TECHNICAL UNIVERSITY OF KOŠICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Visualization of Garbage Collection Algorithms in JRE

Master's Thesis

2012

Bc. Martin Škurla

TECHNICAL UNIVERSITY OF KOŠICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Visualization of Garbage Collection Algorithms in JRE

Master's Thesis

Study Programme:	Informatics
Field of study:	9.2.1 Informatics
Department:	Department of Computers and Informatics (KPI)
Supervisor:	doc. Ing. Ladislav Samuelis, CSc.
Consultant:	doc. Ing. Ladislav Samuelis, CSc.

Košice 2012 **Bc. Martin Škurla**

Abstract

The vast majority of the currently developed software systems is not implemented using low-level programming languages, but rather using higher-level programming languages that are executed in virtual machines. Garbage Collection (GC) is the process dealing with automatic memory management including the cleanup of no longer used objects. GC turns out to be one of the most important research areas around virtual machines over the last decades. There are many metrics that could measure the efficiency of the particular GC algorithm, but one of the most natural one is the visualization of the GC process. The main goal of this thesis is to analyze, design and implement a GC visualization tool built on top of the original GCSpy project foundations.

Keywords

Garbage Collection, Java Virtual Machine, Memory management, Visualization

Abstrakt

Väčšina súčasne vyvíjaných softvérových systémov nie je implementovaná použitím nízkoúrovňových programovaních jazykov, ale použitím programovacích jazykov vyššej úrovne, ktoré sú vykonávané vo virtuálnych strojoch. "Garbage Collection" (GC) je proces zaoberajúci sa automatickou správou pamäte v rátane odstraňovania nepoužívaných objektov. GC sa stalo jednou z najvýznamnejších oblastí výskumu v rámci virtuálnych strojov v posledných desaťročiach. Existuje mnoho metrík, ktoré hodnotia efektivitu konkrétnych GC algoritmov, ale jedna z najprirodzenejších metrík je vizualizácia procesu GC. Hlavným cieľom tejto diplomovej práce je analýza, návrh a implementácia vizualizačného nástroja postaveného na základoch pôvodného projektu GCSpy.

Kľúčové slová

Java virtuálny stroj, Manažment pamäte, Odstraňovanie nepoužívaných objektov z pamäte, Vizualizácia

TECHNICAL UNIVERSITY OF KOŠICE

Faculty of Electrical Engineering and Informatics

Department of Computers and Informatics

DIPLOMA THESIS ASSIGNMENT

Field of study:	**9.2.1 Informatics**
Study programme:	**Informatics**

Thesis title:

Visualization of Garbage Collection Algorithms in JRE
Vizualizácia algoritmov pre odstraňovanie nepoužívaných objektov v JRE

Student (titles, given name, family name): **Bc. Martin Škurla**

Supervisor (titles, given name, family name):	**doc. Ing. Ladislav Samuelis, CSc.**
Supervising department:	**Department of Computers and Informatics**
Consultant (titles, given name, family name):	**doc. Ing. Ladislav Samuelis, CSc.**
Consultant`s affiliation:	**Department of Computers and Informatics**

Thesis preparation instructions:

1. **Study the garbage collection algorithms**
2. **To gain skills in GCSpy environment**
3. **To specify and design enhancements for the GCSpy environment**
4. **To implement the enhancements**
5. **To propose work for further development**
6. **Writing of documentation according to standards of the department.**

Language of the thesis:	English
Thesis submission deadline:	27.04.2012
Assigned on:	31.10.2011

[illegible]

prof. Ing. Ján Kollár, CSc.
Head of the Department

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY

[illegible]

prof. Ing. Liberios Vokorokos, PhD.
Dean of the Faculty

Declaration

I hereby declare that this my own work and effort. Where other sources of information have been used, they have been acknowledged.

Košice, 10.5. 2012 ....

Signature

Acknowledgement

I would like to express my sincere thanks to my supervisor doc. Ing. Ladislav Samuelis CSc., the main Supervisor, for comments and expert opinions during the creation of this thesis.

I would like to say thanks to my family for patience and constant support during my study.

To all other who gave a hand, I say thank you very much.

Preface

We are recently witnesses of continuous performance improvements both in the hardware and software worlds. However, for current software systems with millions of requests and users, performance is still a very important aspect. One of not so obvious performance optimization techniques is garbage collection algorithm selection and tuning. Garbage collectors save us a lot of time (programmers do not need to manually deallocate unreferenced objects) and shield us from low-level memory errors including dangling pointers, double memory frees, premature memory frees and memory leaks.

To be able to take full advantage of garbage collection, we need to make the garbage collection exploration process easy. Visualizing the garbage collection includes visualizing events, memory abstractions and historic memory consumption observation. All previously mentioned aspects are critical parts for the performance tuning, lowering the memory allocation frequency and memory allocation pauses caused by automatic memory management.

Garbage collection visualization is not only important for garbage collection researchers and implementers, but also for anybody who would like to understand the memory management characteristics of one's application better.

The goal of this thesis is to introduce garbage collection in a nutshell, introduce the traditional GCSpy project together with its main goals and characteristics, look at the main disadvantages of the GCSpy project, suggest improvements, implement the proposed changes and provide a working prototype.

Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

API	**Application Programming Interface**
CL	**Command Line**
CMS	**Concurrent Mark-Sweep Garbage Collector**
CORBA	**Common Object Request Broker Architecture**
CPU	**Central Processing Unit**
C4	**Continuously Concurrent Compacting Collector**
DLL	**Dynamic Link Library**
DSL	**Domain Specific Language**
EAR	**Enterprise Archive**
GC	**Garbage Collector / Garbage Collection**
GCC	**GNU Compiler Collection**
GCSpy	**Garbage Collection Spy**
GUI	**Graphical User Interface**
G1	**Garbage-First Garbage Collector**
IDE	**Integrated Development Environment**
JAR	**Java Archive**
JDK	**Java Development Kit**
JNA	**Java Native Access**
JNI	**Java Native Interface**
JRE	**Java Runtime Environment**
JLS	**Java Language Specification**
JPDA	**Java Platform Debugger Architecture**
JSR	**Java Specification Request**
JVM	**Java Virtual Machine**
JVMS	**Java Virtual Machine Specification**
JVMDI	**Java Virtual Machine Debug Interface**
JVMPI	**Java Virtual Machine Profiler Interface**
JVMTI	**Java Virtual Machine Tool Interface**

LAF	**Look and Feel**
MOJO	**Maven plain Old Java Object**
MSVC	**Microsoft Visual C++**
NAR	**Native Archive**
OS	**Operating System**
OSGi	**Open Services Gateway initiative**
POM	**Project Object Model**
RCP	**Rich Client Platform**
RMI	**Remote Method Invocation**
SA	**Serviceability Agent**
SCM	**Source Code Management**
SoC	**Separation of Concerns**
SPI	**Service Provider Interface**
STW	**Stop-The-World Garbage Collector**
TCP/IP	**Transmission Control Protocol / Internet Protocol**
VM	**Virtual Machine**
WAR	**Web Archive**
XML	**eXtensible Markup Language**
YAML	**YAML Ain't Markup Language**

List of Terms

Bytecode instrumentation is a technique dealing with the manipulation of bytecode. Manipulation typically means insertion or any other kind of changing the bytecode during compilation or runtime execution.

Classpath is a list of directories where the java compiler will try to find java source files and the java interpreter will try to find java class files or JAR files. Classpath can be set either as CLASSPATH environment variable or on command line. In a standard application the System ClassLoader will load classes from classpath.

JavaBean is a naming convention for reusable software components written in the Java programming language. Simplified requirements force every JavaBean object to implement `java.io.Serializable` interface, to have a publicly accessible constructor with zero parameters and to have getter and setter methods for accessing object properties.

Java Bytecode is an intermediate language that could be executed inside a JVM. Java bytecode is typically compiled from the Java programming language source files, but there are plenty of other languages that produce a Java bytecode including not only, but also Clojure, Groovy, JRuby, Jython and Scala. Java bytecode instructions are defined in the Java Virtual Machine Specification.

Java Virtual Machine is a virtual machine that allows to execute Java bytecode. Java virtual machine is distributed together with core Java APIs and optionally also development tools.

MOJO is an abbreviation for Maven plain old Java Object. Every MOJO represents a particular plugin's goal and has to implement `org.apache.maven.plugin.Mojo` interface. MOJO can be written in Java or other scripting languages. A packaged set of MOJOs creates a Maven plugin.

Virtual Machine is either a software emulation or a hardware virtualization completely isolated from host operating system. One of the fundamental characteristics of virtual machines is the fact that the software running inside a virtual machine is limited to the resources and abstractions provided by the running virtual machine.

Introduction

Garbage collection turns out to be one of the most important research areas around virtual machines over the last decades. Providing a visual way to measure the particular garbage collection algorithm can be beneficial for virtual machine and garbage collector implementers. In the case when the visualization will be easy to incorporate and use, it can be useful for everyday developers as well. The visualization of garbage collection should be as non-intrusive for the observed system as possible.

The aim of this thesis is to analyze, design and implement a garbage collection visualization tool built on top of the original GCSpy project foundations.

The first chapter describes the problem expression.

The second chapter summarizes analytical considerations. The analytical considerations describe the main technologies, used concepts and necessary theoretical background used across the thesis. The second chapter analysis Maven as not only build tool, Java Virtual Machine Tool Interface as low-level API for gathering virtual machine specific data and NetBeans Rich Client Platform, the only OSGi and Swing based modular desktop application framework. Second chapter also describes garbage collection terms, concepts, metrics, classification and concrete garbage collection algorithms that are part of the standard JRE. Last but not least, original GCSpy project abstractions and architecture description is also part of this chapter.

The third chapter addresses design goals and high level overview.

The fourth chapter deals with implementation details, problems and solutions of chosen topics. The content of the fourth chapter includes the description of initial considerations, design decisions and parts describing how it works for both the build and runtime infrastructure. The majority of the test infrastructure part is the description of dealing with garbage collection indeterminism. The content of the socket communication protocol chapter deals with sent data semantics, connection management, thread model and blocking states. Every kind of portability including operating system, virtual machine and java portability is described in details. At the end of the fourth chapter the summarization of deployment and the comparison with the original GCSpy project is described as well.

The fifth chapter summarizes the conclusion.

1 The problem expression

The aim of this thesis is to analyze, design and implement a garbage collection visualization tool built on top of the original GCSpy project foundations.

Before the actual creation of the visualization tool, it is necessary to analyze the garbage collection principles, concepts and concrete garbage collection algorithms. It is also necessary to study the architecture and characteristics of the original GCSpy project to be able to propose changes. Based on the lessons learned, it is required to design and implement the visualization tool. The documentation is required part of the solution.

2 Analytical considerations

2.1 Maven

Maven is not only a build tool. According to the book *Better Builds with Maven* [1], Maven is a project management framework, a combination of ideas, standards and software. There is also another interesting explanation of what Maven is in that book:

"Maven is a declarative project management tool that decreases your overall time to market by effectively leveraging cross-project intelligence. It simultaneously reduces your duplication effort and leads to higher code quality."

Maven simplifies the process of managing a software project by simplifying the build, documentation, distribution and the deployment process. Maven is an Apache Software Foundation open-source project [2].

Maven defines a set of build standards, the notion of local and remote artifact repository, dependency management, multi-module project structure, standard lifecycle for building, testing and deploying Java software artifacts.

In Maven, the build lifecycle consists of a set of phases that will be executed in defined order. Each phase can execute one or multiple executable actions related to that phase. These actions are called goals. Goals that are at lower level in the build lifecycle hierarchy will be automatically executed behind the scene.

Maven defines the content of Project Object Model (POM), which is a text file that declaratively describes the project.

Maven was inspired by Brad Appleton's pattern definition published in his article *Patterns and Software: Essential Concepts and Terminology* [3]:

"Patterns help create a shared language for communicating insight and experience about problems and their solutions."

As a result, the following Maven core principles were defined:

- convention over configuration,
- reuse of build logic,
- declarative execution,
- coherent organization of dependencies.

2.1.1 Convention over configuration

One of the fundamental steps where Maven will remove a lot of complexity is by providing a sensible set of default values as part of the configuration. You are still flexible and able to change any of those default configuration values, but most of the time it saves enormous amount of time by just respecting Maven defaults.

The idea of "Convention over configuration" was largely popularized by Ruby on Rails web framework [4].

Maven is incorporating convention over configuration by defining:

- standard directory layout for projects,
- one primary output per project,
- standard naming conventions.

Standard directory layout for projects includes well-defined directory structure for project source files, resource files, resource filters, configuration files and any kind of output including compiled classes, packaged artifacts, documentation, tests results and integration tests results.

One primary output per project is supported by the SoC[1] principle. The code base should be divided into multiple modules with well-defined dependency management and thus respect modularity. In Maven, every project including multi-module build, will produce only one output artifact.

Standard naming conventions include naming conventions for directories and build artifacts. All build artifacts will be named according to the following scheme: `artifactId-version.extension` and will be located relative to the Maven local repository inside the directory: `groupId/artifactId/version`.

[1] Separation of Concerns

2.1.2 Reuse of build logic

Maven architecture is built on top of very simple, but also very extensible core. The vast majority of Maven's functionality is provided by Maven build plugins. You can find a Maven plugin for pretty much anything you will need to do as part of the build process, but you can also develop your own plugin. Maven build plugins are the example of the reuse of build logic.

2.1.3 Declarative execution

Every kind of Maven configuration is done declaratively as part of its POM[2]. Traditionally the only supported language to write the POM in was XML[3]. Starting from Maven-3 as part of the polyglot Maven initiative, it is possible to write the POM using multiple DSLs[4] in languages other than XML. Current support includes Clojure, Groovy, JRuby, Scala and YAML[5].

2.1.4 Coherent organization of dependencies

Now let us talk about the Maven dependencies. Maven dependency is uniquely identified by the trio:

- `groupId`,
- `artifactId`,
- `version`.

The `groupId` uniquely identifies the company, organization or development group that created the artifact.

The `artifactId` uniquely identifies the artifact among all artifacts from the same `groupId`. For modular applications it is common to create one artifact per module.

The `version` represents the version of the artifact. Version could be represented as version range, could contain classifier and could represent a snapshot version.

Maven will automatically download all required dependencies (and their transitive dependencies) from remote Maven Central repository and all specified remote

[2] Project Object Model
[3] eXtensible Markup Language
[4] Domain Specific Language
[5] YAML Ain't Markup Language

repositories into Maven local repository as part of the build process. Maven will also correctly place the required dependencies on compile and test classpath according to their `scope`.

The example of dependency declaration to GCSpy[6] `"utilities"` module will look as follows:

```
<dependency>
    <groupId>org.crazyjavahacking.gcspy</groupId>
    <artifactId>
        org-crazyjavahacking-gcspy-utilities
    </artifactId>
    <version>1.0</version>
    <scope>compile</scope>
</dependency>
```

Fig. 1 Declaring Maven dependency

All projects declaring the same Maven artifact dependency will share exactly the same copy of the artifact from Maven local repository. This effectively reduces the overall disk space consumption by eliminating the need to copy the library JAR[7] files for every single project.

2.1.5 Maven default build lifecycle

There are three Maven build-in build lifecycles:

- default,
- clean,
- site.

The default lifecycle handles the project build, test and deployment.

The clean lifecycle handles the project cleaning.

The site lifecycle handles the generation of project documentation.

[6] Garbage Collection Spy
[7] Java Archive

According to the article *Introduction to the Build Lifecycle* [5], the default Maven build lifecycle contains 23 phases. Not all of phases are connected with goals, but some of them were created as a placeholder for additional plugins because of flexibility.

The following table summarizes all Maven default build lifecycle phases together with their descriptions:

Tab. 1 Maven default build lifecycle

Lifecycle phase	**Phase description**
validate	validates if the project is correct and all necessary information is available
initialize	initializes build state (e.g. set properties or create directories)
generate-sources	generates source code for inclusion in the compilation
process-sources	processes the source code (e.g. filter values)
generate-resources	generates resources for inclusion in the package
process-resources	processes and copies the resources into the destination directory
compile	compiles the source code
process-classes	does the post-processing of files generated from compilation (e.g. bytecode enhancement)
generate-test-sources	generates test source code for inclusion in the compilation
process-test-sources	processes the test source code (e.g. filter values)
generate-test-resources	creates resources for testing
process-test-resources	processes and copies the resources into the test destination directory
test-compile	compiles the test source code into the test destination directory
process-test-classes	does the post-processing of files generated from test compilation (e.g. bytecode enhancement)

test	runs tests using a suitable unit testing framework
prepare-package	performs any operations necessary to prepare a package before the actual packaging
package	takes the compiled code and packages it in its distribution format (e.g. JAR)
pre-integration-test	performs actions required before integration tests are executed (e.g. setting up the required environment)
integration-test	processes and deploys the package (if necessary) into an environment where integration tests will run
post-integration-test	performs actions required after integration tests have been executed (e.g. cleaning up the environment)
verify	runs any checks to verify the package is valid and meets quality criteria
install	installs the package into the local repository, to be able to use it as a dependency in other projects
deploy	copies the final package to the remote repository for sharing with other developers and projects

2.2 Java Virtual Machine Tool Interface

Java Virtual Machine Tool Interface [6][7] is a C/C++ interface for accessing low-level JVM[8] functionalities. JVMTI[9] was introduced in Java SE 5 as part of the JSR[10]-163: Platform Profiling Architecture. JVMTI is the lowest level of the JPDA[11] [8].

JVMTI allows you to inspect the state of the JVM and control the JVM execution. JVMTI is mainly used by debuggers and profilers and is a replacement for previously used JVMDI[12] and JVMPI[13] APIs[14] [9]. JVMTI is built on top of the JNI[15] and JNI types are naturally used across the JVMTI source base.

JVMTI provides unique functionalities that are not part of the JDK[16] such as:

- the ability to do the bytecode instrumentation before any Java class will be loaded into the JVM,
- the ability to follow all direct and indirect references of given object,
- the ability to force the garbage collection,
- the ability to get the line number and local variable tables,
- the ability to iterate over all, reachable or unreachable objects in the heap memory,
- the ability to set and clear breakpoints,
- the ability to suspend and resume any of the currently running Java threads.

In order to execute the JVMTI code, the C/C++ code using JVMTI API has to be compiled and linked into a shared library. The shared library will then be loaded and linked together with the JVM at runtime by specifying one of the two possible java command line arguments:

- `-agentlib:<agentName>=<agentOptions>`,
- `-agentpath:<agentPath>=<agentOptions>`.

[8] Java Virtual Machine
[9] Java Virtual Machine Tool Interface
[10] Java Specification Request
[11] Java Platform Debugger Architecture
[12] Java Virtual Machine Debug Interface
[13] Java Virtual Machine Profiler Interface
[14] Application Programming Interface
[15] Java Native Interface
[16] Java Development Kit

The JVMTI programming style includes two functionalities. You are able to directly call the JVMTI functions and be notified about events.

2.2.1 JVMTI phases

JVMTI defines the following order of JVM execution phases:

- `ONLOAD`,
- `PRIMORDIAL`,
- `START`,
- `LIVE`,
- `DEAD`.

The `ONLOAD` phase represents the state of the JVM during the execution of the `Agent_OnLoad` function. During this phase no bytecode has been executed, no classes have been loaded and no objects have been created yet.

The `PRIMORDIAL` phase represents the state of the JVM between returning from the `Agent_OnLoad` function and entering the `VMStart` event callback function.

The `START` phase represents the state of the JVM between returning from the `VMStart` and entering the `VMInit` event callback function.

The `LIVE` phase represents the state of the JVM between returning from the `VMInit` and entering the `VMDead` event callback function.

The `DEAD` phase represents the state of the JVM after returning from the `VMDead` event callback function.

JVMTI agent library can be started in either the `ONLOAD` or `LIVE` phase and appropriate JVMTI functions must be exported by the shared library. The exact mechanism of attaching the agent during the `LIVE` phase is not a part of the JVMTI specification and thus is implementation and JVM specific.

The important note about the JVMTI is the fact that some functions can be called:

- only during `ONLOAD` phase,
- only during `LIVE` phase,
- only during `START` or `LIVE` phase,
- during all phases.

2.2.2 JVMTI capabilities

JVMTI provides the following comprehensive set of self-describing events:

- accessing local variable,
- forcing early method return,
- generating breakpoint, class load hook, compiled method load, exception, field access, frame pop, garbage collection, method entry, method exit, modification, monitor, native method bind, object free, resource exhaustion heap, resource exhaustion threads, single step and vm object alloc events,
- getting bytecode, constant pool, current contended monitor, current thread CPU[17] time, line numbers, monitor info, owned monitor stack depth info, source debug extension, source file name, synthetic attribute and thread CPU time,
- maintaining original method order,
- popping frame,
- redefining class,
- retransforming class,
- setting native method prefix,
- signaling thread,
- suspending and resuming thread,
- tagging object.

2.2.3 JVMTI programming style

The general JVMTI programming style consists of steps in the following order:

1. registering capabilities,
2. registering event callbacks,
3. turning on event notifications.

First you need to register capabilities. Because of performance reasons, you should always register just the capabilities you will really use. A subset of the JVMTI API is

[17] Central Processing Unit

event-based and will execute event callbacks you have previously registered. At the end you need to turn on event notification of events you would like to be notified about.

2.2.4 The tricky parts of JVMTI

Programming JVM agents could be tricky because there are many subtle details that you have to respect and know about, such as:

- JVM native agent will be linked together with the JVM at runtime and will not run in the managed environment. Due to the linkage, errors in C code (such as `NULL` pointer dereference) will trigger the collapse of the whole JVM without throwing an exception.
- Because the native agent code will not run in the managed environment as every other Java code will, fatal errors (e.g. `NULL` pointer dereference) will not throw an exception, but will crash the whole JVM.
- All memory fragments passed by reference to the JVMTI functions have to be allocated using the JVMTI allocation functions. If you have a code that uses the standard C `malloc()/calloc()` memory allocation functions, you need to allocate a memory space (with the same size) using JVMTI allocation functions and copy the original space content.
- Some JVMTI functions can be called only within the execution of some JVMTI phases.
- All JNI functions, except the JNI Invocation API, can only be used during the `START` or `LIVE` phase.
- Heap callback functions must respect a few restrictions. Those functions cannot directly use other JVMTI or JNI functions except callback-safe functions.

2.3 NetBeans Rich Client Platform

The NetBeans Platform is an open-source generic framework for creating modular loosely-coupled desktop Swing applications.

NetBeans Platform will provide you a lot of functionality [10] that you can directly use without spending time to develop your own solution. One of the most difficult pieces of software from the architecture point of view these days are IDEs[18].

What could be a better proof of NetBeans RCP's flexibility and overall design than the fact that NetBeans IDE is built on top of NetBeans RCP? In fact, the NetBeans RCP[19] was extracted from NetBeans IDE after the first few releases and you can see the conceptual structure in Fig. 2 Conceptual structure of the NetBeans IDE.

Before any further diving into NetBeans RCP, let us first describe what is the rich client. The definition of "Rich Client" according to the book *The Definite Guide to NetBeans Platform* [11] is as follows:

"In a client server architecture the term “rich client” is used for clients where the data processing occurs mainly on the client side. The client also provides the graphical user interface. Often rich clients are applications that are extendable via plugins and modules. In this way, rich clients are able to solve more than one problem. Rich clients can also potentially solve related problems, as well as those that are completely foreign to their original purpose."

The main characteristics of a rich client are:

- flexible and modular application architecture,
- platform independence,
- adaptability to the end user,
- ability to work online as well as offline,
- simplified distribution to the end user,
- simplified updating of the client.

As you can see, the rich client is naturally tight to the modular application design.

[18] Integrated Development Environment
[19] Rich Client Platform

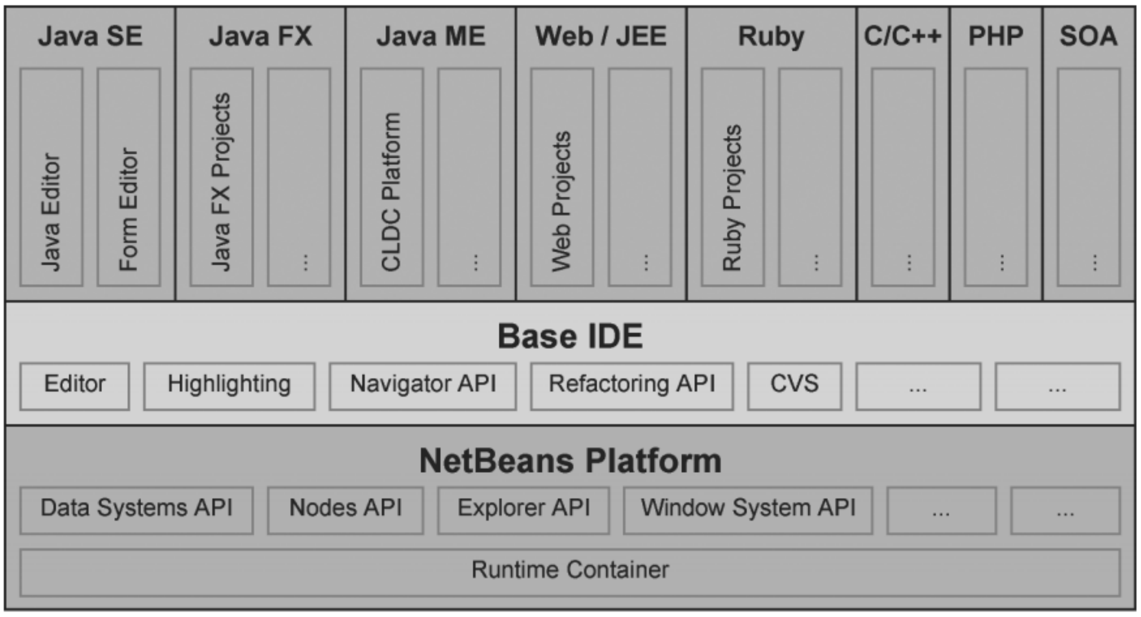


Fig. 2 Conceptual structure of the NetBeans IDE

2.3.1 Main characteristics

NetBeans Platform consists of multiple layers and a set of APIs and SPIs[20] on those levels. For more details see Fig. 3 NetBeans Platform architecture.

According to DZone Refcard *Getting Started with NetBeans Platform 7.0* [12] the following are the main features of the NetBeans RCP:

- Module System,
- Lifecycle Management,
- Pluggability,
- Service Infrastructure,
- File System,
- Window System,
- Standardized UI Toolkit,
- Generic Presentation Layer,
- Advanced Swing Components,
- JavaHelp Integration.

[20] Service Provider Interface

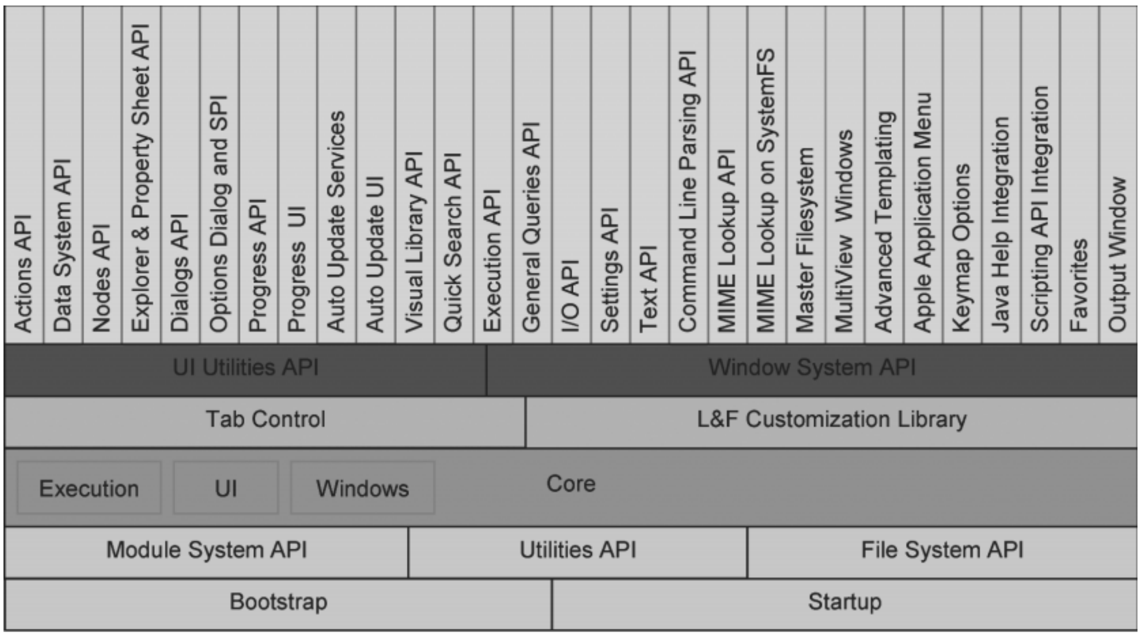


Fig. 3 NetBeans Platform architecture

You can use either standard NetBeans Platform modules or OSGi[21] bundles and NetBeans will handle the versioning support. The NetBeans runtime container provides lifecycle services and allows you to install/uninstall and activate/deactivate modules. NetBeans Platform provides an infrastructure for registering and retrieving service implementations and provides stream-oriented access to flat and hierarchical structures. NetBeans window system allows you to maximize/minimize, dock/undock and drag-and-drop windows. NetBeans APIs provide a generic model for presenting data. Because of Swing, portability and ability to change the LAF[22] is guaranteed.

Another set of characteristics of the NetBeans RCP is summarized in the [11]:

- User interface framework,
- Data editor,
- Customization display,
- Wizard framework,
- Data systems,
- Internationalization,
- Help system.

[21] Open Services Gateway initiative
[22] Look and Feel

2.4 Garbage collection

I cannot imagine a better start of the garbage collection chapter as by quoting the idea from Joshua Bloch, the Chief Java Architect at Google, which he told as part of his talk *Java: The good, the Bad and the Ugly parts* during Devoxx 2011 conference:

"Garbage collection isn't magic, but is pretty close."

Garbage collection [13] is a form of automatic memory management and was invented by John McCarthy in 1958 as part of the Lisp programming language. The first description of garbage collection according to the article *Recursive Functions of Symbolic Expressions and Their Computation by Machine* [14] is as follows:

"Nothing happens until the program runs out of free storage. When a free register is wanted, and there is none left on the free-storage list, a reclamation cycle starts.

First, the program finds all registers accessible from the base registers and makes their signs negative. This is accomplished by starting from each of the base registers and changing the sign of every register that can be reached from it by a car - cdr chain. If the program encounters a register in this process which already has a negative sign, it assumes that this register has already been reached.

After all of the accessible registers have had their signs changed, the program goes through the area of memory reserved for the storage of list structures and puts all the registers whose signs were not changed in the previous step back on the free-storage list, and makes the signs of the accessible registers positive again."

Garbage collectors save us a lot of time (programmers do not need to manually deallocate unreferenced objects) and shield us from low-level memory errors including dangling pointers, double memory frees, premature memory frees and memory leaks.

Garbage collection turns out to be one of the most important research areas around virtual machines over the last decades, even if the beginnings were not easy [13] :

"The first online demonstration of garbage collection was to an MIT Industrial Liaison Symposium. Unfortunately, mid-way through the demonstration, the IBM 704 exhausted (all of!) its 32k words of memory. Fifty years on, garbage collection is no joke but an essential component of modern programming language implementations."

2.4.1 Terms

During this chapter we will talk about garbage collection in more details, but first let us create a shared vocabulary by defining important terms:

Tab. 2 Garbage collection terms

Term name	Term description
collector	Collector executes garbage collection code, discovers unreachable objects and reclaims their storage.
compaction	Compaction performs relocation, moves live objects together and reclaims contiguous empty space.
liveness of an object	Object is "life" if it is accessible from the root references (thread stacks, global storage, statics, ...).
memory fragmentation	Memory fragmentation emerges over time when contiguous dead space between objects may not be large enough to fit new objects.
mutator	Mutator executes application code, allocates new objects and mutates the object graph by changing reference fields
pause	Pause is a time duration in which the mutator is not running any code.
promotion	Promotion is allocation into an older generation.
stop-the-world	Stop-the-world is something that is done in a pause.

2.4.2 Garbage collection metrics

Comparing garbage collectors using various metrics can reveal their behavior, weak and strong characteristics. The following table summarizes garbage collection metrics:

Tab. 3 Garbage collection metrics

Metric name	Metric description
allocation rate	how fast data are allocated
compaction time	how long it takes the collector to free up memory by relocating objects
cycle time	how long it takes the collector to free up memory
heap population	how much of heap is alive
marking time	how long it takes the collector to find all live objects
mutation rate	how fast references in memory are updated
object lifetime	how long objects live
sweep time	how long it takes the collector to locate dead objects

2.4.3 Classifying collectors

Garbage collection theory has defined multiple types of collectors and the most important ones are:

- monolithic collectors,
- stop-the-world collectors,
- concurrent collectors,
- parallel collectors,
- incremental collectors,
- "mostly" variation of some of the previously mentioned collectors.

Once a monolithic collector is started, it cannot be stopped until it finishes the whole collection.

A stop-the-world collector (also known as serial collector) performs the garbage collection while the application is completely stopped.

A concurrent collector performs garbage collection work concurrently with the application's own execution.

A parallel collector uses multiple CPU cores to perform garbage collection.

An incremental collector performs a garbage collection process as a series of smaller discrete operations with gaps in between.

There also exist variations of previously mentioned collectors, the ones that contain "mostly" in their names. In this context "mostly" means sometimes is not and this usually means a different fall-back mechanism exists. For instance "Mostly concurrent" collector means concurrent as long as it could be and then it is not concurrent anymore and another kind of collector will perform the collection.

2.4.4 Conservative vs. precise collection

A collector is conservative if it is unaware of all object references at collection time, or is unsure about whether a field is a reference or not.

A collector is precise if it can fully identify and process all object references at the time of collection. A collector has to be precise in order to move objects. All commercial server JVMs use precise collectors.

All precise garbage collection mechanisms have the following steps in common:

1. identify the live objects in the memory heap,
2. reclaim resources held by dead objects,
3. periodically relocate live objects.

Mark-Sweep-Compact collector, which is commonly used for old generations, performs these operations in 3 different steps. Copying collector, which is commonly used for young generations, performs mentioned operations in one step.

2.4.4.1 Mark, sweep and compact phases

Mark phase, also known as Trace, starts from roots. In this phase all reachable objects will be "painted". At the end of the mark phase, all reachable objects will be marked "live" and all non-reachable objects will be marked "dead".

Sweep phase scans through the heap, identifies the dead objects and tracks them (usually in some form of free list or card map data structure).

Compact phase moves objects in memory and corrects all object references to point to the new object locations. At the end of the compact phase, heap memory will be defragmented.

2.4.4.2 Copy phase

Copy phase moves all live objects from a "from" space to a "to" space and reclaims "from" space.

At the start of the copy phase, all objects are in the "from" space and all references point to the "from" space. Copy phase starts from "root" references, copies all reachable objects to the "to" space correcting all references as it goes. At the end of the copy phase, all objects are in the "to" space and all references point to the "to" space.

2.4.5 Generational collection

According to the article *Garbage Collection in the Java HotSpot Virtual Machine* [15], the following two observations are known as "weak generational hypothesis":

- most allocated objects will die young,
- few references from older to younger objects exist.

That is the reason why most garbage collection algorithms will focus efforts on young generation. Because the live set in the young generation is a small fraction of the whole memory space, moving collectors are usually used for young generation (because the work is linear to the live set). Young generation serves as a "generational filter" and reduces the rate of allocations in older generations. Objects that live long enough will be promoted to older generations. Collectors keep surviving objects in young generation for a while before promoting them to the old generation. Immediately promotion can dramatically reduce generational filter's efficiency, but waiting too long to promote can dramatically increase copying work. Older generations will only be collected as they fill up.

Because most objects die young, the frequency of promotion is much lower than the frequency of allocation. Generational collection is a great way to keep up with high allocation rate. Generational collection tends to be more efficient by at least the order of magnitude.

The generational collection could be applied to multiple generations, but only young and old generations are practically used.

2.4.6 HotSpot Virtual Machine collectors

The HotSpot Java Virtual Machine currently contains the following 3 garbage collectors [16]:

- Parallel garbage collector,
- CMS[23] garbage collector,
- G1[24] garbage collector [17].

Let us summarize the main characteristics of all of them in the following subchapters.

2.4.6.1 Parallel garbage collector

The characteristics of parallel garbage collector within JRE[25] are:

- Parallel garbage collector is the default JRE GC[26].
- Parallel garbage collector uses monolithic STW[27] copying collector for young generation.
- Parallel garbage collector uses monolithic STW Mark-Sweep-Compact collector for old generation.

[23] Concurrent Mark-Sweep Garbage Collector
[24] Garbage-First Garbage Collector
[25] Java Runtime Environment
[26] Garbage Collector
[27] Stop-The-World Garbage Collector

2.4.6.2 CMS garbage collector

The characteristics of CMS garbage collector within JRE are:

- CMS garbage collector can be turned on by specifying the `"-XX:+UseConcMarkSweepGC"` argument as part of the java command.
- CMS garbage collector uses monolithic STW copying collector for young generation.
- CMS garbage collector uses mostly-concurrent, non-compacting Mark-Sweep collector for old generation. Mostly-concurrent means mostly-concurrent marking and concurrent sweeping phase with fallback to full collection (monolithic STW compaction).

2.4.6.3 G1 garbage collector

The characteristics of G1 garbage collector within JRE are:

- G1 garbage collector can be turned on by specifying the `"-XX:+UseG1GC"` argument as part of the java command.
- G1 garbage collector uses monolithic STW copying collector for young generation.
- G1 garbage collector uses mostly-concurrent marking phase and STW mostly-incremental compacting phase for old generation with fallback to full collection (monolithic STW compaction of popular objects and regions).

There are two problems with all the mentioned standard Java garbage collectors:

1. Young generation always uses STW collector which will eventually cause unacceptable pauses with big enough heaps.
2. Old generation always uses STW compaction phase which will again cause unacceptable pauses with big enough heaps.

The only currently known JVM that does non-STW compaction phase and both young and old generations use truly concurrent collectors is Azul's C4[28] collector [18].

[28] Continuously Concurrent Compacting Collector

2.5 Original GCSpy project

Let us first start with the description of the original GCSpy project [19]:

"GCSpy is a heap visualization framework. It is designed to visualize a wide variety of memory management systems, whether they are managed by a garbage collector or implement explicit deallocation. Its target users are mainly memory management implementers and it allows them to observe the behavior of their system. GCSpy is not limited to small toy systems; it can also visualize loads of realistic sizes, and do so dynamically, as the system operates. However, if the user needs it, a trace storing and replaying facility is also available."

Original GCSpy could visualize any software system that was created by a number of components (usually low number) and these components can be further divided into a number of partitions (usually large number).

Original GCSpy allowed the visualization of both local and remote software systems and the visualization of already running system was also possible. Original GCSpy was especially suited for visualizing the heap memory fragmentation.

As the program executes, the state of the heap changes because the program acts as a mutator. The original GCSpy project periodically captured the current state of the heap and sent it to the visualizer.

2.5.1 GCSpy abstractions

According to the research paper *GCSpy: An Adaptable Heap Visualisation Framework* [20], the original GCSpy project defined the following abstractions:

- spaces,
- blocks,
- streams.

2.5.1.1 Spaces

Each system consists of one or more components. Each component consists of one or more spaces. The heap memory is an example of such a component. In the case of the generational garbage collection, generations are examples of spaces.

Each space is described by the following characteristics:

- space name,
- number of blocks (size),
- space streams.

2.5.1.2 Blocks

Each space consists of one or more blocks. The memory block with well-defined size is an example of a block. Blocks define granularity of the visualization. Blocks from within different spaces do not have to have the same size.

Every block of the same space has exactly the same attributes. Blocks are visually represented as tiles and have an integer fixed-range values.

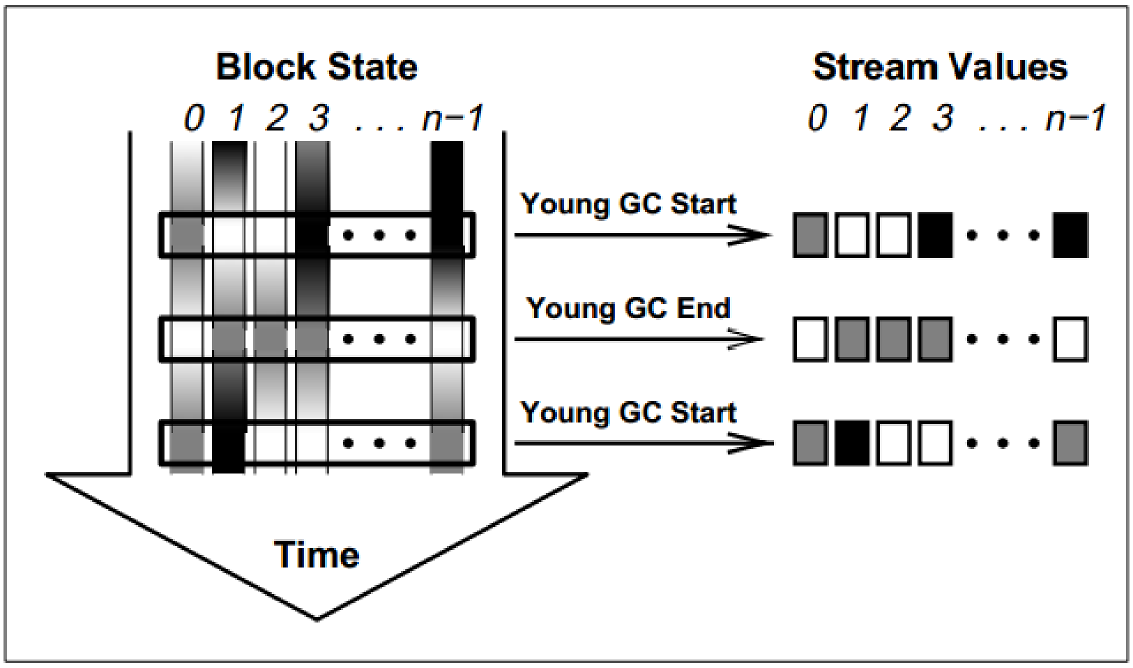


Fig. 4 Original GCSpy data gathering

2.5.1.3 Streams

Each block within a space could have multiple attributes and all blocks of the same space have exactly the same attributes. These attributes are modeled as streams.

The current state of the stream is described as a sequence of integers, one integer value per each block.

Each stream is described by the following characteristics:

- stream name,
- stream description,
- stream type,
- type and range of stream data values,
- other presentation information.

2.5.2 GCSpy architecture

The architecture of the original GCSpy was strictly client-server with TCP/IP[29] socket communication. This choice was made mainly because the need of language independence and performance. From the architecture point of view, the observed system acts as a server and the visualization GUI[30] acts as a client.

Data transmission between client and server usually happens at "safe points". Safe point represents a STW situation where all mutator threads (every thread except memory observer thread) are stalled.

The communication between the client and the server is performed through a driver. The driver has 2 responsibilities:

- do the mapping between the collected data and the appropriate streams within spaces,
- collect any additional data (e.g. control and summary streams).

Driver extracts data specific for every garbage collection algorithm and provides the data further to the lower layers of GCSpy. Every driver is specific for a particular system (e.g. JVM) and component (e.g. various GC algorithms).

[29] Transmission Control Protocol / Internet Protocol
[30] Graphical User Interface

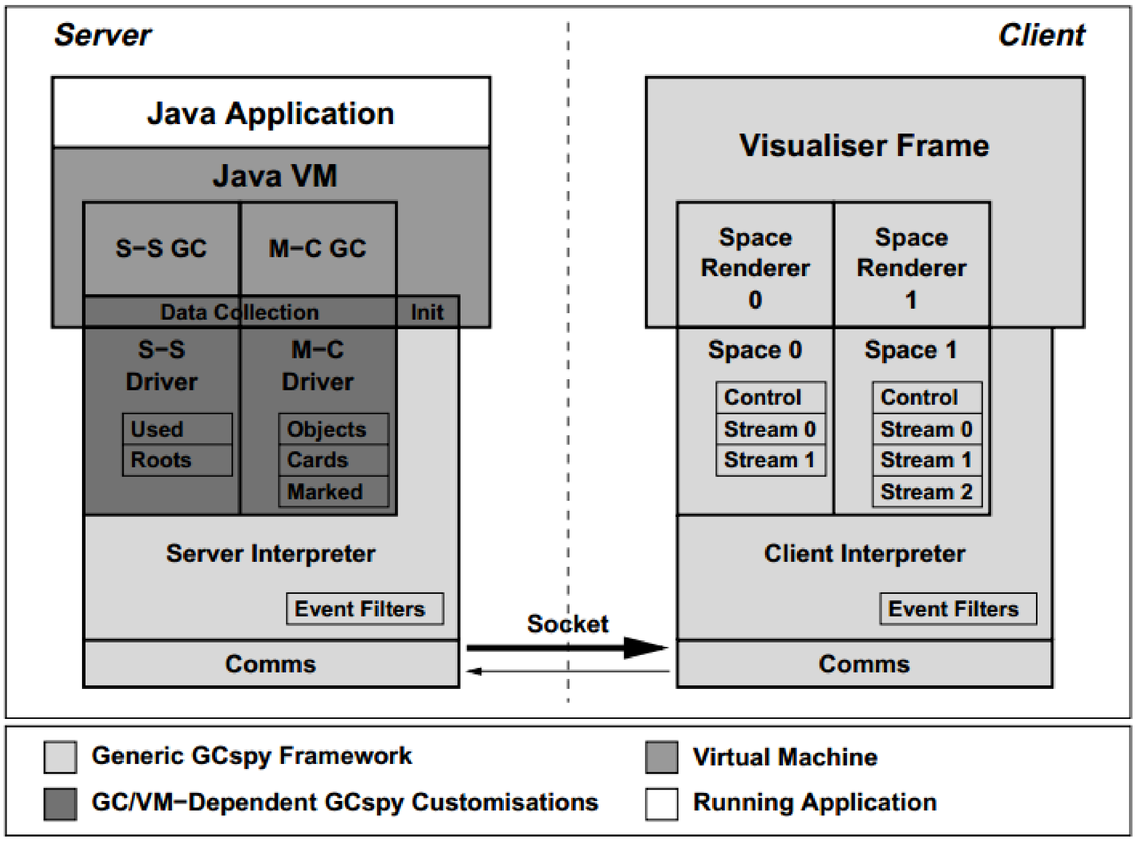


Fig. 5 Original GCSpy architecture

3 Proposal

After summarizing all the main technologies, used concepts and necessary theoretical background, let us describe proposed main design goals together with high level system overview.

3.1 Design goals

Main design goals of proposed GCSpy are:

- the ease of use,
- modular design and reusable components,
- non-intrusiveness,
- preciseness of visualization,
- transparency,
- virtual machine independence.

The ease of use is one of the most important goals of the project. A user should be able to use the system in a very easy way although the initial process of setting the system up does not have to be trivial (because of the overall complexity of the system).

Very crucial aspect of the system is modular design and reusable components. System should be composed from modules with proper dependencies from day one. General functionality should be abstracted and reusable for other parts of the system (build, runtime and test infrastructure) as well as end users (GCSpy Data Provider API).

To provide good user experience, non-intrusiveness is very important. The system should be as performing as possible and should not cause huge performance penalties. System could apply performance optimization techniques if those will be done behind the scenes and transparently.

Preciseness of visualization includes two aspects. The visualization (and thus also the data gathering code) should provide fine-grained data. Preciseness also means lowering the amount of data in the heap that are not related to the observed application, but are used because of observing code (e.g. anything starting from Java sockets to higher level concurrency support).

The need of transparency is obvious because of the system complexity. There are two examples of transparency that system should provide. From user perspective, system should be transparent in the way how the application will be started. The difference between the start up of the application with and without observation should be minimal. From the programmer perspective, the build process should be transparent even if native code (and its build and linkage) is part of the code base. The data provision should be transparent as well. It does not matter what was the real source of the data (e.g. real observed JVM or just a stored trace on disk), but all should implement the same interface.

Last, but not least virtual machine independence plays an important role in the overall design. Unlike original GCSpy, proposed solution should be virtual machine and garbage collection algorithm independent and thus the portability, with comparison to the original GCSpy, should be easier to achieve.

3.2 High level overview

The system should be designed according to the following figure:

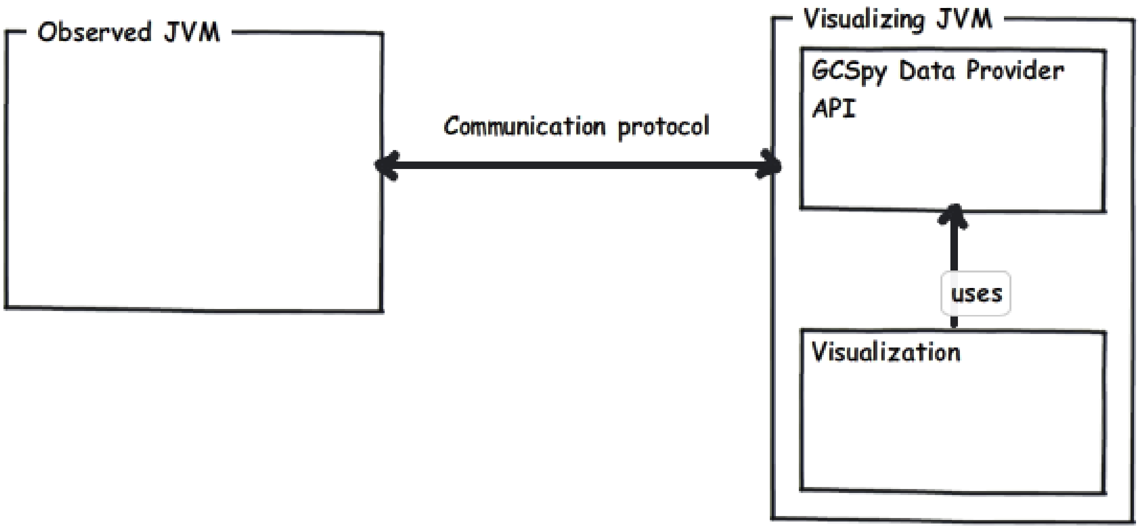


Fig. 6 Proposed GCSpy high level overview

The essential part of the high level overview is the GCSpy Data Provider API. The visualization should use the API and should not depend on any of the API implementations. One of the implementations should be using sockets with a well-defined communication protocol between the observed and visualizing JVM.

4 Implementation details

This chapter will focus on the implementation details. We will go through the build infrastructure, runtime infrastructure, test infrastructure, socket communication protocol, portability and we will put things together in the last chapter.

Some of the following chapters will be divided into these three subchapters:

1. "Initial considerations",
2. "Design decisions",
3. "How does it work?".

The part “Initial considerations” summarizes the initial thoughts and considerations I made either before further implementation or in the very early stage of the development process. There were always multiple ways how to solve a given task and that part will be summarizing possible options.

The part “Design decisions” summarizes decisions I made during the development process together with explanations of the reasons and comparisons of possible options including their advantages and disadvantages.

The part “How does it work?” describes in more details how things work, what is required to do in order to make things working correctly. This part also contains limitations of the given solution.

4.1 Build infrastructure

Build infrastructure was clearly one of the most important parts of the application design. It was not trivial to develop such infrastructure totally from scratch, but from the long-term perspective, it definitely deserved its cost of development. Being able to build not only Java classes, but also native code as part of one central step is a huge win.

One of the main design goals of the build infrastructure was not to be forced to commit any binary data into SCM[31] tool. This complicated the development process quite significantly, especially from the beginning until the build, runtime and test infrastructures were fully developed, but later proved as a good idea.

As a result you are able to build the whole application including Java and C source code into proper deployment units within one build step and no binary data will have to be committed into SCM repository.

4.1.1 Initial considerations

The decision to choose Maven as a build infrastructure technology was obvious right from the development beginning. The question was how the native code (in our case written in the C language) will be compiled and linked. There already exist 2 Maven subprojects trying to introduce C/C++ support for Maven:

1. Maven Native plugin (supported directly by Sonatype),
2. Maven NAR[32] plugin (community driven effort).

However none of them was chosen for building the native part of GCSpy mainly because they did not really fit into the overall GCSpy architecture. First let us focus on the Maven Native plugin and its disadvantages:

- Maven Native plugin is not under active development and is still in the *alpha* development cycle.
- Maven Native plugin uses *customized lifecycle*, however it is quite complex and still not powerful enough to do simple tasks easily.
- Maven Native plugin uses `"exe"` or `"uexe"` packaging type for producing executable files. This is however not flexible because you need to define 2

[31] Source Code Management
[32] Native Archive

Maven projects with different packaging types if you would like to build the same code to be executable on different platforms.

- Maven Native plugin uses `"dll"` and `"so"` packaging types for producing shared libraries which introduces exactly the same problem as mentioned with `"exe"` and `"uexe"` packaging types.
- Maven Native plugin supports only `GCC`[33] + `GCC`, `BCC32` + `ILINK32` and `CL` + `LINK` as compiler and linker pairs.
- Maven Native plugin requires you to define the proper packaging type, compiler and linker provider together with environment factories (most of which could be automatically determined according to the underlying running platform).
- Maven Native plugin's lifecycle contains `"install"` and `"deploy"` phases. This does not fit into the GCSpy architecture at all. It does not make sense to do the installation into the local Maven artifact repository or the deployment into the remote Maven artifact repository.

Secondly, let us focus on the Maven NAR plugin and its disadvantages:

- Maven NAR plugin is much more flexible (for instance it supports integration tests among other interesting features), but it is still missing some essential features.
- Maven NAR plugin will produce a NAR archive (zip compressed file with all native OS[34] and platform specific binaries) and will install packaged modules into local Maven artifact repository.
- Maven NAR plugin's lifecycle contains `"install"` and `"deploy"` phases. It is again the same problem as with Maven Native plugin.
- Maven NAR plugin will do the compilation and the linkage process together in one step. This effectively blocks any further considerations about choosing this plugin.

As a result, none of those 2 Maven plugins was suitable for what I needed to achieve and thus I have decided to develop my own Maven packaging type together with customized build lifecycle.

[33] GNU Compiler Collection
[34] Operating System

4.1.2 Design decisions

Unlike Maven Native plugin or Maven NAR plugin, I have created `"shared-library"` packaging type, which differs from both of them quite significantly. Let us summarize the most important design decisions and characteristics of the custom `"shared-library"` packaging type:

- The only supported packaging type is `"shared-library"` which will produce a shared library, whatever it means on the underlying platform (on Windows machines it will produce a `DLL`[35] file and on Linux/Unix/Solaris machines it will produce a `SO` file). This allows to create just one Maven project with C source files and package them automatically according to the running platform.
- There is no direct way how you can produce any other deployment format (like for example executable file). This is intentional, but appropriate API backs that kind of functionality.
- Current support includes Windows platform together with MSVC[36] compiler and linker.
- There is an easy way how you can add additional compiler support which will be automatically recognized on defined platform.
- The only thing you need to do, in order to execute the packaging, is to define the `"shared-library"` as packaging type and add compiler and linker executables to the PATH environment variable. Everything else will be handled automatically behind the scenes (the default platform compiler and linker pair will be selected and any required system properties, environment variables and settings will be set).
- There is neither direct nor indirect mapping into Maven `"install"` or `"deploy"` phases because it does not make sense to install binary artifacts into local or remote Maven artifact repository.
- There is a build-in support for integration tests within the packaging type (using standard Maven Failsafe plugin). As a result you do not need to specify the Maven Failsafe plugin explicitly in the project's POM that uses

[35] Dynamic Link Library
[36] Microsoft Visual C++

the `"shared-library"` packaging and so the project POM is more clear and concise.

- Compile and linkage processes are not done in the same step. This turns out to be a very good and useful idea and I am taking advantage of the separate build and link process within tests.
- The compiler and linker abstractions are implemented as a well-defined API. This allows me to be able to do the programmatic compilation and linkage.
- Some of the compiler and linker options were general enough to be abstracted away. For instance you should be able to link a set of object files into an executable file or a shared library (whatever it means on underlying running platform). Another example of a very common option is the ability to define the output directory (output directory is actual compiler working directory where all compiled or linked files will be created). Last but not least, the API also handles compiler standard and error output. This is important not only for Maven MOJOs[37], but also for testing purposes and Groovy runner script.

  For the above mentioned reasons, those functionalities are abstracted away to be part of the API. Most of the remaining attributes are not general enough to be abstracted away but they are rather compiler or linker specific. The build infrastructure is flexible enough to be able to specify any number of compiler and linker arguments as a set of *String* based options passed to the compiler and linker.

The reason why I am mentioning those design decisions in details is the fact that most of the other parts of the system were designed in a very similar fashion (find minimal suitable abstraction that will be abstracted away as part of the API, encapsulate what varies, use proper level of abstraction, ...).

[37] Maven plain Old Java Object

4.1.3 How does it work?

Instead of creating a completely new Maven build lifecycle, I decided to reuse and customize the default Maven build lifecycle (mainly because the default Maven build lifecycle contains 23 phases and is flexible). I am only mapping my MOJOs together with a subset of chosen standard Maven MOJOs to the final GCSpy build lifecycle. Only the phases required by GCSpy are part of the mentioned mapping.

There is one fundamental difference between the lifecycle mappings of both the Maven Native plugin as well as the Maven NAR plugin and GCSpy `"shared-library"` plugin. When I was thinking about the mapping that should be used for C-based projects I have found similarities between C compilation/linking and Java compilation/packaging processes.

From the Java perspective, source code is saved in `*.java` files and then compiled into `*.class` files. Those `*.class` files can be used independently, but usually they will be packaged into bigger deployment units (JAR, WAR[38], EAR[39], ...) with well-defined structure described by appropriate JSRs. In the case where a JAR file contains appropriate key-value pair as part of its manifest to define the main method ("Main-Class") or a WAR file contains appropriate data in its deployment descriptor, the deployment unit could be understood as "executable".

I tried to apply similar semantics to the C compilation/linking MOJOs. Both Maven Native plugin and Maven NAR plugin do the compilation/linking either together in one step, or as two independent steps one after another. In their case the `"package"` phase is used to package all the produced files into Zip archives. In the case of GCSpy, `"compile"` phase is mapped to Compilation MOJO and the linkage will be performed later as the part of the `"package"` phase where linkage will produce the shared library.

For the purpose of `"shared-library"` packaging, the following phases are mapped into standard Maven MOJOs:

[38] Web Archive
[39] Enterprise Archive

- `process-test-resources`,
- `test-compile`,
- `test`,
- `integration-test`,
- `verify`.

The exact mapping is showed on the following figure:

```
<compile>
    org.crazyjavahacking.gcspy:
          org-crazyjavahacking-gcspy-shared-library-packaging:compile
</compile>
<process-test-resources>
    org.apache.maven.plugins:maven-resources-plugin:testResources
</process-test-resources>
<test-compile>
    org.apache.maven.plugins:maven-compiler-plugin:testCompile
</test-compile>
<test>
    org.apache.maven.plugins:maven-surefire-plugin:test
</test>
<package>
    org.crazyjavahacking.gcspy:
             org-crazyjavahacking-gcspy-shared-library-packaging:link
</package>
<integration-test>
    org.apache.maven.plugins:maven-failsafe-plugin:integration-test
</integration-test>
<post-integration-test>
    org.crazyjavahacking.gcspy:
           org-crazyjavahacking-gcspy-shared-library-packaging:verify
</post-integration-test>
<verify>
    org.apache.maven.plugins:maven-failsafe-plugin:verify
</verify>
```

Fig. 7 GCSpy Maven lifecycle mapping

The following Maven best practices are used along the GCSpy project:

- Convention over configuration:
    - Standard directory layout for projects:

      `src/main/native/header` - directory for all C header files

      `src/main/native/source` - directory for all C source files

      `target/obj` - directory where all object files will be created

      `dist/sharedLibrary` - directory where shared library will be created

  The compilation step will compile all C source files located in `"src/main/native/source"` directory and put them into `"target/obj"` directory. The final shared library will be linked from all produced object files located in `"target/obj"` into `"dist/sharedLibrary"` directory relative to Maven parent project.
- One primary output per project
- Reuse of build logic:

  Both compiling and linking MOJOs will just reuse prepared utility classes, which are part of the `"utilities"` module. Other parts of the system infrastructure could reuse those parts as well. One example of such reuse is unit tests.
- Declarative execution:
    - Maven's project object model:

      The usage of both plugins is wrapped by packaging type which is used in a declarative way and executed automatically every time a build will run.
    - Maven's build life cycle:

      The fact that I am reusing default Maven build life cycle allows further customization.

4.1.4 Dependency management

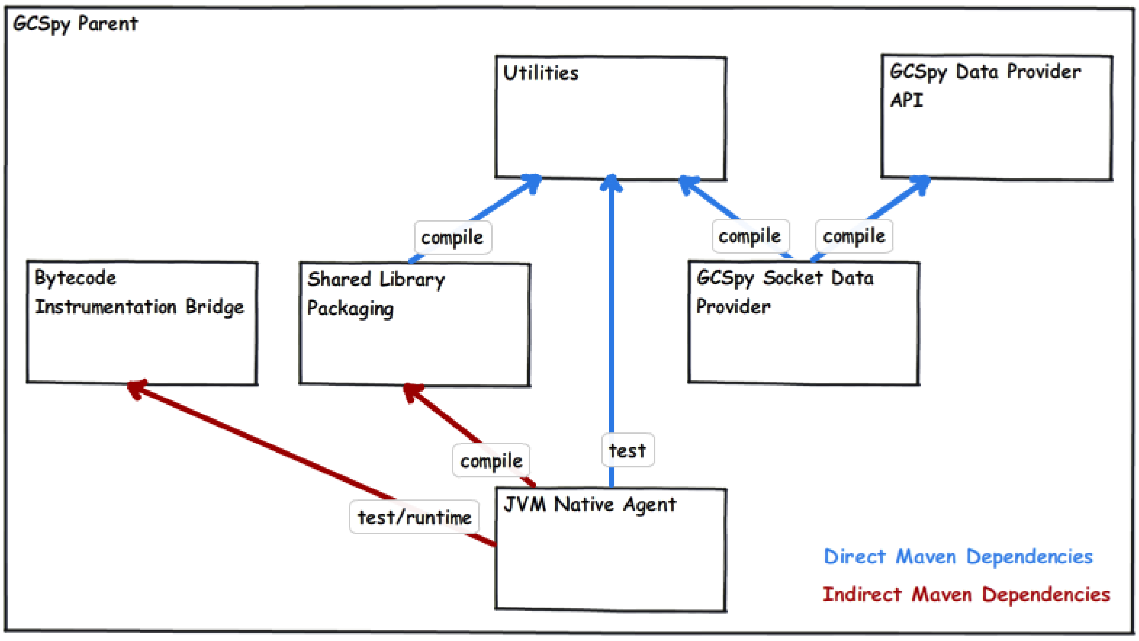


Fig. 8 GCSpy Maven dependency management

The upper figure visualizes Maven dependency management. As you can see, project currently contains the following Maven modules:

- Utilities,
- Bytecode Instrumentation Bridge,
- Shared Library Packaging,
- JVM Native Agent,
- GCSpy Data Provider API,
- GCSpy Socket Data Provider.

The purpose of GCSpy Parent module is to unify the properties, versioning of dependencies, build plugins and profiles.

Utilities module contains various reusable components including compiler and linker abstractions, external process execution wrappers, file utilities, file system watching wrappers, socket utilities, test utilities and utility classes specific to operating system and virtual machine.

Bytecode Instrumentation Bridge contains just one class serving as bridge between Java calls (which were added as a result of the bytecode instrumentation process) and native calls (returning the execution back to the native code).

Shared library packaging contains all Maven specific code required to integrate the `"shared-library"` packaging. This includes 3 MOJOs - compiling, linking and verifying MOJO. Module contains lifecycle mapping as well.

JVM Native Agent is the most important module, the one that contains core HotSpot VM[40] integration. This includes bytecode instrumentation, server socket implementation and JVMTI bridge.

GCSpy Data Provider API is a high level object-oriented java API, which allows you to register object, VM and GC type of listeners.

GCSpy Socket Data Provider is a socket-based GCSpy Data Provider API implementation. In fact it is the high-level counterpart to the low-level socket-based communication protocol.

Very important aspect is the fact that there is no direct or indirect dependency from JVM Native Agent to GCSpy Socket Data Provider or vice versa. This is exactly how it should be because C code cannot have dependency to any java code and java client socket implementation has no dependency to the C-based server socket.

There are 4 types of dependencies:

- direct compile dependency,
- direct test dependency,
- indirect compile dependency,
- indirect test/runtime dependency.

Direct compile dependencies are required in order to compile the module. Those dependencies will be automatically put on compile classpath.

Direct test dependencies are required in order to test the module. Those dependencies will be automatically put on test-compile classpath.

Direct Maven dependencies are referenced by Maven `<dependency>` declarations. They will effectively dictate the order in which the modules will be compiled.

[40] Virtual Machine

Indirect dependencies are not forced by Maven in any way. They exist either because a module needs another module to function properly, but the dependency is not required during the compilation (JVM Native Agent requires Bytecode Instrumentation Bridge at runtime, otherwise runtime error will occur) or because a module is packaged using nonstandard packaging type that is defined in another module (JVM Native Agent requires Shared Library Packaging, otherwise the build process will fail).

4.1.5 Maven profile activation strategy

Maven profiles are a way how to customize the build process. In the GCSpy project, profiles are useful because they will allow you to build only "shared-library" packaging, determine if the final shared library will be 32bit or 64bit and allow you to pass compiler and linker specific options. Currently, the GCSpy project contains the following profiles:

- `buildOnlySharedLibraryPackaging`,
- `buildAllModules`,
- `32BitNativeAgent`,
- `64BitNativeAgent`,
- `windowsOS`.

The first profile "`buildOnlySharedLibraryPackaging`" will only build the "`shared-library`" packaging. The purpose of this profile will be described in the following chapter. This profile has to be activated manually mentioning it on the command line.

The second profile "`buildAllModules`" will build all modules. This profile will be activated by default.

The third and fourth profiles "`32BitNativeAgent`" and "`64BitNativeAgent`" will set the "`nativeAgentBits`" system property to 32 or 64. This system property will be propagated to both Maven Surefire and Failsafe plugins. Most of the users will probably use the 32bit JVM that is why the "32BitNativeAgent" will be selected by default and the latter one will have to be activated manually on the command line.

The last profile "windowsOS" is used to set all Windows operating system specific settings to build plugins (including compiler and linker options). This plugin will be activated automatically on Windows operating system.

Now let us summarize all the aspects of Maven profile activation in the following table:

Tab. 4 GCSpy Maven profile activation strategy

Profile id	Profile activation	Activation process	Modules to build
buildOnlySharedLibraryPackaging	manually	by mentioning the profile explicitly on CL[41]	`utilities, shared-library-packaging`
buildAllModules	automatically	by default	`utilities, shared-library-packaging, bytecode-instrumentation-bridge, jvm-native-agent, gcspy-data-provider-api, gcspy-socket-data-provider`
32BitNativeAgent	automatically	by default	-
64BitNativeAgent	manually	by mentioning the profile explicitly on CL	`utilities, shared-library-packaging, bytecode-instrumentation-bridge, jvm-native-agent, gcspy-data-provider-api, gcspy-socket-data-provider`
windowsOS	automatically	on Windows OS	-

[41] Command Line

4.1.6 Building modules

Until now everything looked like customized, but not very special Maven-based project. Here comes the place where it is getting more interesting.

You should be able to compile the project executing:

```
  mvn clean install
```

Maven will try to compile the modules in the following order:

1. GCSpy Parent
2. Utilities
3. Shared Library Packaging
4. Bytecode instrumentation bridge
5. JVM Native Agent
6. GCSpy Data Provider API
7. GCSpy Socket Data Provider

What might be not so obvious from the Fig. 8 GCSpy Maven dependency management, is the consequence of the indirect compile dependency of "`jvm-native-agent`" module to "`shared-library-packaging`" module. The important point is the fact that the "`shared-library-packaging`" module is part of the GCSpy parent module and thus will be built every time the whole application will be built. This decision was made in the very early stage of the development process and I still think it was a good choice. The "`shared-library-packaging`" module is not general-purpose packaging type with tons of settings to build the final shared library, but it was created specifically for "`jvm-native-agent`" module. All of the implemented features were driven by requirements from the "`jvm-native-agent`" module. That is why it is part of the multi-module Maven project and not in an independent Maven project. The question is: Why do I mention this issue?

It is important because for the first time, the mentioned command will not work and the build will fail. The problem is the fact that the packaging type is a part of the same module hierarchy and it is not a standalone Maven module. Further, Maven treats packaging with higher priority in comparison to common modules, it will fail because it will be not able to find or download "`shared-library`" packaging (not available in the local Maven artifact repository yet and not exported to the Maven Central repository at all).

The overall build will fail because Maven will not be able to find the packaging type. However the packaging type cannot be compiled until the parent module will be successfully resolved. And parent module will not be successfully resolved until all of its child modules (including the packaging) will be build. As a result we have just described a "deadlock-like" situation.

Now the purpose of the "`buildOnlySharedLibraryPackaging`" profile should be obvious. For the first time one will build the project, one is required to build it using the mentioned profile. So for the first time, you need to execute the following Maven command:

```
    mvn -P buildOnlySharedLibraryPackaging clean install
```

4.2 Runtime infrastructure

Because of the overall build infrastructure complexity, running the whole application is not as easy as it might look like. In the simplest situation you have to set *bootclasspath*, *agentpath* and multiple system properties (not including any other Java interpreter options every user can specify).

Nobody really wants to write extensive command on command line and hope that there is no typo in that command. To simplify the experience of starting the application, the notion of runtime infrastructure was added.

4.2.1 Initial considerations

Most of the open-source projects use system specific launcher scripts (`*.bat` files on Windows and `*.sh` on Linux/Unix). I did not want to do that mainly because of the need to maintain multiple scripts written in different scripting languages.

Instead of doing that, I decided to use Groovy. Groovy is a dynamic programming language that runs on top of the JVM. Groovy was chosen because of its very similar syntax to Java, its dynamic typing and scripting capabilities. At the end of the day you will have to maintain only one script.

One important aspect that every user should have in mind is the fact that the Groovy script is just a recommended way to start the whole java application with given JVM native agent and required libraries. Nobody is forced to use it and one can write native OS shell scripts or build the final java command manually.

4.2.2 Groovy scripting advantages

Right after I established the runtime infrastructure it turned out that using Groovy for scripting added multiple advantages:

1. You do not have to maintain multiple scripts (adding code to multiple scripts, synchronizing between them and the necessary testing on multiple platforms).
2. Because of Groovy powerful language construct called Grape, I was able to reuse statically typed code already implemented and well tested in the `"utilities"` module. Grape is a Groovy language extension that allows

you to reuse Maven artifacts in Groovy script (both from local Maven artifact repository and remote Maven Central artifact repository). Important thing is the fact that Grape will put the artifacts to runtime classpath of the Groovy script. This will however not change the runtime classpath of the final java command.

3. Runtime infrastructure is validated every time the build will run. In fact integration tests use the Groovy runner script. This significantly reduces the code needed to run the integration tests by simplifying the application startup.

4.2.3 Design decisions

The fact that Groovy script will run in a fully initialized JVM environment turns out to be essential. You can get any of the system properties and pass them to the final Java command as command line arguments. Operating system specific scripts will need to gather the values of Java system properties in other way (they will probably need to startup a JVM anyway).

The final java command will be executed as an external process. For external process execution a process execution wrapper from "`utilities`" module is used instead of build-in Groovy process support (mainly because of much higher flexibility).

Groovy script will accept various command line options and will preprocess, filter and transform them into the final java command. Let us be more specific about what Groovy script will really do.

4.2.4 Groovy script enrichment

Groovy script will do a lot of things behind the scenes. It might not be obvious, but Groovy script will:

1. Accept and validate the existence of the following arguments:
    - `-nativeAgentBits`,
    - `-nativeAgentDebugPrint`,
    - `-serverSocketHost`,
    - `-serverSocketPort`,

  - any other arguments starting with the prefix `"-J"`.
2. Transform Groovy arguments into Java arguments or system properties by:
   - transforming optional argument `"-nativeAgentDebugPrint"` into `"-DnativeAgentDebugPrint"`,
   - transforming required argument `"-serverSocketHost=..."` into `"-DserverSocketHost=..."`,
   - transforming required argument `"-serverSocketPort=..."` into `"-DserverSocetPort=..."`,
   - removing the `"-J"` prefix from any argument starting with that prefix.
3. Add additional argument `"-DhostJVMArchitectureBits=..."`.
4. Enrich the bootclasspath by adding all JAR files from `"dist/bootclasspath"` directory and the end of bootclasspath.
5. Register JVM native agent from `"dist/sharedLibrary"` directory.
6. Improve performance by:
   - removing `"-client"` argument if available,
   - adding `"-server"` argument if not already available,
   - adding `"-XX:+TieredCompilation"` if not already available.

As you can see, some of the arguments have no meaning outside the Groovy script and thus need to be transformed into Java arguments. The transformation will create additional system properties and Java command arguments. Groovy script will enrich the bootclasspath by adding items at the end of it and register a JVM native agent.

Groovy script will also optimize the performance. Even if you explicitly define to execute the java code using the HotSpot C1 compiler (`"-client"` switch), Groovy will filter this option and the final command will always execute the HotSpot C2 compiler (using `"-server"` switch).

Groovy script will also turn on the newly added JDK 7 feature called "Tiered compilation" automatically.

4.2.5 How does it work?

Let us say you are running you application with following java command:

```
java -server -cp build/classes ... myPackage.MainClass
```

or similarly executing a main class part of a JAR file:

```
java -client -cp build/dist ... -jar Main.jar
```

In general you are executing either

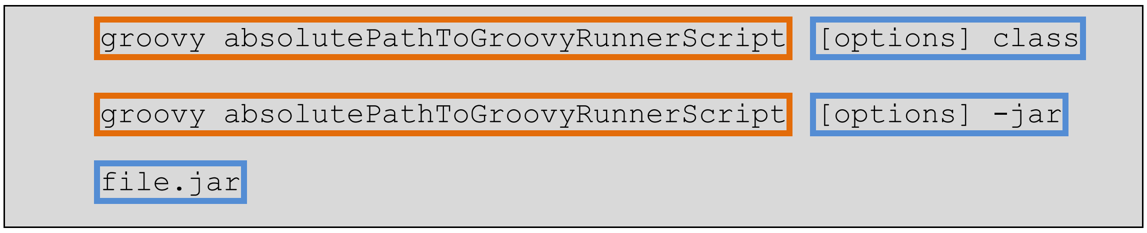
```
java [options] class
```

or

```
java [options] -jar file.jar
```

which could be practically handled as a special case of the first command. For both cases incorporating the Groovy runner script should be as seamless as possible. Ideally, just putting the Groovy command together with the Groovy runner script should be enough.

In the GCSpy case, the general scheme is to use

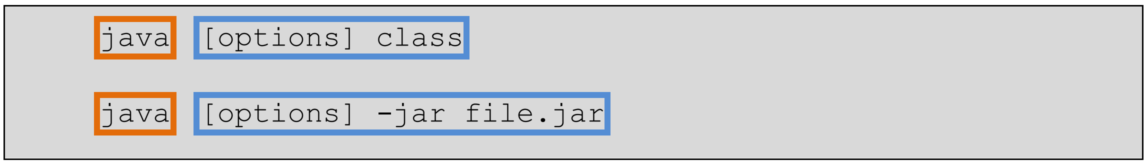


```
groovy absolutePathToGroovyRunnerScript [options] class

groovy absolutePathToGroovyRunnerScript [options] -jar
file.jar
```

Fig. 9 Executing GCSpy with Groovy runner script

instead of



```
java [options] class

java [options] -jar file.jar
```

Fig. 10 Executing GCSpy with Java command

As you can see, the only difference is the replacement of the java command by groovy command and the addition of "`absolutePathToGroovyRunnerScript`" at the end of the groovy command. Everything else remains the same.

4.3 Test infrastructure

Test infrastructure is the last missing puzzle piece to the overall understanding of the overall application architecture. The test infrastructure always plays an important role in software development and our project is no exception. Testing infrastructure includes both "Unit" as well as "Integration" testing.

The best way to start is to quote a very interesting idea from David Farley, the author of the book Continuous Delivery, which he told as part of his talk *Continuous Delivery* during Devoxx 2011 conference:

"Build binaries once. We want to be damn sure that the software we were performance testing, the software we were acceptance testing, the software that gets released into manual test environments, the software that gets released into production is precisely the same version that we build and verified earlier in the stage. Otherwise those tests are at some level invalid."

This idea turns out to be a crucial aspect which significantly influenced the test infrastructure. Building binaries once is important in Java, but even more important in native code (like C), where all the optimizations are done at compile time. If you will compile the C source codes with and without the optimizing `"/O2"` compiler argument, it will definitely produce different binary files. In general you can specify a lot of compiler options effectively creating different binary packages.

Different parts of the C source base are located in different header and source files. Those compiled parts needed to be tested independently and ideally outside the integration testing.

Let us describe how the socket communication is tested. After the compilation of `winsock2ServerSocket.c` on Windows platform, `winsock2ServerSocket.obj` object file will be created. It is however not something we can directly execute because it is neither executable nor shared library file. There is a C source file, with the *main* function that uses the functionalities defined in `serverSocket.h` header file, for testing purposes. That testing file will be first compiled and then linked together with *WinSock2* server socket implementation into an executable file. This executable file will be executed in the background thread waiting for client connection and then sending it test data. From within the test method java socket client will be created and all the received data will be checked. Other native (C-based) parts are tested in similar way.

4.3.1 Dealing with garbage collection indeterminism

One generally interesting aspect of the garbage collection is the transparency. Garbage collection will be handled behind the scenes and the programmer has no direct option to influence the garbage collection process. From the programmer point of view, the garbage collection is a nondeterministic process because the programmer has no idea if and when the garbage collection runs.

Because of the garbage collection transparency, no further implementation details of garbage collection are part of the JLS[42] [21] or JVMS[43] [22]. This is how it should be, however this also makes the testing process much more difficult. Without deeper understanding of garbage collection algorithms and HotSpot virtual machine specific details, testing process is tricky.

Even if there is a standard Java API for invoking garbage collection by calling `System.gc()` method, after careful reading the indeterminism leaks out. The method contract states that:

"Calling the `gc` method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects."

Did you catch it? The method contract does not guarantee that the garbage collection will really run. So in practice no code, including the tests, could ever rely on that assumption.

The Javadoc of the `OutOfMemoryError` exception class is pretty much the only guarantee about the garbage collection in every current JVM implementation, with the following content:

"Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector."

[42] Java Language Specification
[43] Java Virtual Machine Specification

That practically means that you will not get an `OutOfMemoryError` exception if there is a free space in the memory heap (either directly free, or occupied by no longer used objects that will be removed by garbage collector) and the space will be big enough to allocate requested object.

Now let us focus on concrete problems. From the test perspective, there are two problematic situations we need to test:

1. we need to force the garbage collector to run,
2. we need to force the garbage collector to move objects in the heap memory.

There is the need to test both of these situations because these functionalities are part of the GCSpy Data Provider API.

So, how did I solve the garbage collection indeterminism problem?

The first situation was not that hard. I was able to make the garbage collection process deterministic for a concrete test case by specifying the special combination of JVM arguments. These arguments were explicitly setting the heap sizes (both initial and maximum). JVM initialized in such a way, together with specially prepared test method, results in garbage collection determinism. It is important to say that it was possible only because the thread model of testing method was trivial. In real-world applications, where multiple threads are running simultaneously and thus all of them can allocate objects, the thread model prevents the previously mentioned solution to be 100% predictive and deterministic.

The second situation was more tricky. There is practically no guaranteed way how to determine the address from java object using a java API. Moving objects in memory is similarly transparent as the whole garbage collection process. Again, no further assumptions can be made about when an object could change its native memory address. This fact was confirmed by David Holmes, Senior Java Technologist, Java SE VM Real-Time and Embedded Group, as part of our mailing conversation on "hotspot-dev" mailing list where he answered:

"Object addressing is an implementation artifact and so has no mention in either the VM or language spec."

It is clear that as part of the garbage collection (especially using the copying algorithms) the native object memory address could change during the VM lifetime. But, are there any other circumstances when an object could change its address? If such circumstances exist, then the questions is: when and why? I assumed that HotSpot should not move objects in memory except during the garbage collection and I was right. This was again confirmed by David Holmes:

"I'm fairly sure that in Hotspot only the GC will move objects. Any movement requires updating all references to the object and that requires locating them, which requires at least some phases of the GC to execute."

That practically merges both mentioned problematic situations into the same one: How to force the garbage collector to run?

4.4 Socket communication protocol

Now let us focus on the definition of the socket communication protocol. Socket communication protocol in the GCSpy case is a high-level term specifying the following items:

- send data semantics,
- connection management,
- thread model,
- blocking states.

Programmer should be able to implement additional implementation of the GCSpy Data Provider API built on top of the communication protocol just by reading the following contract.

The whole chapter will focus on the socket communication contract.

4.4.1 Sent data semantics

Socket is a communication abstraction to send and receive data over computer network. From the programmer point of view, the real data that programmer will be working with are streams of bytes. In object-oriented languages this could be wrapped into higher-level objects (in Java it will be `InputStream` and `OutputStream`).

The real question, which arises from the general "stream of bytes" abstraction, is the semantics of the sent data. Both the server and the client have to have exactly the same understanding of bytes order. This includes understanding of which bytes represent event flags and which represent the real data. We also have to distinguish between constant and variable data length for each event.

From the architecture point of view, the sent data semantics is not directly mapped into java API, but it is rather wrapped in a higher-level and object-oriented way. One of the examples is the `VMEventListener.vmStarted(VMDescription)` event method. Even if the socket client will receive multiple numeric values and byte arrays, the user of the API will be working with objects on the proper level of abstraction.

Very crucial part of the sent data semantics is the synchronization between the C and Java side. During the development communication protocol was changed every time I have added new event. In order to guarantee the synchronization, there is a Unit test part of the build that will test the proper C and Java synchronization every time the build will run.

One aspect every client implementation has to respect is the byte order from operating system point of view. Here we are talking about "Endianness" [23]. The server as well as the client both need to transform the OS specific endianness to the Big-endian (the standard for network communication).

Now let us look at the java implementation of the socket communication protocol:

```
package org.crazyjavahacking.gcspy.socket.api;


/**
 * Socket communication protocol.
 *
 * @author Martin Skurla (crazyjavahacking@gmail.com)
 */
public enum SocketCommunicationProtocol {
    //---------------------------------------------------------------
    // Constants.
    //---------------------------------------------------------------
    VM_STARTED_EVENT ((byte) 1),
    // followed by:
    // 1.) 1 byte  (host JVM architecture bits)
    // 2.) 2 bytes (length of VM info) + x bytes (content)
    // 3.) 2 bytes (length of VM name) + y bytes (content)
    // 4.) 2 bytes (length of VM vendor) + z bytes (content)
    // 5.) 2 bytes (length of VM version) + w bytes (content)

    VM_INITIALIZED_EVENT((byte) 2), // no data follow
    VM_DEAD_EVENT       ((byte) 3), // no data follow

    GC_STARTED_EVENT ((byte) 4), // no data follow
    GC_FINISHED_EVENT((byte) 5), // no data follow
    // followed by 4+4 bytes (object address & size)
    _32_BIT_OBJECT_ALLOCATED_EVENT((byte) 6),
    // followed by 4 bytes (object address)
    _32_BIT_OBJECT_DEALLOCATED_EVENT ((byte) 7),
    // followed by 4+4 bytes (old and new object address)
    _32_BIT_OBJECT_MOVED_IN_MEMORY_EVENT((byte) 8),

    // followed by 8+4 bytes (object address & size)
    _64_BIT_OBJECT_ALLOCATED_EVENT((byte)  9),
    // followed by 8 bytes (object address)
    _64_BIT_OBJECT_DEALLOCATED_EVENT((byte) 10),
    // followed by 8+8 bytes (old and object address)
    _64_BIT_OBJECT_MOVED_IN_MEMORY_EVENT((byte) 11),

    CLIENT_CONNECTED_EVENT((byte) 101),    // no data follow
    CLIENT_DISCONNECTED_EVENT((byte) 102); // no data follow
    ...
}
```

Fig. 11 Java implementation of the socket communication protocol

4.4.2 Connection management

Another important thing is the connection management. Here we are not talking about the creation of connections where both client and server are responsible for creating their connections respectively. Rather, we are talking about closing the connection. Connection closing is transparently handled by the server and client should never close the socket. This is not only more transparent for the user, but the client implementation is easier as well. Anyway, the server should manage (and close) the connections at the first place.

4.4.3 Thread model

Because socket communication is by definition not thread-safe, it does not include any kind of build-in synchronization. Because the function *callbacks* will be called concurrently, in the case when multiple threads will allocate the objects or parallel garbage collection algorithm will be selected, manual synchronization is needed. Handling concurrency in the C language is very low-level and due to the fact that it will be executed in the JVM, nothing else than *monitors* (as a model of concurrency) can be used. For simplification, I decided to use just one global monitor for all threads. This will quite significantly decrease the overall performance on one side, but this is the easiest way how to deal with concurrent event notification.

Improving performance is always an important step. In this case, more important than looking for proper monitors is to find events that will not need any kind of synchronization at all. If we could send event object notifications, which were fired from the same thread in the same but independent communication channel, the synchronization will not be necessary anymore. Basically, one communication channel per one thread will increase the performance significantly (for concurrent applications).

For this aim we will need to map an object to the thread from which it was created and the thread to unique communication channel. The only way how you can uniquely identify the object is by its native memory address. The native object memory address could however be changed during the JVM lifecycle. In order not to introduce race conditions, an internal synchronization will be needed and it will decrease the performance.

Even more importantly, especially for applications with huge number of threads (for example application servers), this could be not possible at all. You are allowed to create just a limited number of socket connections because of limited number of ports, but you can theoretically create much higher number of threads.

To sum up, more performing solution will be not usable in every situation and the real question will be whether the performance will be increased significantly.

4.4.4 Blocking states

One important note about blocking states is the fact that blocking states have nothing to do with concurrency.

The contract of blocking states is important for additional implementations of the client and server side in order to be able to interchange implementations.

In practice, JVM will block its execution during the `STARTED` state until a client will connect to the server socket. This blocking state will eliminate the need to have code that would need to handle events until the client would connect and will also eliminate needed concurrency synchronization. As a side-effect, memory will not be pointlessly occupied by event data.

The second blocking state will occur when server will block the observed JVM execution at the end of the server lifecycle until the client will acknowledge it. The reason is the fact that all events did not have to be already processed when the JVM reaches its `DEAD` state. And because the server socket will be closed right after the client notification event will be received by the server, no additional data will be readable from server socket after the client notification.

4.5 Portability

Let us start with a provocative idea from Jim McCarthy, which he wrote as part of his article *21 Rules of Thumb – How Microsoft develops its Software* [24]:

"Portability is for canoes."

Even though it may look like portability is not the case for Java applications because everybody is familiar with the famous motto:

"Write once, run everywhere."

our application is a little bit special and the portability plays an important role from development, testing and architecture point of view.

Portability in the GCSpy case includes three parts:

- operating system portability,
- virtual machine portability,
- Java portability.

4.5.1 Operating system portability

Operating system portability means portability across various operating systems. This is very important because the code written in the C language will be compiled, linked into shared library for build purposes and linked into executable for testing purposes.

Operating system portability is the most complicated portability we will talk about. If you would like to add support for new operating system, you would need to do changes across various parts of the system, most likely in all of them:

- native level,
- application level,
- build infrastructure level,
- runtime infrastructure level.

Now let us talk about all of them in more details. For every kind of the above mentioned levels there will be the description of the root and also the purpose, how it is handled in code, code samples how the code looks like and also how would you recognize that some parts of the system are not implemented for your running operating system.

4.5.1.1 Native level

Portability on the native level (C code) makes the most sense.

Because of lack of functions inside the core C language libraries, I was forced to use some operating system specific code. This includes:

- native server socket implementation,
- conversion of running platform endianness into network endianness,
- conversion of string (`char*`) into 64bit unsigned long number.

Native server socket implementation is crucial because without the data we cannot visualize anything. Sending data also means the conversion from running platform endianness into network endianness which is Big-endian. For testing purposes I also needed a C function that would do the conversion from given C-based string into 64bit long unsigned number. Because the size of the numeric types in the C programming language depends on the running platform, it will be operating system specific as well.

All mentioned examples use conditional compilation (handled by preprocessor) to separate operating system specific functionality. As an example, let us look how the string conversion into 64bit long number looks like for Windows host operating system:

```
#ifdef _WIN32
    testNumber = _strtoui64(argv[3 + count], NULL, 10);
#else
    fprintf(stderr, "!!! Conversion from string to 64bit
              number is not implemented on running platform !");
    exit(-1);
#endif
```

Fig. 12 Example of operating system specific code on native level

As you can see from Figure 12 (and it is also a general principle), operating system specific code is always placed in conditionally executed code blocks guarded by preprocessor. If you will try to run the binaries on different platform (even if it makes sense only in some combinations of Solaris/Unix/BSD/Linux operating systems) you will get an error at runtime. However, because the native code is executed as part of tests that are part of the build, at the end of the day, this will cause the compilation to fail.

4.5.1.2 Application level

Portability on the application level (Java code) is important because of core libraries and abstractions I made on top of them.

There were multiple situations when I needed to implement operating system specific behavior in the Java code:

- compiler and linker support,
- ability to execute executable file,
- operating system specific file extensions.

Compiler and linker support will naturally differ on different operating systems, not only because of various compiler and linker pairs (MSVC, GCC, ...) but also because different kinds of initializations could be needed for different compilers (MSVC needs to set a lot of system properties to work properly). The next example is the way how to run an executable file. Also operating system file extensions will be different for various operating systems (for example a shared library will have `*.dll` extension on Windows, but `*.so` on the Solaris operating system).

In such cases the solution is treated on the API level. As an example, let us look how the default compiler support on Windows host operating system works:

```
public static CompilerSupport getDefault() {
    if (OSUtilities.isHostWindows()) {
        return new MicrosoftVisualCCompilerSupport();
    }

    throw new UnsupportedOperationException(
                    "Host operating system is not supported !");
}
```

Fig. 13 Example of operating system specific code on application level

As you can see from Figure 13, the error case is handled by throwing a runtime exception. Compiling MOJO, linking MOJO, groovy script and tests are all using such kind of operating system specific code. Because all mentioned occurrences are part of tests that are part of the build, error cases will cause the build to fail.

4.5.1.3 Build infrastructure level

Portability on the build infrastructure level (Maven) may look like a little bit weird, but because of native code compilation it makes sense.

There are two examples of such code:

- compiler flags,
- linker flags.

It should be obvious that different operating systems will use different compilers, and different compilers will almost certainly require different sets of compiler and linker arguments. As a result setting compiler and linker commands is really an operating system specific task.

In such cases, Maven profiles are the solution. Let us see how you can set operating system specific compiler and linker flags using Maven profile.

What may be not that straightforward from Figure 14, is the fact that if you will try to build the application on platform that is not supported, the profile will not be activated. Because of that, Maven will not be able to recognize the "`shared-library`" packaging and thus the compilation will fail.

```
<profile>
    <activation>
        <os>
            <family>windows</family>
        </os>
    </activation>
    <build>
        <plugins>
            <plugin>
               <groupId>org.crazyjavahacking.gcspy</groupId>
                <artifactId>
                    org-crazyjavahacking-gcspy-shared-
                                              library-packaging
                </artifactId>
                <version>1.0</version>
                <extensions>true</extensions>
                <configuration>
                    <compilerOptions>
                        <compilerOption>...</compilerOption>
                    </compilerOptions>
                    <linkerOptions>
                        <linkerOption>...</linkerOption>
                    </linkerOptions>
                </configuration>
            </plugin>
        </plugins>
    </build>
</profile>
```

Fig. 14 Example of operating system specific configuration on build infrastructure level

4.5.1.4 Runtime infrastructure level

Portability on the runtime infrastructure level (Groovy) may look like a little bit weird again. Portability on the runtime infrastructure level is needed because of the multi-platform groovy script.

There is one specific example of the operating system specific handling in the groovy script. Groovy script uses shared library file extension, which will differ across various operating systems. That portability handling is already done in the java code, but because the groovy script initiated the java call, the exception will be propagated back to groovy.

4.5.1.5 Summary

Now let us summarize all the above mentioned kinds of portability in the following table:

Tab. 5 GCSpy operating system portability handling

Level	**Handled by**	**Handling language**	**Original error handling**	**Propagated error handling**
native	conditional compilation	C	runtime error	compilation failed
application	Java API	Java	runtime exception	compilation failed
build infrastructure	Maven profiles	XML	compilation failed	compilation failed
runtime infrastructure	Groovy script	Groovy	runtime error	compilation failed

4.5.2 Virtual machine portability

Virtual machine portability means portability across various virtual machines. The overall implementation is tight to JVMs and HotSpot specifically.

You are able to add support for additional JVMs if the required JVM supports following technologies:

- JNI,
- JVMTI,
- Serviceability agent.

Currently there are 2 code pieces of the application that are VM specific:

- recognition of the CPU type (number of bits) the underlying JVM is running on,
- getting the native memory heap address from object.

Recognition of the CPU type on which the underlying JVM is running on will be JVM specific and may need the JVM to be fully initialized. In the HotSpot case it is done by reading a HotSpot-specific system property. That situation is handled by Java API with appropriate exception handling on error cases.

Getting native memory address from object is handled in native C code and error cases are handled by conditional compilation.

4.5.3 Java portability

Last but not least, we need to talk about Java portability a little bit. The minimal Java requirement for running and building the application is Java 7.

There was one very good reason to select Java 7 and not older version of Java as a minimal requirement. Serviceability agent functionalities, which are one of the core parts of the application, were not part of the JDK on Windows platforms until JDK 6u30. Because I started the development earlier than Java 6u30 was released, for development purposes I had to use Java 7 naturally.

As soon as Java 7 was selected as the minimal required Java version, I started taking advantage of most of the new Java 7 language features:

- multi-catch,
- diamond operator,

- try-with-resources,
- binary literals,
- underscores in numeric literals.

I am also taking advantage of NIO2, mainly for the purpose of file system file and directory watch notification. In that case I am using `WatchService` Java 7 class.

If you would like to support older version of Java, first you need to be sure that *Serviceability agent* is available on that platform. The porting process will include rewriting all the code that uses above-mentioned newly added Java language features and rewriting the native file system watch notification.

4.6 Comparison with original GCSpy

Finally let us summarize the most important differences between the original GCSpy and the proposed GCSpy projects in the following table:

Tab. 6 Original and proposed GCSpy comparison

	Original GCSpy	**Proposed GCSpy**
API	not really exposed API	high level object-oriented Java API
architecture	strictly client-server	modular
data gathering strategy	strictly using socket	using GCSpy Socket Data Provider Java API (with build-in socket implementation)
data transmission strategy	periodically the whole heap	event-based
portability	specific code for every VM and GC algorithm is needed	across every JVM with JNI, JVMTI and SA[44]
summary data	gathered by server and transmitted separately	gathered by client
supported systems	already running, local or remote	"not fully initialized" local or remote
visualizing systems	software systems with components and partitions	software systems with at least single GCSpy Data Provider API implementation

Comparing API is easy because the original GCSpy does not really expose public API. Proposed solution is exposing well-defined high level object-oriented GCSpy Socket Data Provider API.

[44] Serviceability Agent

Original GCSpy was strictly using client-server architecture. That kind of architecture introduced leaking abstraction when the need to read stored traces needed to start the server. Proposed solution uses modular architecture design.

Data gathering strategy is another very different aspect of both systems. Original GCSpy was strictly using sockets. Proposed solution can use any GCSpy Data Provider API implementation. Current support includes socket implementation, but there many other technologies (RMI[45], CORBA[46] and Google Protocol Buffers) that can be easily incorporated.

Data transmission strategy is handled very differently at the fundamental level. Original GCSpy was transmitting the data about the whole heap periodically. Proposed solution is using event-based strategy.

The big disadvantage of original solution is the portability. Original GCSpy needed code specific to every VM (or more general any software system being observed) and specific to every garbage collector algorithm within that system. Proposed solution will work across every JVM that supports JNI, JVMTI and SA.

Because of different data transmission strategy, handling summary data is different as well. Summary data (e.g. number of live objects per GCSpy space) in the case of the original GCSpy had to be gathered by server and transmitted separately. Proposed solution is not posting any kind of summary data. Because the data sent to the client are fine-grained, the client can count any required summary data by himself.

Original GCSpy supported already running, local or remote systems. Proposed solution supports "not fully initialized" local or remote systems. The term "not fully initialized" is described in the Chapter 4.4.4 Blocking states.

Last but not least there is also a difference between the kinds of systems both solutions can visualize. Original GCSpy can visualize a software system that can be modeled using components and partitions. Proposed solution can visualize any software system with at least single GCSpy Data Provider API implementation supported by that system.

[45] Remote Method Invocation
[46] Common Object Request Broker Architecture

5 Conclusion

Garbage collection turns out to be one of the most important research areas around virtual machines over the last decades. Garbage collection visualization is not only important for garbage collection researchers and implementers, but also for anybody who would like to understand the memory management characteristics of one's application better.

The aim of this thesis was to analyze, design and implement a garbage collection visualization tool built on top of the original GCSpy project foundations. This goal has been met in full and the created tool solves a lot of problems.

Together with the tool, the custom build, runtime and test infrastructures were developed. The advantages of the tool include the ease of use, modular design, reusable components, higher-level object-oriented Java API, non-intrusiveness, preciseness of visualization, transparency and virtual machine independence.

The tool will cause inevitable performance penalty for the observed application because of bytecode instrumentation, thread model, data gathering and data transmission. The tool will be less scalable in comparison with original GCSpy because of different data transmission strategy.

The opportunities to enhance the tool functionalities include the addition of additional visualizations and the support for additional operating systems.

Bibliography

[1] Vincent Massol, Jason van Zyl, Brett Porter, John Casey, and Carlos Sanchez, *Better Builds with Maven*. United Stated of America: Exist Global, 2008.

[2] Maven project home page. [Online].
http://maven.apache.org/

[3] Brad Appleton, "Patterns and Software: Essential Concepts and Terminology," 1998.

[4] Edd Dumbill. (2005, August) Ruby on Rails: An Interview with David Heinemeier Hansson. [Online].
http://www.oreillynet.com/pub/a/network/2005/08/30/ruby-rails-david-heinemeier-hansson.html

[5] Introduction to the Build Lifecycle. [Online].
http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

[6] Java Virtual Machine Tool Interface home page. [Online].
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

[7] Creating a Debugging and Profiling Agent with JVMTI. [Online].
http://java.sun.com/developer/technicalArticles/Programming/jvmti/

[8] Java Platform Debugger Architecture home page. [Online].
http://java.sun.com/javase/technologies/core/toolsapis/jpda/index.jsp

[9] The JVMPI Transition to JVMTI. [Online].
http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition/

[10] Jürgen Petri, *NetBeans Platform 6.9 Developer's Guide*. Birmingham: Packt Publishing, 2010.

[11] Heiko Böck, *The Definite Guide to NetBeans Platform*. United States of America: Apress, 2009.

[12] Heiko Böck et al., *DZone Refcard: Getting Started with NetBeans Platform 7.0*. Cary, North Carolina: DZone.

[13] Richard Jones, Antony Hosking, and Eliot Moss, *The Garbage Collection Handbook*. Great Britain: CRC Press, 2012.

[14] John McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," p. 34, April 1960.

[15] Tony Printezis. Garbage Collection in the Java HotSpot Virtual Machine. [Online]. http://www.devx.com/Java/Article/21977

[16] Sun Microsystems, *Memory Management in the Java HotSpot Virtual Machine*., 2006.

[17] The Garbage-First Garbage Collector. [Online]. http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html

[18] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The Continuously Concurrent Compacting Collector. [Online]. http://www.azulsystems.com/products/zing/c4-java-garbage-collector-wp

[19] GCSpy home page. [Online]. http://labs.oracle.com/projects/gcspy/

[20] Tony Printezis and Richard Jones, "GCSpy: An Adaptable Heap Visualisation Framework," 2002.

[21] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley, *The Java Language Specification: Java SE 7 Edition*. California, United States of America, 2012.

[22] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley, *The Java Virtual Machine Specification: Java SE 7 Edition*. California, United States of America, 2012.

[23] Endianness. [Online]. http://en.wikipedia.org/wiki/Endianness

[24] David Gristwood. 21 Rules of Thumb – How Microsoft develops its Software. [Online]. http://blogs.msdn.com/b/david_gristwood/archive/2004/06/24/164849.aspx

Appendices