Bachelorthesis

Konzeption und Entwicklung eines Auswertungstools für die Log Files des JRockit Garbage Collectors

von

Raffael Schmid

Schule:	Hochschule für Technik, Zürich
Betreuer:	Mathias Bachmann
Experte:	Marco Schaad
Zeitraum:	22. Juni 2011 - 22. Dezember 2011

Zusammenfassung

Inhaltsverzeichnis

Einleitung

TODO

Teil I

Projektbeschreibung

Kapitel 1

Aufgabenstellung

1.1 Ausgangslage

Für die Ermittlung von Java Performance-Problemen braucht es Wissen über die Funktionsweise der Java Virtual Machine (JVM), deren Ressourcenverwaltung (Speicher, I/O, CPU) und das Betriebssystem. Die Verwendung von Tools zur automatisierten Auswertung der Daten kann in den meisten Fällen sehr hilfreich sein. Die Auswertung von Garbage Collection Metriken kann im laufenden Betrieb durch Profiling (online) gemacht werden, sie ist aber bei allen JVMs auch via Log-Datei (offline) möglich. Die unterschiedlichen Charakteristiken der Garbage Collectors bedingen auch unterschiedliche Auswertungs- und Einstellungsparameter. JRockit ist die Virtual Machine des Weblogic Application Servers und basiert entsprechend auch auf anderen Garbage Collector Algorithmen als die der Sun VM. Aktuell gibt es noch kein Tool, welches die Daten der Logs sammelt und grafisch darstellt.

1.2 Ziele der Arbeit

Ziel der Bachelorthesis ist die Konzeption und Entwicklung eines Prototypen für die Analyse von Garbage Collection Log Dateien der JRockit Virtual Machine. Die Software wird mittels einer Java Rich Client Technologie implementiert. Zur Konzeption werden die theoretischen Grundlagen der Garbage Collection im Allgemeinen und der JRockit Virtual Machine spezifisch erarbeitet und zusammengestellt.

1.3 Aufgabenstellung

Im Rahmen der Bachelorthesis werden vom Studenten folgende Aufgaben durchgeführt:

1. Studie der Theoretischen Grundlage im Bereich der Garbage Collection (generell und spezifisch JRockit Virtual Machine)
2. Stärken- / Schwächen-Analyse der bestehende Rich Client Frameworks (Eclipse RCP Version 3/4, Netbeans)
3. Durchführung einer Anforderungsanalyse für einen Software-Prototyp.
4. Auswahl der zu verwendenden Frameworks
5. Konzeption und Spezifikation des Software-Prototypen (auf Basis des ausgewählten Rich Client Frameworks), der die ermittelten Anforderungen erfüllt.
6. Implementation der Software
7. Bewertung der Software auf Basis der Anforderungen

1.4 Erwartete Resultate

Die erwarteten Resultate dieser Bachelorthesis sind:

1. Detaillierte Beschreibung der Garbage Collection Algorithmen der Java Virtual Machine im Generellen und spezifisch der JRockit Virtual Machine.
2. Analyse über Stärken und Schwächen der bestehenden (state of the art) Java Rich Client Technologien
3. Anforderungsanalyse des Software Prototyps
4. Dokumentierte Auswahlkriterien und Entscheidungsgrundlagen
5. Konzept und Spezifikation der Software
6. Lauffähige, installierbare Software und Source-Code
7. Dokumentierte Bewertung der Implementation

1.5 Geplante Termine

Kick-Off:	Juni 2011
Design Review:	August 2011
Abschluss-Präsentation:	Ende November 2011

Teil II

Umsetzung

Kapitel 2

Analyse der Aufgabenstellung

2.1 Übersicht

Wie in der Aufgabenstellung definiert, ist das Ziel dieser Arbeit die Konzeption und Implementation einer Analysesoftware für die Garbage Collection Log Dateien der JRockit Virtual Machine. Grundlegendes Ziel einer solchen Software ist es, die anfallende grosse Menge an Daten übersichtlich darzustellen und verschiedene Sichten auf diese Daten zu ermöglichen. Um die Anforderungen an diese Software zu ermitteln, ist ein die Studie der Grundlagen der Garbage Collection und im spezifischen die der JRockit Virtual Machine notwendig. Anschliessend wird eine Stärken-/Schwächen-Analyse der bestehenden Richt-Client Frameworks gemacht, aufgrund welcher nach der Aufnahme der Anforderungen der Entscheid für das zu verwendende Framework getroffen werden kann. Im Anschluss an diese Tätigkeiten folgt die Konzeption und Implementation der Software. Die genauere Untersuchung der einzelnen Teilaufgaben wird in den folgenden Abschnitten gemacht:

2.1.1 Erarbeitung der Grundlagen Bereich der Garbage Collection

Die Erarbeitung der Grundlagen dient als Basis in zwei Bereichen:

- **Anforderungsanalyse:** Aus Sicht des Analysten respektive dem Benutzer dieser Software ist es essentiell, dass er die richtigen Daten der Garbage Collection, die richtige Sicht auf diese Daten und die richtigen Filter-Funktionen vorfindet. Voraussetzung für diese Anforderungen sind die Kenntnisse der verschiedenen Garbage Collection Algorithmen im Generellen und spezifisch die der JRockit Virtual Machine.
- **Konzept:** Die Konzeption des Domänen-Modells und des Einlese-Prozesses

der Log Dateien bedingt eine genaue Kenntnis der verschiedenen Strategien und Formaten der Ausgaben.

2.1.2 Stärken- / Schwächen-Analyse der bestehenden Rich Client Frameworks

Aus unterschiedlichen Gründen ist es sinnvoll, die Applikation als einen Rich Client zu implementieren. Als Basis kommen entweder die Netbeans- oder die Eclipse-Plattform in Frage. Die Stärken- und Schwächen-Analyse soll den Entscheid für die eine dieser Plattform begründen.

2.1.3 Durchführung einer Anforderungsanalyse für den Software-Prototypen

Die Anforderungen werden zusammen mit einem Performance-Analysten ermittelt und nach [7, 4.3.2 Angepasste Standardinhalte] dokumentiert. Laut dem IEEE[1] und [7, 4.5 Qualitätskriterien für das Anforderungsdokument] müssen Anforderungen folgende Kriterien erfüllen:

- Eindeutigkeit und Konsistenz
- Klare Struktur
- Modifizierbarkeit und Erweiterbarkeit
- Vollständigkeit
- Verfolgbarkeit

2.1.4 Auswahl der zu verwendenden Frameworks

Basierend auf der Stärken-/Schwächen-Analyse wird ein Entscheid für das jeweilige Framework gewählt. Nebst der Auswahl der Rich Client Plattform muss auch ein Entscheid hinsichtlich Charting-Bibliothek gefällt werden.

2.1.5 Konzeption und Implementation

Basierend auf den Anforderungen wird das Konzept erstellt und anschliessend die Software implementiert.

2.1.6 Bewertung

Die Bewertung der Software wird auf Basis der Anforderungen gemacht.

[1] Institute of Electrical and Electronic Engineers

Kapitel 3

Grundlagen Garbage Collection

Schon seit den ersten Programmiersprachen ist das Aufräumen von verwendeten Ressourcen / Speicher ein wichtiges Thema. Im Unterschied zu den ersten Sprachen, bei denen das Memory Management in der Verantwortung des Entwicklers war (explizit), findet allerdings das Recycling von Memory bei Sprachen der dritter Generation automatisch statt und macht Operatoren wie "free" unwichtig. Bei Formen dieser automatischen Speicherverwaltung spricht man von Garbage Collection[1]. In den meisten neueren Laufzeitumgebungen spricht man zusätzlich von adaptivem Memory Management was bedeutet, dass Feedback der Laufzeitumgebung zur Anpassung der Garbage Collection Strategie verwendet wird. Probleme die nur beim expliziten Memory Management auftreten sind Dangling References/Pointers[2] und Space Leaks[3]. Trotzdem sind Memory Leaks auch bei automatischer Speicherverwaltung noch möglich, nämlich dann wenn Memory noch referenziert wird, auch wenn es schon nicht mehr gebraucht wird. Der folgende Abschnitt beschreibt die Grundlagen der Java Garbage Collection und geht im zweiten Teil auf die spezifischen Eigenheiten der JRockit Virtual Machine ein.

3.1 Funktionsweise

Alle Techniken der Garbage Collection zielen darauf ab, die "lebenden" von den "toten" Objekten im Speicher zu unterscheiden. Sprich, es müssen die Objekte gefunden werden, welche innerhalb der Software oder des Systems nicht mehr referenziert werden. Die aktuellen Strategien lassen sich laut[1, S. 77] in zwei

[1] vom englischen Wort "garbage collector" für Müll-, Abfallsammler

[2] Man spricht von Dangling Pointers oder Dangling References, wenn ein Pointer auf ein Objekt im Memory freigegeben wurde, obwohl es noch gebraucht wird.

[3] Man spricht von Space Leaks, wenn Memory alloziert und nicht mehr freigegeben wurde, obwohl es nicht mehr gebraucht wird.[5]

unterschiedliche Familien aufteilen: “Reference Counting” und “Tracing techniques”.

3.1.1 Reference counting[1, S. 77]

Beim Reference counting behält die Laufzeitumgebung jederzeit den Überblick, wie viele Referenzen auf jedes Objekt zeigen. Sobald die Anzahl dieser Referenzen auf 0 gesunken ist, wird das Objekt für die Garbage Collection freigegeben. Trotzdem der Algorithmus relativ effizient ist, wird er aufgrund der folgenden Nachteile nicht mehr verwendet:

- Sofern zwei Objekte einander referenzieren (zyklische Referenz), wird die Anzahl Referenzen nie 0.
- Es ist relativ aufwendig, die Anzahl Referenzen immer auf dem aktuellsten Stand zu halten, insbesondere in nebenläufigen Systemen.

3.1.2 Tracing techniques[1, S. 77]

Bei den Tracing Techniken werden von vor jeder Garbage Collection die Objekte gesucht, auf welche aktuell noch eine Referenz zeigt. Die anderen werden zur Garbage Collection freigegeben. Diese Art von Garbage Collection Algorithmen verwenden ein Set von Objekten, bestehend aus den Stacks und Registern der aktuellen Threads und globalen Daten wie “static final” variablen, als Startpunkt für die zu markierenden Objektgraphen.

3.2 Ziele der Garbage Collection[5, S. 4]

Garbage Collectors unterliegen grundsätzlich folgenden zwei Bedingungen:

- **Sicherheit:** Garbage Collectors dürfen nur Speicher/Objekte freigeben, der effektiv nicht mehr gebraucht wird,
- **Umfassend:** Garbage Collectors müssen Speicher/Objekte die nicht mehr gebraucht werden nach wenigen Garbage Collection Zyklen freigegeben haben.

Wünschenswert für Garbage Collection Algorithmen sind folgende Punkte:

- **Effizienz:** Die Anwendung soll vom laufenden Garbage Collector möglichst wenig mitkriegen: keine langen Pausen[4], möglichst wenig verwendete Ressourcen[5]
- **Fragmentierung:** Zwecks schneller Allokation von Speicher sollte der Speicher möglichst wenig

[4] man spricht von Stop-the-World wenn zwecks Garbage Collection die Anwendung gestoppt wird und ihr damit keine Ressourcen zur Verfügung stehen

[5] Ressourcen der CPU sollen der Anwendung zur Verfügung gestellt werden und nicht für Garbage Collection verwendet werden.

3.3 Eingliederung von Garbage Collection Algorithmen[5, S. 5]

Bei der Selektion von Garbage Collection Algorithmen gibt es verschiedene entscheidende Kriterien:

3.3.1 Serielle versus Parallele Collection

Eine Multi-Core Maschine ist eine mit mindestens 2 Prozessorkernen. Sofern ein paralleler Algorithmus verwendet wird, besteht auf diesen auch die Möglichkeit, auch die Garbage Collection zu parallelisieren. Meistens bringt dies zwar einen kleinen Overhead mit sich, wirkt sich aber trotzdem in einer Verkürzung der Garbage Collection Zeit aus.

3.3.2 Konkurrierend versus Stop-the-World

Wenn aufgrund der Garbage Collection der Heap der Laufzeitumgebung blockiert (freeze) werden muss, führt das implizit zum Stopp (Stop-the-World) der Anwendung.

3.3.3 Kompaktierend, Kopierend

Fragmentierung ist ein eigentlich nicht wünschenswertes Resultat der Garbage Collection. Sie tritt dann auf, wen Algorithmen verwendet werden die den Heap im Anschluss an die Speicherfreigabe weder kompaktieren noch die lebenden Objekte in einen anderen Bereich aneinanderliegend kopieren.

3.4 Grundlegende Algorithmen

Der Prozess der Garbage Collection beginnt bei allen Algorithmen mit der Marking-Phase. Für jedes Objekt des Root Sets[6] werden rekursiv die transitiv abhängigen Objekte auf dem Heap bestimmt. Das hat zum Ziel, dass man aufgrund des Resultats die nicht mehr referenzierten (löschbaren) Objekte herausfinden kann.

3.4.1 Mark & Sweep Algorithmus[4]

Beim Mark & Sweep Algorithmus wird nach der oben beschriebenen Mark-Phase der Speicher der nicht mehr referenzierten Objekte freigegeben. In den meisten Implementationen bedeutet dies das einfügen dieses Objekts in eine sogenannte Free-List.

[6] Objekte im Heap die aus den Call-Stacks der aktuellen Threads referenziert werden, globale ("static final" definierte) Variablen mit Referenzen auf den Heap)

Nachteil dieses Algorithmus

Der zwar simple Mark & Seep Algorithmus hat den Nachteil, dass eine Freigabe dieser Speicherlöcher zu einer Fragmentierung des Speichers führt. Ein fragmentierter Speicher ist deshalb nicht wünschenswert, weil es für den Allokator immer schwieriger wird, die Speicherlöcher zu füllen. In einem solchen Fall kann es bis zu einer OutOfMemoryException führen, wenn zwar insgesamt genügend Memory frei ist, allerdings nicht genügend am Stück.

3.4.2 Mark & Copy Algorithmus[4]

Ein alternativer Algorithmus bei dem der Heap nach der Garbage Collection nicht fragmentiert ist, ist der Mark & Copy Algorithmus. Bei diesem wird der Heap in einen "from space" und einen "to space" unterteilt. Objekte die noch immer am Leben sind, werden im Anschluss an die Markierung vom "from space" in den "to space" kopiert. Die Allokation von Speicher ist im Anschluss mit grosser Effizienz möglich, weil der Heap nach der Garbage Collection nicht fragmentiert ist.

3.4.3 Mark & Compact Algorithmus[3]

In gewissen Situationen respektive für die unterschiedliche Lebensdauer der Objekte (siehe Generational Garbage Collection) macht es Sinn, unterschiedliche Algorithmen einzusetzen: Mark & Compact ist ein Algorithmus, bei welchem nach der Markierungs-Phase und der Elimination der "toten" Objekte eine Kompaktierung des Speichers stattfindet. Das hat ebenfalls den Vorteil, dass im Gegensatz zu Mark & Copy nicht der doppelte Speicher benötigt wird, ist allerdings relativ Ressourcen-intensiv.

3.4.4 Tri-Coloring Mark and Sweep

Eine Variante des Mark & Sweep Algorithmus die sich besser parallelisieren lässt ist der Tri-Coloring Mark & Sweep Algorithmus. [1, S. 79]. Im Gegensatz zur normalen Version des Mark & Sweep Algorithmus wird anstelle eines Mark-Flags ein ternärer Wert genommen der den Wert "weiss", "grau" und "schwarz" annehmen kann. Der Status der Objekte wird in drei Sets nachgeführt. Das Ziel des Algorithmus ist es, alle weissen Objekte zu finden. Schwarze Objekte sind die, die garantiert keine weisse Objekte referenzieren und die Grauen sind die, bei denen noch nicht bekannt ist, was sie referenzieren. Der Algorithmus funktioniert folgendermassen:

1. Als erstes haben alle Objekte das Flag weiss.
2. Die Objekte des Root-Sets werden grau markiert.
3. Solange es graue Objekte hat, werden rekursiv die Nachfolger dieser Objekte grau markiert.

4. Sobald alle Nachfolger des Objekts grau markiert sind, wird das aktuelle Objekt auf den Status schwarz geändert.

Das Prinzip des Tri-Coloring Mark &Sweep Algorithmus ist folgende Invariante: **Kein schwarzes Objekt zeigt jemals direkt auf ein Weisses.** Sobald es keine grauen Objekte mehr gibt - sprich das Set der grauen Objekte leer ist, können die weissen Objekte gelöscht werden.

3.5 Generational Garbage Collection

In der Regel gibt es innerhalb einer Anwendung unterschiedliche Altersgruppen von Objekten und wir können die Objekte in Short Living, Medium Living und Long Living Objekte kategorisieren. Es gibt viele Objekte die nicht lange leben, zum Beispiel Objekte welche die Lebensdauer einer Methode haben, und wenige Objekte die lange Leben, wie zum Beispiel Daten in einem Applikations-Level Cache. Aus diesem Grund ist es oft Sinnvoll, die Garbage Collection an die entsprechende Lebensdauer der Objekte anzupassen.

Je nach Implementation wird der Heap in unterschiedliche Speicherbereiche aufgeteilt. Bei der Oracle HotSpot Virtual Machine sieht dies folgendermassen aus[2]:

- **Young Collection:** Die Young Collection ist nochmals unterteilt in Eden Space (Bereich für neue Objekte) und zwei Survivor Spaces ("from-spaceäls aktiver Bereich, "to-spaceäls der Bereich in den die Objekte nach einer Collection kopiert werden)
- **Old Collection:** In der Old Collection befinden sich die Objekte, die eine gewisse Anzahl an Young-Generation Collections (Minor Collections) überlebt haben und dann in den Bereich der alten Objekte befördert (Promotion) wurden.
- **PermGen:** Der PermGen Bereich ist keine Generation, sondern ein Non-Heap-Bereich, und wird von der VM für eigene Zwecke verwendet. Dieser Bereich ist eine Eigenheit der Oracle HotSpot Virtual Machine. Hier werden beispielsweise Class-Objekte inklusive Bytecode für alle geladenen Klassen und JIT-Informationen[7]. gespeichert.

Die JRockti Virtual Machine unterteilt die Bereiche folgendermassen:

- **Nursery[8]:** Die Nursery entspricht der Young Collection der HotSpot Virtual Machine und ist der Bereich der jungen Objekte. Bei JRockit ist es möglich, eine zusätzliche Keep-Area innerhalb der Nursery zu haben. Diese Keep-Area ist der Platz der Objekte mittlerer Lebensdauer. Ein Objekt wird also zuerst von der Nursery in die Keep-Area promoted - was

[7] Die Just-in-Time (Kompilierung) führt zu einer Veränderung respektive Optimierung des Bytecodes.

[8] Nursery bedeutet im übertragenen Sinn Kindergarten

den Vorteil hat, dass es wenn nicht mehr referenziert immer noch einer Young Collection unterliegt - erst dann gelangt es nach einer nochmaligen Young-Garbage-Collection in die Old Collection.

- **Old Generation:** Die Old Generation beinhaltet wie bei der HotSpot Virtual Machine die Objekte mit langer Lebensdauer.

Für weitere Details zu den JRockit Garbage Collection Eigenheiten siehe Kapitel Garbage Collection in der Oracle JRockit Virtual Machine.

Kapitel 4

Garbage Collection in der Oracle JRockit Virtual Machine

4.1 Grundlage

Die Grundlage der JRockit Garbage Collection bildet der Tri-Coloring Mark & Sweep Algorithmus (siehe Abschnitt Tri-Coloring Mark and Sweep). Er wurde hinsichtlich besserer Parallelisierbarkeit und der optimalen Verwendung der Anzahl Threads optimiert. Die Garbage Collection der JRockit VM kann mit oder ohne Generationen arbeiten - so gibt es die beiden Algorithmen "Concurrent Mark & Sweep" und "Parallel Mark & Sweep" in beiden Ausführungen Generational und Single:

- Generational Concurrent Mark & Sweep
- Single Concurrent Mark & Sweep
- Generational Parallel Mark & Sweep
- Single Parallel Mark & Sweep

4.1.1 Old Collection

Im Unterschied zum normalen Tri-Coloring Algorithmus verwendet der Algorithmus der JRockit, egal ob parallel oder concurrent, zwei Sets für die Markierung der Objekte. In einem werden die grauen und schwarzen Objekte gespeichert, im anderen die weissen. Die Trennung zwischen Grau und Schwarz wird gemacht indem die grauen Objekte in Thread-Local Queues jedes Garbage Collection Threads gespeichert werden.

Die Verwendung von Thread-lokalem Speicher hat hinsichtlich den folgenden Punkten einen Vorteil:[1, S. 79]:

- Thread-lokaler Speicher führt zu einer besseren Parallelisierbarkeit.
- Thread-lokaler Speicher kann prefetched werden, was die Geschwindigkeit des Algorithmus als Ganzes erhöht.

Bei der Verwendung der Concurrent Algorithmen - bei welcher die Garbage Collection neben der Anwendung statt findet - werden logischerweise auch nebenbei von der Applikation neue Objekte erzeugt. Diese neuen Objekte werden in einem sogenannten Live-Set getrackt.

4.1.2 Intergenerationale Referenzen

Die Idee der Generational Garbage Collection ist, dass zumindest während einer Young Collection nur ein Teilbereich des Heaps aufgeräumt wird. Während der Mark-Phase einer Young Collection werden also Referenzen in die Old Collection nicht verfolgt. Problematisch bei diesem Vorgehen ist, dass es eventuell Referenzen aus der Old Generation in die Young Generation gibt, aufgrund welcher man die Objekte im Young Space nicht löschen darft. Dieses Szenario kann auf zwei unterschiedliche Wege passieren:

- **Ein Objekt wird durch eine Promotion von der Young Generation in die Old Generation verschoben.** Diesen Vorgang merkt sich der Garbage Collector in einem **Remembered Set**.
- **Es findet eine Zuweisung der Referenz durch die Applikation statt.** Diese Problematik wird durch Write Barriers und der Aufteilung des Heaps in sogenannte Cards[1] Bei der Änderung einer Variablen respektive einer Referenz die dann vom Tenured Space auf den Young Space zeigen soll, wird auf der Card in welchem diese Änderung statt findet das "Dirty Bit" gesetzt. Objekte innerhalb von dirty Cards werden nach der Markierungsphase nochmals überprüft.

Concurrent Mark & Sweep

Beim Concurrent Mark & Sweep handelt es sich eigentlich um einen Mostly Concurrent Mark & Sweep Algorithmus. Das heisst er findet nicht in allen Phasen konkurrierend zur Applikation statt. Die Markierung dieses Algorithmus ist in vier Phasen aufgeteilt:

- **Initial Marking (Nicht konkurrierend):** Hier wird das Root Set zusammengestellt.
- **Concurrent Marking (konkurrierend):** Mehrere Threads gehen nun diesen Referenzen nach und markieren Sie als lebende Objekte.
- **Preclean (konkurrierend): Änderungen im Heap während den vorherigen Schritten werden nachgeführt, markiert.**

[1] Eine Card ist typischerweise ein ungefähr 512 Byte grosser Bereich auf dem Heap.

- **Final Marking (Nicht konkurrierend):** Änderungen im Heap während der Precleaning Phase werden nachgeführt, markiert.

Die Sweep Phase findet ebenfalls konkurrierend zur Applikation statt. Im Gegensatz zur HotSpot VM findet es in zwei separaten Schritten statt. Als erstes wird die erste Hälfte des Heaps von toten Objekten befreit. Während dieser Phase können Threads Speicher in der zwieten Hälfte des Heaps allozieren. Nach einer kurzen Synchronisationspause findet das Sweeping auf dem zweiten Teil des Heaps statt, wieder gefolgt von einer Synchronisationspause.

Parallel Mark & Sweep

Beim parallelen Mark & Sweep findet die Garbage Collection parallel mit allen verfügbaren Prozessoren statt. Dazu werden aber alle Threads der Applikation während dieser Zeit gestoppt.

Kompaktierung

Aufgrund der im Speicher stattfindenden Fragmentierung wird bei jeder Old Collection eine Kompaktierung des Speichers durchgeführt. Zu Beginn der Sweep-Phase oder während werden dazu Objekte auf dem Heap umher kopiert, so dass danach wieder grössere Speicherbereiche zur Allokation frei stehen.

4.2 Übersicht der Garbage Collection Algorithmen auf der JRockit Virtual Machine

Die Entscheidung der Garbage Collection Strategie hängt von den Anforderungen der Applikation ab. Auf der JRockit VM gibt es drei verschiedene Modi:

- **Durchsatz (throughput):** Optimiert die Garbage Collection hinsichtlich möglichst grossem Durchsatz. Hier handelt es sich um Applikationen wie Batch-Systeme, etc. Um auf der Virtual Machine dieses Ziel zu erreichen, werden Parallele Algorithmen verwendet. Parallele Algorithmen laufen nicht-konkurrierend mit dem System in mehreren parallelen Threads. Das führt zwar zu kurzen Pausenzeiten für die Applikation, allerdings steht in der restlichen Zeit sehr viel Prozessor-Zeit zur Verfügung.
- **Pausenzeit (pause time):** Die Optimierung der Pausenzeit führt dazu, dass der Benutzer der Virtual Maschine keine langen Pausenzeiten bemerkt. Das ist insbesondere für Client-Server Applikationen notwendig. Wenn die JRockit Laufzeitumgebung auf möglichst kurze Pausenzeiten eingestellt ist, wird ein konkurrierender Garbage Collection Algorithmus verwendet, bei welchem es nur sehr wenige Stop-the-World Phasen gibt.
- **Determinismus (deterministic):** Optimiert den Garbage Collector auf sehr kurze und deterministische Garbage Collection Pausen.

Tabelle 4.1: Übersicht der Garbage Collection Algorithmen

Alias	**Aktivierung**	**Beschreibung**	**Pause**	**Durchs.**	**Heap**	**Mark**	**Sweep**
singlecon, singleconcon		Garbage Collection findet konkurrierend statt. Der Heap besteht nur aus einer Generation.	++	–	single	konk.	konk
gencon, genconcon	-Xgc:gencon -Xgc:genconcon	Neue Objekte kommen in die Nursery. Ist diese voll, werden die Lebenden in die Old Generation verschoben. Die Old Collection findet konkurrierend statt.	++	–	gen	konk.	konk.
singlepar, singleparpar		Wenn der Heap voll ist, wird bei gestoppter Applikation eine Garbage Collection des gesamten Heaps gemacht.	–	++	single	parallel	parallel
genpar, genparpar	-Xgc:genpar	Objekte werden in der Nursery angelegt. Ist diese voll, wird bei gestoppter Applikation parallel eine Young Collection durchgeführt. Bei vollem Heap gibs eine volle Garbage Collection.	–	++	gen	parallel	parallel
genconpar	-Xgc:genconpar		+	+	gen	konk.	parallel
genparcon	-Xgc:genparcon		+	+	gen	parallel	konk.
singleconpar	-Xgc:singleconpar		+	+	single	konk.	parallel
singleparcon	-Xgc:singleparcon		+	+	single	parallel	konk.

Tabelle 4.2: Übersicht der Garbage Collection Modi

Alias	**Aktivierung**	**Beschreibung**	Einstellungsmöglichkeiten
troughput	-Xgc:throughput	Der Garbage Collector wird auf maximalen Durchsatzder Applikation eingestellt. Er arbeitet so effektiv wie möglich und erhält entsprechend viele Java-Threads, was zu kurzen Pausen der Applikation führen kann.	
pausetime	-Xgc:pausetime	Der Garbage Collector wird auf möglichst kurze Pausen eingestellt. Das bedeutet, dass ein konkurrierender Algorithmus verwendet wird der insgesamt etwas mehr CPU Ressourcen benötigt.	-XpauseTarget=value (default 200msec)
deterministic	-Xgc:deterministic	Der Garbage Collector wird auf eine möglichst kurze und deterministische Pausenzeit eingestellt.	-XpauseTarget=value (default 30msec)

4.3 Garbage Collection Log Dateien

Die Auswertung der Garbage Collection findet auf Basis der Garbage Collection Log Dateien statt. Das Format dieser Log Dateien hängt mitunter auch von den aktivierten Log-Modulen der JRockit Virtual Machine ab. Der Aufbau der Log Dateien und die in der Analyse verwendeten Daten werden in diesem Abschnitt genauer beleuchtet.

4.3.1 Aktivierung Log Ausgaben

Defaultmässig macht die JRockit keine Angaben darüber, wie sie die Garbage Collection durchführt. Um an diese Informationen zu gelangen, muss das entsprechende Log-Module beim Start der Applikation aktiviert werden. Die Ausgaben werden dann auf die Standard Ausgabe geschrieben - können aber optional mit dem Argument -Xverboselog:logdatei.log in eine Datei umgeleitet werden. Das Format für die Kommandozeile sieht folgendermassen aus:

Listing 4.1: Format Aktivierung Log Modul

```
1 -Xverbose:<modul>[=log_level]
```

Um das Memory Log Modul (gibt Informationen über die Garbage Collection aus) zu aktivieren:

Listing 4.2: Garbage Collection Log (Info)

```
1 -Xverbose:memory
```

Die Umleitung der Ausgaben in eine separate Log-Datei kann mit dem Flag -Xverboselog:¡Datei-Namen¿ eingestellt werden:

Listing 4.3: Garbage Collection Log (Info) - Umleitung in gc.log

```
1 -Xverbose:memory -Xverboselog:gc.log
```

Zusätzlich können mit der Angabe des Log-Levels die Ausgaben verfeinert werden:

Listing 4.4: Garbage Collection Log (Debug) - Umleitung in gc.log

```
1 -Xverbose:memory=debug -Xverboselog:gc.log
```

Die ersten paar Zeilen der Ausgaben des Memory Moduls im Log Level “info” ohne explizite Angabe der Garbage Collection Strategie [2] sehen folgendermassen aus:

Listing 4.5: Garbage Collection Log (Debug) - Umleitung in gc.log

```
1  [INFO ][memory ] GC mode: Garbage collection optimized for throughput, strategy: Generational Parallel Mark
       & Sweep.
2  [INFO ][memory ] Heap size: 65536KB, maximal heap size: 1048576KB, nursery size: 32768KB.
3  [INFO ][memory ] <start>-<end>: <type> <before>KB-><after>KB (<heap>KB), <time> ms, sum of pauses <pause>
       ms.
4  [INFO ][memory ] <start>  - start time of collection (seconds since jvm start).
5  [INFO ][memory ] <type>   - OC (old collection) or YC (young collection).
6  [INFO ][memory ] <end>    - end time of collection (seconds since jvm start).
7  [INFO ][memory ] <before> - memory used by objects before collection (KB).
8  [INFO ][memory ] <after>  - memory used by objects after collection (KB).
9  [INFO ][memory ] <heap>   - size of heap after collection (KB).
10 [INFO ][memory ] <time>   - total time of collection (milliseconds).
11 [INFO ][memory ] <pause>  - total sum of pauses during collection (milliseconds).
12 [INFO ][memory ]            Run with -Xverbose:gcpause to see individual phases.
13 [INFO ][memory ] [OC#1] 0.843-0.845: OC 428KB->78423KB (117108KB), 0.003 s, sum of pauses 1.614 ms, longest
       pause 1.614 ms.
14 [INFO ][memory ] [OC#2] 1.393-1.442: OC 78449KB->156488KB (233624KB), 0.049 s, sum of pauses 47.104 ms,
       longest pause 47.104 ms.
15 [INFO ][memory ] [YC#1] 1.494-1.496: YC 156524KB->156628KB (233624KB), 0.002 s, sum of pauses 1.670 ms,
       longest pause 1.670 ms.
16 [INFO ][memory ] [YC#2] 1.496-1.496: YC 156652KB->156755KB (233624KB), 0.001 s, sum of pauses 0.605 ms,
       longest pause 0.605 ms.
17 [INFO ][memory ] [YC#3] 1.497-1.497: YC 156780KB->156884KB (233624KB), 0.001 s, sum of pauses 0.602 ms,
       longest pause 0.602 ms.
18 [INFO ][memory ] [YC#4] 1.497-1.498: YC 156908KB->157011KB (233624KB), 0.001 s, sum of pauses 0.592 ms,
       longest pause 0.592 ms.
```

[2] JRockit Virtual Machine verwendet damit die Default-Einstellung: Optimierung auf maximalen Durchsatz

4.3.2 Log Module

Die folgende Tabelle beschreibt die für die Auswertung der Garbage Collection relevanten Module alphabetisch sortiert.

Tabelle 4.3: Beschreibung der verschiedenen relevanten Log Modulen

Modul	**Beschreibung**	**Relevanz**
alloc	Informationen betreffend Speicher Allokation und "Out-of-Memory"	niedrig
compaction	zeigt abhängig vom Garbage Collection Algorithmus Informationen zur Kompaktierung	niedrig
gcheuristic	zeigt Informationen und Entscheidungen zur Garbage Collection Heuristik	mittel
gcpause	zeigt aktuell welche Pausen durch die Garbage Collection entstanden sind und wie lange sie gedauert haben.	mittel
gcreport	zeigt verschiedene Auswertungen (Anzahl Collections, Anzahl promotete Objekte, maximal promotete Objekte per Zyklus, totale Zeit, Durchschnittszeit, Maximale Zeit) der Garbage Collection zum aktuellen Lauf	niedrig
memory	zeigt Informationen zum Memory Management System wie Start-, Endzeitpunkt der Collection, Speicher bevor, nach Collection, Grösse des Heaps, etc.	**hoch**
memdbg	zeit Informationen die für das Debugging von Themen im Bereich des Arbeitsspeichers relevant sein können	**hoch**
systemgc	zeigt Garbage Collections die durch System.gc() gestartet wurden.	niedrig

Die folgenden Log Level können aktiviert werden: quiet, error, warn, info, debug, trace. Für mehr Informationen siehe [6].

4.3.3 Modul: memory

4.3.4 Modul: memdbg

Kapitel 5

Stärken- / Schwächen-Analyse Rich Client Frameworks

Die zu Implementierende Analysesoftware für Garbage Collection Log Dateien soll als einen Rich Client oder als Plugin für eine bestehende Entwicklungsumgebung implementiert werden. Die am meisten verbreiteten Plattformen dieser Art sind die Netbeans Plattform und Eclipse. Die Weiterentwicklung von Eclipse findet aktuell in zwei Entwicklungssträngen statt, während die Version 3.x (aktuell 3.7 Eclipse Indigo) eine weit verbreitete Plattform ist, soll die Version 4.x (aktuell 4.1) aber wegweisend sein und die Zukunft von Eclipse aufzeigen. Dahinter verstecken sich zwar ähnliche Konzepte, das Programmiermodell ist allerdings ziemlich stark überarbeitet und lehnt sich in einigen Bereichen auch an bereits vorhandene Java Standars[1] an. Im ersten Teil dieses Abschnitts gibt es eine kurze Übersicht der beiden Frameworks, im zweiten Teil die Bewertung der verschiedenen Stärken. Der dritte Teil beinhaltet dann übersichtshalber schon hier die Entscheidungsgrundlagen und den Entscheid für eine dieser Plattformen.

5.1 Beschreibung

5.1.1 Eclipse RCP

Bis zur Version 2.1 war Eclipse bekannt als eine Open Source Entwicklungsumgebung für Programmierer. Der Vorgänger hiess Visual Age vor Java und wurde von IBM entwickelt.

Auf die Version 3.0 wurde die Architektur von Eclipse relativ stark umgestellt und modularisiert. Nun handelt es sich um einen relativ kleinen Kern

[1] als Beispiel sei hier “Dependency Injection for Java” (JSR 330) genannt

der Applikation, der die eigentliche Funktionalität der Applikation als Plugins lädt. Diese Funktionalität basiert auf Eclipse Equinox, einer Implementation der OSGi Spezifikation. Die grafischen Benutzeroberflächen sind in SWT implementiert. Eclipse ist für 14 verschiedene Systeme und Architekturen bereitgestellt und gilt somit als plattformunabhängig [**?**].

Die Plattform kann nun auch als Framework zur Entwicklung von Desktop Applikationen oder Plugins für die Entwicklungsumgebung entwickeln verwendet werden. Bei eigenen Desktop-Anwendungen spricht man auch von Eclipse-RCP[2] Anwendungen.

5.1.2 Netbeans RCP

Wie bei Eclipse RCP handelt es sich bei Netbeans RCP ebenfalls um ein Framework zur Entwicklung von Desktop Anwendungen. Dem Entwickler wird ein API für typische Anforderungen in diesem Bereich zur Verfügung gestellt. Der Kern der Netbeans Plattform besteht ebenfalls aus einem Modul-Loader und im Bereich der grafischen Benutzeroberfläche wird Swing verwendet.

5.2 Entscheidungskriterien

Tabelle 5.1: Entscheidungskriterien Rich Client Frameworks

	Gw.[3]	**Eclipse 3.x**	**Pkt.**[4]	**Eclipse 4.x**	**Pkt.**	**Netbeans 3.x**	**Pkt.**
Architektur, Komponenten							
Modularisierung	5	OSGi (5)	25	OSGi (5)	25	OSGi / Netbeans Module System (5)	25
GUI[5]	5	SWT (4)	20	XWT / SWT (5)	25	Swing (4)	20
Tooling							
Build Tool	3	PDE, Maven Tycho (5)	15	PDE, Maven Tycho (5)	15	Ant, Maven (5)	15
Deployment	3	PDE, Maven Tycho (5)	15	PDE, Maven Tycho (5)	15	Ant, Maven (5)	15

[2] Rich Client Platform
[3] Gewichtung [1-5]
[4] Anzahl Punkte: Gewichtung x erziehlte Punkte
[5] Graphical User Interface

CI[6]	2	Maven[7] (5)	10	Maven (5)	10	Ant[8], Maven	10
Strategisch							
Ressourcen (Bücher, Tutorials, Foren, Internet)	5	viele (5)	25	sehr wenige (1)	5	mittel (3)	15
Reife	5	stark (5)	25	sehr gering (1)	5	mittel (3)	15
Verbreitung	5	stark (5)	25	sehr gering (1)	5	mittel (3)	15
Potential	3	mittel (3)	9	viel (4)	12	viel (4)	12
Diverses							
Präferenzen, Erfahrung Entwickler	5	mittel (3)	15	wenig (2)	10	sehr wenig (1)	5
Total	4		**184**		127		147

5.3 Entscheid

Insbesondere aufgrund der Präferenzen des Entwicklers, der Verfügbarkeit an Ressourcen wie Bücher, Tutorials und Internet-Einträgen und des Reifegrades fiel der Entscheid auf Eclipse 3.x.

[6] Continuous Integration
[7] Maven Projekte lassen sich automatisch auch in CI-Server wie Hudson integrieren.
[8] Ant-Projekte lassen sich ebenfalls gut in CI-Server integrieren.

Kapitel 6

Anforderungsanalyse

6.1 Einleitung

6.1.1 Zweck

6.1.2 Systemumfang

6.1.3 Stakeholder

6.1.4 Übersicht

6.2 Allgemeine Übersicht

6.2.1 Systemumfeld

6.2.2 Architekturbeschreibung

6.2.3 Systemfunktionalität

6.2.4 Nutze und Zielgruppen

6.2.5 Randbedingungen

6.2.6 Annahmen

6.3 Anforderungen

6.3.1 Funktionale Anforderungen

6.3.2 Qualitätsanforderungen

Kapitel 7

Konzept

Kapitel 8

Implementation

Kapitel 9

Review

Glossar

Tabelle 9.1: Glossar

Wort	Beschreibung	Herkunft

Literaturverzeichnis

[1] M. Lagergren and M. Hirt. Oracle Jrockit: The Definitive Guide. Packt Publishing, Limited, 2010.

[2] A. Langer and K. Kreft. Generational garbage collection. Java Magazin, 3:26–30, 2010.

[3] A. Langer and K. Kreft. Mark-and-compact. Java Magazin, 7:20–24, 2010.

[4] A. Langer and K. Kreft. Young generation garbage collection. Java Magazin, 5:24–30, 2010.

[5] Sun Microsystems. Memory management in the hotspot virtual machine. http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf, 2006.

[6] Oracle. Oracle jrockit command-line reference. http://download.oracle.com/docs/cd/E15289_01/doc.40/e15062.pdf, 2011.

[7] K. Pohl and C. Rupp. Basiswissen Requirements Engineering: Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering– Foundation Level. Dpunkt.Verlag GmbH, 2010.

Listings

Abbildungsverzeichnis

Tabellenverzeichnis

Teil III

Anhang