:	-
Overhead:	less
Special Features:	Integrated into NetBeans, connection to remote applications
Support:	Profiler Blog News, features, tips and tricks and all around the profiler.
License:	Common Development and Distribution License CDDL v1.0 and GNU General Public License GPL v2.

A.17 Thermostat

Name:	Thermostat
Manufacturer:	Redhead
Language:	Java
GUI:	yes, Swing
State / Activity:	development state, last stable 0.92
IDE Integration:	no, will be published as embedded eclipse plugin soon
Command line capability:	yes
Supported OS:	only Fedora at the moment
Time Measurement:	yes, per host : record loaded classes
Memory Measurement:	yes, Able to observe Garbage Collector, memory allocation of single instances, heap etc.
Thread Measurement:	yes
Data Collection:	jstatd, JMX, JVMTI, and Systemtap
Overhead:	less
Special Features:	provides monitoring of several Java Virtual Machines on several hosts
Support:	Bug Tracker, wiki, mailinglist
License:	GPLv2+

A.18 VisualVM

Name:	VisualVM
Manufacturer:	Oracle/Sun
Language:	java
GUI:	yes, swing
State / Activity:	active, stable
IDE Integration:	Eclipse / Netbeans
Command line capability:	no
Supported OS:	OS able to run Java
Time Measurement:	yes, able to observe single methods
Memory Measurement:	yes, Garbage Collector-, Heap-, loaded classes observation
Thread Measurement:	yes
Data Collection:	jvmstat, JMX, the Serviceability Agent (SA), and the attached API
Overhead:	less
Special Features:	already integrated into java jdk, locale and remote applications are supported, plugins may be installed
Support:	JIRA BugTracker
License:	GPLv2 + CE

A.19 YourKit Java Profiler

Name:	YourKit Java Profiler
Manufacturer:	YourKit, LLC
Language:	English
GUI:	yes
State / Activity:	active, stable
IDE Integration:	Eclipse / intellij / Oracle JDeveloper / Netbeans Win/Linux/Unix/Mac/Solaris/etc.
Command line capability:	yes
Supported OS:	all common - inform at http://www.yourkit.com/features/index.jsp#multiplatform
Time Measurement:	yes, observation of requests, Hot Spots, statistics of methods
Memory Measurement:	yes, very detailed (size of objects, instances, call paths, classes, heap-structure, Garbage Collector)
Thread Measurement:	yes
Data Collection:	no details
Overhead:	"zero overhead"
Special Features:	locale and remote applications are supported, on-demand profiling,
Support:	Online documentation, Request Support / Bug Report
License:	commercial

Glossary

Application under Test The application which is tested by the current profiler. In this study this is either Columba or Proguard. 87

Bytecode Instrumentation A technique used to gain data on execution time and duration during profiling of a certain method. Explained in detail in Section 2.4. 7, 87

Java Management Extensions JMX provides tools for building distributed, modular, and dynamic solutions for managing and monitoring applications. Mostly used for monitoring. Available starting Java 1.5 JMX. 9

Java Virtual Machine The machine responsible for executing Java applications and Java bytecode. 87

Java Virtual Machine Tooling Interface A programming interface used by development, profiling, and monitoring tools. Provides functions to inspect the state and control the execution of applications running in the JVM. Explained in detail in Section 2.2. 7–9, 87

jvmstat A small tool without GUI whose only purpose is to dump memory heaps. Is often used by other profilers to get a memory heap dump. 13

Method Sampling A technique used to gain data on execution time and duration during profiling of certain methods. Explained in detail in Section 2.3. 7, 8

Acronyms

AuT Application under Test. 8, 9, 58, 64

BCI Bytecode Instrumentation. 8, 9, 26

IDE Integrated Development Environment. 35

JDK Java Development Kit. 54

JMX Java Management Extensions. 7, 9, 13

JVM Java Virtual Machine. 3, 8, 9, 25, 26, 54, 58, 85

JVM TI Java Virtual Machine Tooling Interface. 6–8

MAT Eclipse Memory Analyzing Tool. 14, 59, 60

SSL Secure Sockets Layer. 9

TPTP Eclipse Test & Performance Tools Platform. 13, 14, 32, 33, 35

Bibliography

[Col] Columba. `http://sourceforge.net/projects/columba/`.

[JMC] Java Mission Control 5.2 release notes. URL `http://www.oracle.com/technetwork/java/javase/2col/jmc-relnotes-2004763.html`.

[JMX] Java management extensions (jmx) technology. URL `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`.

[JVM] Jvm tooling interface 1.2 documentation. URL `http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html`.

[Pro] Proguard. `http://proguard.sourceforge.net/`.

[JVM 2004] Creating a debugging and profiling agent with jvmti, 2004. URL `http://www.oracle.com/technetwork/articles/javase/jvmti-136367.html`.

[JPM 2011] Java performance messen – mit sampling oder instrumentierung, Oct. 2011. URL `https://blog.codecentric.de/2011/10/java-performance-messen-mit-sampling-oder-instrumentierung/`.

[Jiri July 28 ,2008] S. Jiri. Netbeans profiler
the netbeans profiler and visualvm blog
profiling with visualvm, part 1. `https://blogs.oracle.com/nbprofiler/entry/profiling_with_visualvm_part_2`, July 28 ,2008.

[Siegl May 17, 2013] S. Siegl. Inspectit overview. `https://documentation.novatec-gmbh.de/display/INSPECTIT/Overview`, May 17, 2013.

[YourKit GmbH] YourKit GmbH. Yourkit profiler. `http://www.yourkit.com/`.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

Stuttgart, den 04.12.2013

Declaration

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.

Signature:

Stuttgart, 4th December 2013