JUnit Multithreaded Tools

by Hölzl Gerold
Kurz Kornelius
Martin Lehofer
Markus Gaisbauer

Inhaltsverzeichnis

1. Multithreading
1.1 Was ist Multithreading
1.2 Race Condition
1.3 Deadlock

2. Best Practices in Java
2.1 Bestimmen von gemeinsam genutzten Daten
2.2 Volatile / Synchronized Schlüsselwörter
2.3 Wie synchronisieren?
2.4 Performance

3. Ansätzte zur Erkennung von Synchronisationsproblemen
3.1 „Happen-Before“ Analyse
3.2 Instrumentation
3.3 Alnalyse des Laufzeitverhaltens
3.4 Disziplin bei Zugriff

4. Tools im Test
4.1 JProbe Threadanalyzer
4.2 JLint
4.2.1 Beschaffung & Installation
4.2.2 Benützung
4.2.3 Arbeitsweise
4.2.4 Analyse von Deadlocks in JLint
4.2.5 Analyse von Race Conditions in Jlint
4.3 JMTUnit
4.3.1 Threading Model
4.3.2 Testcase
4.3.3 Konfiguration
4.3.4 Beispiel Deadlock
4.3.5 Beispiel Racecondition
4.3.6 Performance Report

5. Zusammenfassung

6. Quellen

1. Multithreading

1.1 Was ist Multithreading?

Moderne Betriebssysteme ermöglichen es, dass verschiedene Programme quasi gleichzeitig also scheinbar parallel ausgeführt werden. Diese Betriebssysteme werden als multitaskingfähig bezeichnet.
Die quasiparallele Ausführung wird durch das Betriebssystem ermöglicht, welches jedem Programm für eine gewisse Zeit die Systemressourcen zur Verfügung stellt und diese alle paar Millisekunden umschaltet.
Der „Scheduler“ welcher für das Umschalten verantwortlich ist spielt dem Anwender durch die geschickte „Verzahnung“ der einzelnen Programme die Parallelität also nur vor.

Die dem Betriebssystem bekannten, aktiven Programme bestehen aus einzelnen Prozessen. Ein Prozess setzt sich aus dem Programmcode und den dazugehörigen Daten zusammen und besitzt einen eigenen Adressraum. Die virtuelle Speicherverwaltung des Betriebssystems trennt die Adressräume der einzelnen Prozesse. Dadurch ist es unmöglich, dass ein Prozess den Speicherraum eines anderen Prozesses korrumpiert, da er auf den anderen Speicherbereich nicht zugreifen kann.

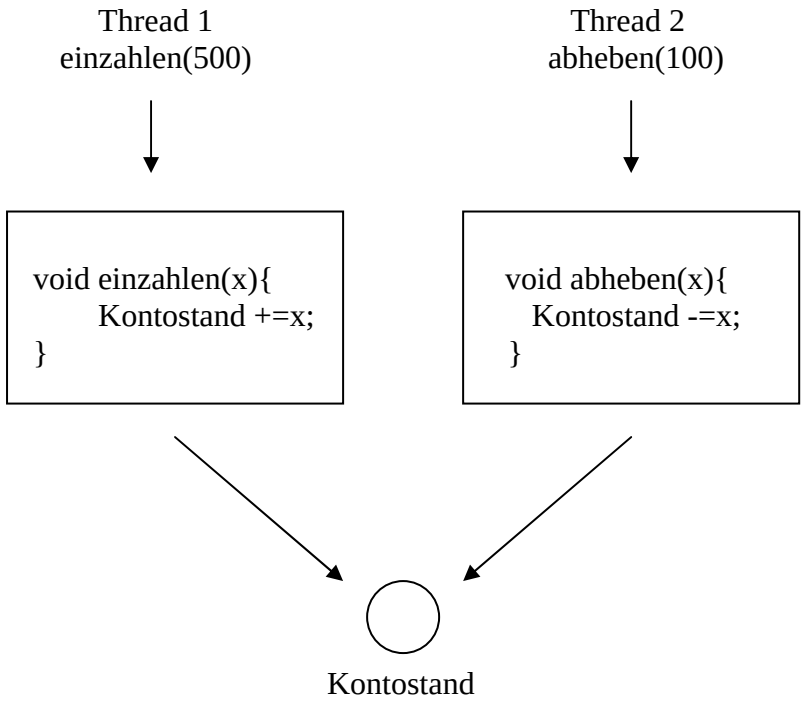
Mehrere Threads können wie Prozesse verzahnt ausgeführt werden, sodass sich für den Benutzer der Eindruck von Gleichzeitigkeit ergibt. Ist dies der Fall, dann wird die Umgebung, in der das Programm läuft als „multithreaded“ bezeichnet.

Mittlerweile unterstützen viele Betriebssysteme direkt Threads und auch in C(++) wird paralleles Programmieren durch passende Bibliotheken populär.
Doch besonders die Integration in die Sprache macht das Entwerfen nebenläufiger Anwendungen in Java einfacher.
Durch die verzahnte Ausführung kommt es allerdings zu Problemen. Es besteht die Gefahr konkurrierender Zugriffe auf gemeinsam genutzte Ressourcen („race condition“). Dies kann zwar durch Programmkonstrukte abgefangen werden, jedoch steigt dann die Gefahr von sog. Deadlocks wo Threads bis in alle Ewigkeit aufeinander warten.

[HPM]

1.2 Race Conditions

Wenn Threads auf gemeinsame Daten zugreifen, können sie sich dabei in die Quere kommen. Nehmen wir zum Beispiel an, wir hätten eine Klasse Konto um ein Bankkonto zu verwalten. Ein Feld *Kontostand* enthält den aktuellen Kontostand, eine Methode *einzahlen(x)* erhöht den Kontostand um x Euro und eine Methode *abheben(y)* bucht y Euro vom Konto ab. Nehmen wir weiters an, dass *abheben(y)* und *einzahlen(x)* gleichzeitig von verschiedenen Threads aufgerufen werden.



Der kritische Bereich bei diesem gemeinsamen Zugriff ist in jenem Codeteil, indem *abheben(y)* und *einzahlen(x)* auf Kontostand zugreifen. Das Problem dabei ist erstens, dass dieser Zugriff nicht atomar ist, das heißt vom Compiler in mehrere Einzelschritte aufgelöst wird.
Zweitens können Threads zu beliebigen unvorhergesehenen Zeitpunkten unterbrochen werden und die Kontrolle an einen anderen Thread verlieren.
Es könnte der Fall eintreten, dass Thread 1 bei der Aktualisierung von Kontostand unterbrochen wird. Was geschieht in diesem Fall?

Thread 1 liest den aktuellen Wert von *Kontostand* (z.B.: 1000) und addiert 500. Bevor er aber den Wert 1500 zurück schreiben kann, wird er unterbrochen und die Kontrolle wechselt zu Thread 2. Dieser liest nun ebenfalls den Wert von *Kontostand*, der noch immer 1000 ist, subtrahiert 100 und speichert das Ergebnis 900 zurück. Irgendwann wechselt die Kontrolle wieder zurück zu Thread 1, der nun seinerseits sein Ergebnis von 1500 nach *Kontostand* zurückspeichert. Der Wert von *Kontostand* ist nun fälschlicher Weise 1500 und nicht 1400. Tritt so etwas ein, spricht man von einer „race condition“

Fehler dieser Art sind nur sehr schwer zu finden, da sie sich kaum reproduzieren lassen. Sie treten nur auf, wenn die Umschaltung zwischen den Threads zu einem besonders ungünstigen Zeitpunkt erfolgt, was nur sehr selten der Fall ist.
Der Grund des Übels liegt darin, dass Thread 1 in seinem kritischen Bereich unterbrochen wurde, in dem er schreibend auf gemeinsame Daten zugreift. Man muss solche Unterbrechungen verhindern, bis der Thread seinen kritischen Bereich verlassen hat. In Java kann man das erreichen, indem man kritische Bereiche mit einer Sperre (lock) versieht, die nur einen der Threads durchlässt, und alle anderen Threads, die zur gleichen Zeit versuchen, ihre kritischen Bereiche zu betreten, so lange blockiert, bis der 1. Thread fertig ist. Dann darf der nächste der wartenden Threads in seinen kritischen Breich und so weiter, bis kein Thread mehr wartet. Die Sperre sorgt also dafür, dass sich immer nur ein einziger Thread in seinem kritischen Bereich befindet und somit keine „race condition“ auftreten kann.

[HPM]

1.3 DeadLock

Soll ein Programmabschnitt oder eine Methode atomar ablaufen, so gibt es in Java wie oben beschrieben, ein Schlüsselwort *synchronized*, welches dies bewerkstelligt.

Nehmen wir als Beispiel die Methode foo():

```
public class IPlusPlus{
        static int i;
        static void foo(){
                i++;
        }
}
```

Würde diese Methode wieder von mehreren Threads aufgerufen und gleichzeitig geändert, würde der Wert der Variablen I nicht mehr stimmen.
Darum benötigen wir das Schlüsselwort „*synchronized*“ welches vor die Methode geschrieben wird.

Synchronized void foo() {i++}

Kommt es nun zu einem Konflik, dass mehrere Threads die Methode foo() aufrufen, verhindert *synchronize*, dass sich mehr als ein Thread gleichzeitig im kritischen Abschnitt, dem Rumpf der Methode foo(), befinden kann.

Tritt ein Thread in den atomaren Block ein, so kann man sich vorstellen, dass die virtuelle Maschine die Methode wie eine Tür abschließt. Erst wenn die Methode wieder beendet wird, schließt die JVM die Tür wieder auf und ein anderer Thread kann die Methode betreten. Das Ein- und Austreten wird von der Java-Viturell-Maschine übernommen, und man muss das nicht kontrollieren. Kommt ein zweiter Thread zu der abgeschlossenen Methode (das Objekt ist verriegelt), muss er warten und wird erst dann hineingelassen, wenn die Markierung gelöscht ist. So ist die Abarbeitung über mehrere Threads einfach synchronisiert.

Wie kommt es nun zu einem *Deadlock*?
Ein *Deadlock* kommt dann vor, wenn ein Thread A eine Ressource belegt, die ein anderer Thread B haben möchte. Dieser Thread B belegt aber eine Ressource, die A gerne bekommen möchte. In dieser Situation können beide nicht vor und zurück und befinden sich in einem dauernden Wartezustand. *Deadlocks* können in Java-Programmen nur schwer erkannt und verhindert werden. Die neue Sun Java Virtual-Machine besitzt eine eingebaute Deadlock-Erkennung, die auf der Konsole aktiviert werden kann.
Es ist jedoch sehr schwierig, *Deadlocks* zu reproduzieren womit die nicht triviale Aufgabe ein Programm zu erstellen in dem keine *Deadlocks* auftreten dem Programmierer überlassen bleibt. [CU]

2. Best Practices in Java

2.1 Bestimmen von gemeinsam genutzten Daten

Bereits beim Entwurf eines Programms sollte geklärt werden welche Daten gemeinsam benutzt werden. Objektorientierte Enwurfstechniken wie Information-Hiding und Kapselung reduzieren nach außen sichtbare Daten und verringern somit den Aufwand zur Absicherung von Threads. Es müssen aber nicht nur komplexe Datenstrukturen geprüft werden sondern auch scheinbar triviale Fälle, wie das prüfen ob eine Referenz null ist.

2.2 volatile / synchronized-Schlüsselwörter

Alle Leseoperationen von primitiven Datentypen sollten synchronisiert durchgeführt werden, da die JVM nur bei Anwendung des volatile-Schlüsselwortes die Konsistenz von Variablen garantieren kann. Warum müssen auch primitive Datentypen synchronisiert werden, wo solche Operationen schon von Hardwareseite sicher atomar sind? Kopien von lokalen Variablen werden von jedem Thread einzeln gespeichert werden und müssen nicht überall konsistent sein, obwohl man es vielleicht erwarten würde. Wie verhindert man dass? Java hält für diese Fälle das (eher unbekannte) volatile-Schlüsselwort bereit. Was ist der Unterschied von synchronized zu volatile? synchronized kann Sperren auf Code-Blöcke setzen und lösen um so Code von nur einem Thread ausführen zu lassen und synchronisiert den gesamten lokalen Thread-Speicher mit dem „main“-Speicher. volatile modifiziert ein Feld und verhindert dass ein Thread von einem Feld lokale Kopien hält und zwingt ihn, immer auf das Feld im „main“-Speicher zuzugreifen.

2.3 Wie synchronisieren?

In einfachen Fällen reicht es aus einzelne Felder als volatile zu kennzeichnen. Die Fehler beim versuchen Synchronisationsprobleme zu vermeiden treten aber nicht durch Inkonsistenz von Feldern auf sondern durch zu enge synchronized-Blöcke. Es reicht nicht aus nur das lesen und schreiben von Daten nur auf synchronisierten Feldern zuzulassen, vielmehr muss auch die Bearbeitung der gelesen Daten durch synchronized geschützt sein. Bereits im Entwurf sollte feststehen welche Transaktionen atomar sind. Um solche Fehler zu vermeiden und die Übersicht im Code zu steigern sollten möglichst nur ganze Methoden und nicht einzelne Blöcke als synchronized gekennzeichnet werden.

2.4 Performanz

Wahrscheinlich denken die meisten Entwickler beim Stichwort „Synchronisation“ an nur ein Wort: Performanz. Natürlich ist die Synchronisation der Performanz abträglich, allerdings ist der Schaden durch nur eine einzige verzichtete Synchronisation um vieles höher als der Verlust durch eine um ein paar Prozent längere Laufzeit.

3. Ansätze zur Erkennung von Synchronisationsproblemen

Die einfachste, aber weder zuverlässigste noch schnellste Möglichkeit Probleme bei nebenläufigen Programmen auf die Spur zu kommen ist der Peer-Review durch Kollegen und die Überprüfung ob alle oben angeführten Best Practices eingehalten wurden.
Spätestens seit der weiten Verbreitung von Test-Tools wie JUnit erfreut sich aber das automatische Testen vieler Freunde. Warum also nicht auch etwas ähnliches für nebenläufige Programme? Noch gerne würden es Programmierer sehen wenn bereits der Compiler oder die Entwicklungsumgebung auf solche Fehler hinweist. Im Folgenden werden einige Ansätze und Algorithmen vorgestellt die dies ermöglichen.

3.1 „happens-before“-Analyse

[Lamport78] formulierte dass der Zugriff auf gemeinsame Daten in einer wohldefinierten Reihenfolge erfolgen muss, und führte dazu die happens-before Beziehung ein. Ein Werkzeug das den „happens-before“-Algorithmus verwendet kann aber nur Fehler im Ablauf von synchronisierten Blöcken feststellen, nicht aber ob Transaktionen vollständig sind.

3.2 Instrumentation

Ein weiterer Ansatz ist in ConTest [Edelstein01] zu finden, wo an entscheidenden Stellen sleep(), yield() und priority() Befehle eingefügt werden und das Programm mehrfach ausgeführt wird. Ein positiver Testfall tritt dann ein wenn bei unterschiedlichen Schedulings unterschiedliche Ergebnisse auftreten.

3.3 Analyse des Laufzeitverhaltens

Das Tool JProbe Threadalizer versucht(e) das Laufzeitverhalten von Programmen durch Aufzeichnung von Sperrungs-Anfragen und Sperrungs-Auflösungen zu analysieren. Dieses Verhalten hängt aber sehr stark von nicht direkt beeinflussbaren Faktoren wie dem Scheduler

der JVM und des Betriebssystems ab und ist deswegen nicht reproduzierbar. Obwohl für alle Testmethoden natürlich gilt dass sie die Fehlerfreiheit eines Programms nicht verifizieren können gilt dies für solche Methoden natürlich ganz besonders. [Kunz03] zeigt außerdem dass leicht falsch-Positive generiert werden können, da solche Methoden unter Umständen auch (beabsichtigte) Endlosschleifen und wait()-Anweisungen als Deadlocks interpretiert. Unter diesen Umständen klingt der Ansatz von ConTest besser, ein Vergleich beider Tools wäre wohl sehr interessant.

3.4 Disziplin bei Zugriff

[Savage97] zeigt außerdem noch eine andere Möglichkeit, nämlich die Disziplin eines Programms bei der Synchronisation zu prüfen, konkret ob auf ein gemeinsames Feld immer im Besitz desselben Schlüssels zugegriffen wird. Ein Algorithmus hierzu findet sich bei [Kunz03]
Dieser Algorithmus schlägt aber auch bei einigen Konstrukten an die völlig harmlos sind: für Java relevante aus [Savage97]:

Initialisierung: Gemeinsame Daten müssen nicht synchronisiert initialisiert werden.
Gemeinsames Lesen: Ist sichergestellt dass Daten nicht verändert werden ist ebenfalls keine Synchronisation notwendig.
Eigene Synchronisationsmechanismen: Es ist schwierig bis unmöglich eigene Mechanismen zur Synchronisation als solche zu erkennen.

4. Tools im Test

4.1 JProbe Threadalizer

Im Readme der aktuellen Version 5.2.2 ist der Hinweis zu finden dass das Threadalizer-Modul eingestellt worden ist, was aus der Website des Herstellers leider nicht hervorgeht.

4.2 Jlint

4.2.1 Beschaffung & Installation

JLint ist ein OpenSource-Tool das ähnlich wie das FindBugs-Projekt sich auf die Analyse von Java-Programmen konzentriert. Es ist unter der GNU-Lizenz verfügbar [Artho04], ebenso wie ein Plugin für die Eclipse-Entwicklungsumgebung [Willow04]. Die Installation ist wie von Eclipse-Plugins gewohnt sehr einfach: Plugin im Eclipse-Ordner entpacken, JLint in denselben Ordner kopieren und in Eclipse konfigurieren. Jlint bietet auch eine Integration in Ant und lässt sich so in einen automatischen Build-Prozess integrieren.

4.2.2 Benützung

Ein Test von Projekten, Packages oder Dateien durch JLint erfolgt unter Eclipse im Package Explorer durch einen rechten Mausklick und Auswahl des Jlint Items im Kontexmenu. Das Plugin erzeugt nun aus dem Output von JLint Eclipse-spezifische Warnungen, die in der Problem-View angezeigt werden können (Filter anpassen!). Es kann (zumindest derzeit noch) keine Hintergrundüberwachung der erzeugten Klassen durchgeführt werden. Sobald eine der Sourcen, die von JLint Warnungen betroffen ist, neu gespeichert wird, verschwinden automatisch ALLE JLint Warnungen. Ein erneuter Aufruf von JLint muss explizit durch den Benutzer durchgeführt werden.

4.2.3 Arbeitsweise

Die Arbeitsweise von JLint beruht auf Analysen des Datenflusses und von lock-Graphen. Es besteht aus zwei Programm, AntiC und JLint. AntiC überprüft die Syntax von C ähnlichen Programmiersprachen (C, C++, Objective C, Java) auf mögliche Fehler bei der Folge von Operatoren, fehlenden breaks in switch() Anweisungen usw. AntiC lässt sich auch getrennt von Jlint einsetzen. JLint arbeitet mit kompilierten Class-Files und nicht direkt auf den Programmquellen. Die Entwickler geben dies auch als Grund für die hohe Geschwindigkeit

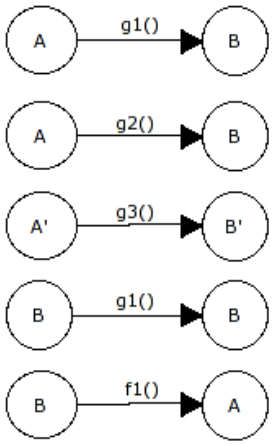
des Tools an. Durch Debug-Information verknüft JLint dann mögliche Fehler mit den Quellen um schnelle Fehlersuche zu ermöglichen (im Eclipse-Plugin etwa).
Jlint führt des Weiteren lokale und globale Datenstromanalysen durch und berechnet mögliche Werte von lokalen Variablen um etwa redundante Anweisungsblöcke zu erkennen. JLint kann auch erkennen ob Methoden möglicherweise mit null-Werten als Parameter aufgerufen werden und ob dies nicht überprüft wird. Eine ausführliche Dokumentation aller erkennbaren Fehler bietet [Artho04].

4.2.4 Analyse von Deadlocks in Jlint

JLint geht bei der Analyse von Deadlocks folgendermaßen vor: Da die einzelnen Instanzen eines Objektes vor der Kompilierung nicht bekannt sind wird nach der Kompilierung eine Analyse der Abhängigkeiten zwischen den Klassen durchgeführt. Jede Klasse wird durch 2 Knoten dargestellt, einen für die Klasse selbst (C), und einen für die Metaklasse (C'). Die Knoten für die Klasse selbst wird für Abhängigkeiten von Objektmethoden verwendet, der Knoten für die Metaklasse für Abhängigkeiten von statischen Methoden. Die Kante (A,B) „foo“ bedeutet also dass die synchronisierte Methode foo der Klasse B direkt oder indirekt von einer von einer synchronisierten Methode der Klasse A aufgerufen werden kann. Ein Deadlock *kann* möglich sein wenn der Graph nicht kreisfrei ist. JLint produziert auch Warnungen wenn keine Deadlocks möglich sind, aber Kreise im Graphen auftreten. Da die Aufgabe alle Kreise in einem Graphen zu finden zu der Klasse der NP-Probleme gehört, beschränkt Jlint die Suchtiefe.
Eine weiteres Verfahren ist die Analyse von wait()-Anweisungen. Die wait() und notify() Methode sollten nur aufgerufen werden wenn der Monitor gesperrt ist. Wenn der Thread durch einen notify() Aufruf aufwacht, versucht er den Monitor wieder zu sperren. Ein Problem ist, dass nur eine einzige Sperre auf einen Monitor gelöscht wird. Wenn wait() aus einer synchronized-Methode eines anderen Objektes aufgerufen wird, wird dieser Monitor von wait() nicht freigegeben. Wenn ein Thread, der einen schlafenden Thread aufwecken soll eine synchronized-Methode von dem Objekt aufruft tritt ein Deadlock auf: Ein Thread schläft und der andere, der ihn aufwecken soll wartet dass der Monitor entsperrt wird. Jlint kann solche Situationen aufspüren.

Beispiel

```
class A {
    public synchronized void f1(B b) {
        b.g1();
        f1();
        f2();
    }
    public void f2(B b) {
        b.g2();
    }
    public static synchronized void f3() {
        B.g3();
    }
}
class B {
    public static A ap;
    public static B bp;
    public synchronized void g1() {
        bp.g1();
    }
    public synchronized void g2() {
        ap.f1();
    }
    public static synchronized void g3() {
        g3();
    }
}
```



(Aus: [Artho04])

4.2.5 Analyse von Race-Conditions in Jlint

Jlint erstellt eine Liste aller Methoden die nebenläufig ausgeführt werden können. Zuallererst sind dies natürlich Methoden die mit dem synchronized-Statement gekennzeichnet sind. Auch run-Methoden von Klassen die das Interface Runnable implementieren oder von der Klasse Thread abgeleitet sind werden in diese Liste aufgenommen. Dann werden rekursiv alle Methoden die von einer schon auf der Liste stehenden Methode aufgerufen werden ebenfalls aufgenommen.

Jlint gibt eine Warnung aus wenn alle Folgenden Bedingungen zutreffen:

1. Die Methode die auf ein Feld zugreift ist in der Liste nebenläufiger Methoden
2. Feld ist nicht als volatile oder final deklariert
3. Feld gehört nicht zum this-Objekt der Methoden
4. Feld gehört nicht zum einem neu instanziertem Objekt auf das nur durch eine lokale Variable zugegriffen wird
5. Auf das Feld kann von Methoden anderer Klassen zugegriffen werden

(Punkt 5 verhindert Warnungen da nicht alle Objekte auf die Zugriffe von Threads möglich sind synchronisiert werden müssen und dürfen). Die Jlint Entwickler sehen diese Regeln aber nicht als vollständig oder unverrückbar an.

4.3 JMT Unit

Im folgenden wird JMTUnit [Moser04], ein Framework für Unit Tests von nebenläufigen, Java-basierterten Applikationen, beschrieben. Das bereitgestellte API ist stark an das von JUnit angelehnt.
Viele der von JUnit bekannten Konzepte für Unit Tests, wie zum Beispiel TestCases, TestSuits, TestResults und Assert-Funktionen wurden übernommen. Zusätzliche wurde dem Framework ein Threading Modell hinzugefügt, das eine Simulation quasiparalleler Ausführung ermöglicht.

4.3.1 Threading Model

Es wird eine Testumgebung mit einer fixen Anzahl an Worker-Threads und virtuellen Benutzern simuliert. In Normalfall werden viele Benutzer auf einigen wenigen Worker-Threads ausgeführt. Jeder Benutzer führt eine bestimmte sequentielle Folge von Simulations-schritten aus, die sich zeitlich nicht überlappen und für alle Benutzer identisch sind. Jeder Simulationsschritt wird durch die ID des zugehörigen virtuellen Benutzer und einer fortlaufenden Nummer eindeutig identifiziert.

Alle Worker-Threads warten in einer gemeinsamen Input-Queue auf den nächsten Simulationsschritt. Sobald ein Simulationsschritt beendet wurde wird dies auf einer Output-Queue vermerkt und auf die Ausführung des nächsten Schrittes gewartet.
Ein weitere Thread überwacht die Output Queue auf abgeschlossenen Simulationsschritte und fügt Folgeschritte der Input Queue hinzu.

Das Programm endet sobald alle virtuellen Benutzer erfolgreich alle Simulationschritte durchgeführt haben oder aufgrund eines Fehlers (z.B. wegen eines nicht erfüllter asserts) ihre Ausführung abgebrochen haben.

Während der Ausführung der Simulationsschritte werden ständig momentane Zeitpunkte mitaufgezeichnet und am Ende des Tests werden aus diesen Daten statistische Eckdaten (Durchsatz/Wartzeit/Antwortzeit) berechnet und ausgegeben.

Jeder virtuelle Benutzer hat ein eigenes Log Verzeichnis. Dort werden Fehler geloggt und optional kann auch direkt darauf zugegriffen werden.

4.3.2 TestCase

Um einen JMTUnit Test auszuführen muss eine Unterklasse von TestCase erzeugt werden in der folgende Methoden überschrieben sind.

void runTest(TestCaseParams params)
Methode wird für jeden Simulationsschritt einmal ausgeführt. TestCaseParams stellt unter anderem Zugriff auf das Log und die Daten des virtuellen Benutzers sicher.

Object setUpVirtualUser()
Methode gibt benutzerspezifische Daten für aktuellen virtuellen zurück.

Zusätzlich können Methoden überschrieben werden, die am Anfang bzw. Ende der Lebenszyklen von Worker Threads, virtuellen Benutzern oder Tests ausgeführt werden.

4.3.3 Konfiguration

Der Ablauf von JMTUnit Tests wird über eine einfache XML-Datei konfiguriert.

```
<test>
  <testrun numofworkerthreads="5" numofvirtualusers="15"
           testsequencelength="10" debuglevel="3" />
  <userparameters></userparameters>
</test>
```

4.3.4 Beispiel Deadlock

```
public class TestDeadLock extends TestCase {

	Object a = new Object(); Object b = new Object(); Object c = new Object();

	public Object setUpVirtualUser() throws Exception {
		return new Deadlock(a, b, c);
	}

	public void runTest(TestCaseParams params) {
		Deadlock d = (Deadlock) params.getVirtualUserData()
		d.doThis(); d.doThat(); d.doSomething();
	}

	public static void main(String arg[]) {
		TestSuite t = new TestSuite();
		TestDeadLock tc = new TestDeadLock();
		t.run(tc);
	}
}
```

Jeder virtuelle User führt in einem Simulationsschritt die drei Methoden doThis, doThat und doSomething aus. Alle Benutzer arbeiten auf den selben Objekten a, b und c. In der main-Methode wird eine TestSuite mit einem Test erzeugt und ausgeführt.

```
/**
 * Circular Wait may occur calling toThis, doThat, doSomething
```

```
 * Mutual Exclusion, Hold and Wait, No Preemption caused by usage of java-monitors
 * ==> DEADLOCK
 */
public class Deadlock {

	Object a, b, c;

	public Deadlock(Object a, Object b, Object c) {
		this.a = a; this.b = b; this.c = c;
	}

	public void doThis() {
		synchronized (a) {
			System.out.println("waiting for b, holding a");
			synchronized (b) {
	}}}

	public void doThat() {
		synchronized (b) {
			System.out.println("waiting for c, holding b");
			synchronized (c) {
	}}}

	public void doSomething() {
		synchronized (c) {
			System.out.println("waiting for a, holding c");
			synchronized (a) {
	}}}
}
```

Wie gleich ersichtlich ist, kann es bei der verzahnten nebenläufigen Ausführung der drei Methoden zu einem Deadlock kommen. Folgender Output wird produziert:

```
Creating  5 Worker threads...
Creating tests objects for 15 Virtual Users...
Each user runs a test sequence consisting of 10 steps
Running requests ...
--> get request [thread0 user 0 test 0]
waiting for b, holding a
waiting for c, holding b
waiting for a, holding c
--> finish request [thread0 user 0 test 0]
...
...
--> get request [thread3 user 6 test 3]
--> get request [thread4 user 0 test 8]
--> get request [thread2 user 9 test 1]
--> get request [thread1 user 8 test 3]
waiting for b, holding a
waiting for c, holding b
waiting for a, holding c
waiting for b, holding a
waiting for c, holding b
waiting for b, holding a
```

An dieser Stelle tritt der Deadock auf. 3 Threads warten zyklisch auf die Freigabe der Monitore a, b und c. Der Deadlock kann aber nicht automatisch erkannt werden, die Ausführung bleibt einfach stehen, der Benutzer muss dann seine eigenen Schlüsse aus dem Ergebnis ziehen.

4.3.5 Beispiel Race Condition

```
class Account {

        protected int balance;


    public boolean get(int sum) {

        if (sum <= balance) {

            balance -= sum;

            return true;

        }

        return false;

    }

    public synchronized void add(int sum) {

        balance = sum;

    }

}
```

Die Klasse modelliert ein einfaches Bankkonto von dem ein Geldbeträge abgehoben und eingezahlt werden dürfen. Ein Überziehen des Kontos soll nicht erlaubt sein. Da die Methode get nicht synchronisiert ist können Race Conditions auftreten. Zur Vollständigkeit die Methode runTest für unseren Testfall:

```
public void runTest(TestCaseParams params) {
    if(!account.get(100)) account.add(500);
    assertTrue("Balance below zero, is "+account.balance, account.balance>=0);
}
```

Die Race Condition konnte von JMTUnit trotz 50 Threads, 200 virtuellen Benutzern und 10.000 Simulationsschritten nicht entdeckt werden. Die Wahrscheinlichkeit, dass der Thread genau an der ungünstigen Stelle unterbrochen wird, ist so gering, dass die Race Condition wohl laut Murphy’s Gesetz erst nach der Auslieferung der Software zum ersten Mal auftritt. Um etwas nachzuhelfen beschäftigen wir den Thread, nach der Überprüfung ob sum <= balance ist, ein wenig mit einer ArcCos Berechnung (`Math.acos(0.323525235235323);`). Jetzt wird die Race Condition entdeckt und im Output vermerkt.

```
There are users that threw an exception in runTest method.
Failure 2004.11.18 at 05:01:27::34 CET
[user=183 teststep=726 thread=12]: Balance below zero, is -100
```

```
[stack]
JMTUnit.AssertionFailedError: Balance below zero, is -100
       at JMTUnit.AssertUtil.fail(AssertUtil.java:47)
       at JMTUnit.AssertUtil.assertTrue(AssertUtil.java:20)
       at Bank.TestAccountManager.runTest(TestAccountManager.java:21)
       at JMTUnit.WorkerThread.run(WorkerThread.java:70)
       at java.lang.Thread.run(Thread.java:595)
```

4.3.6 Performance Report

JMTUnit kann auch als Framework für Performance Tests verwendet werden. Sollte das Programm Deadlock frei sein, bzw. sollten vorhandene Deadlocks nicht aufgespürt werden, wird am Ende ein Performance Report ausgegeben. Testläufe die aufgrund nicht erfüllter "asserts" abbrechen werden als "crashed" bezeichnet.

```
================================================
Performance Report
================================================
Crashed sessions: 3
Requests serviced: 48 (including requests that crashed)

Throughput    - serviced requests per second
  mean time: 1
  minimum:   0
  maximum:   30
Waiting Time  - time that requests waits in queue
  mean time: 1
  minimum:   0
  maximum:   20
Response Time - time from submission of request until response
  mean time: 2
  minimum:   0
  maximum:   40
```

5. Zusammenfassung

Ein bekannter Spruch besagt zwar dass man aus Fehlern lernt, jedoch gehören Synchronisierungsfehler sicher nicht dazu. Da sie unvorhersagbar auftreten, unter Umständen völlig unbemerkt und sich kaum Rückschlüsse auf ihre Ursachen ziehen lassen sind sie besonders tückisch. Sie lassen sich am Besten bereits im Entwurf durch sauberes Design und Festlegung welche Methoden atomar durchgeführt werden müssen vermeiden.
Java-Entwickler sollten die volatile-Anweisung in ihren Wortschatz aufnehmen und synchronized-Anweisungen die nicht in der Methodendeklaration vorkommen grundsätzlich kritisch betrachten.
Es stimmt zwar dass Synchronisation Laufzeit kostet, doch existieren wohl nur wenige reale Fälle wo dies Leistungsmäßig relevant ist, davon abgesehen welche Folgekosten Synchronisationsprobleme verursachen können.
Jlint ist genial einfach zu benutzen und sehr hilfreich bei allen möglichen Arten von Programmierfehlern. Es ist für jeden Java-Programmierer ein absolutes Muss, da es nicht nur viele Arten von Synchronisationsproblemen aufdecken kann.

6. Quellen

[HPM] Hanspeter Mössenböck – Sprechen sie Java 2.Auflage

[CU] Christian Ullenboom – Java ist auch eine Insel Auflage 2003

[Artho04] Cyrille Artho, Konstantin Knizhnik: Jlint – Java Program Checker, http://artho.com/jlint/

[Kunz03] Philipp Kunz: Korrektheits-Prüfung von Java-Programmen mit mehreren Threads: JProbe Threadalizer http://www.cs.fh-aargau.ch/~gruntz/courses/sem/ws02/threadalyzer.pdf

[Lamport78] L. Lamport: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM (CACM), Vol. 21, No. 7, Juli 1978

[Savage97] Stefan Savage: Eraser: A Dynamic Data Race Detector for Multithreaded Programs ACT Transactions on Computer Systems, Vol. 15, No. 4, Nov. 1997 p391-411

[Shir03] Jack Shirazi: Question of the Month – What does volatile do? http://www.javaperformancetuning.com/news/qotm030.shtml

[Willo04] Willow River Information Systems: Jlint Eclipse-Plugin, http://www.willowriver.net/products/download_jlint.php

[Moser04] Michael Moser: Java Multithreaded Unit (JMTUnit) http://www.michaelmoser.org/jmtunit/

[Edelstein01] O.Edelstein, E.Farchi, Y.Nir, G. Ratsaby, S.Ur - Java Program test generation http://www.research.ibm.com/journal/sj/411/edelstein.pdf