Christian-Albrechts-Universität, 24098 Kiel

Programmierung verteilter Systeme

Thema:

Tools für Java: Debugger und Testumgebung
Überblick über existierende Tools für die Programmiersprache Java

Seminararbeit von Melanie Zessack

Studienfach: Informatik

Betreuer: Frank Huch

Inhaltsverzeichnis

Einleitung

Dieses Seminar behandelt das Thema "Verteilte Systeme". Dabei habe ich mich mit der Programmierung verteilter Systeme in der Programmiersprache Java beschäftigt und nach geeigneten Tools gesucht, um verteilte Programme zu debuggen und zu testen.
Ein Programm ist verteilt, wenn Programmteile bzw. Objekte zur gemeinsamen Kommunikation nicht mehr den normalen Methodenaufruf verwenden können. Es liegt eine logische oder physikalisch räumliche Entfernung zwischen den Objekten vor. In der Programmiersprache Java werden dabei verschiedene Programmteile auf unterschiedlichen Java Virtuel Machines ausgeführt. Ein Beispiel für ein verteiltes Programm ist eine Client-Server-Anwendung. Hierbei kann es vorkommen, dass ein Server zwei Client-Anfragen gleichzeitig bearbeiten muss. Damit ist also die Nebenläufigkeit ein wichtiger Bestandteil verteilter Systeme, denn Nebenläufigkeit ist die tatsächliche oder scheinbare Parallelität von Kontrollflüssen. Nebenläufigkeit birgt allerdings auch neue Fehlerquellen. Der bekannteste Fehler in nebenläufigen Programmen ist der Deadlock.
Daher möchte ich in meiner Seminararbeit im ersten Kapitel kurz auf die Mechanismen eingehen, wie man in Java nebenläufige und verteilte Programmierung realisieren kann.
Im zweiten Kapitel beschäftige ich mich mit Debuggern. Bei der Suche nach geeigneten Tools musste ich feststellen, dass für Java sehr viele Debugger existieren. Viele sind jedoch eingebettet in Entwicklungsumgebungen wie zum Beispiel BlueJ, TogetherJ, Eclipse und Forte. Ich habe mich für meinen Vortrag allerdings für zwei eigenständige Debugger entschieden, die meiner Meinung nach am bekanntesten sind. Weitere Tools, die beim Schreiben von Java-Programmen behilflich sein können, sind Testumgebungen. Abschließend werde ich daher im dritten Kapitel noch einmal auf das Testen von nebenläufigen Programmen eingehen und die Testumgebung JUnit vorstellen.

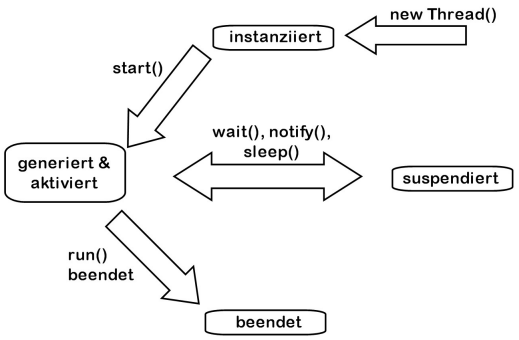
Kapitel 1: Grundlagen

Kapitel 1.1: Threads

Nebenläufigkeit war von Anfang an fester Bestandteil der Sprache Java. Um Nebenläufigkeit zu implementieren stellt Java die Klasse Thread und das Interface Runnable zur Verfügung, die sich beide im package java.lang befinden. Man hat also die Möglichkeit, entweder eine Klasse MyThread von der Klasse `Thread` abzuleiten oder das Interface `Runnable` zu implementieren, wobei auch die Klasse `Thread` bereits `Runnable` implementiert. `Runnable` hat nur die Methode `run()`.

Ein Thread kann vier verschiedene Zustände haben. Zuerst wird er instanziiert. Mit der Methode `start()` wird der Thread gestartet und führt die Methode `run()` aus. Mit `wait()` kann man einen Thread suspendieren, d.h. er befindet sich dann in einer Art "Wartezustand", bis er mit `notify()` wieder aufgeweckt wird. Die Methoden `wait()` und `notify()` sind in der Klasse Object definiert. Auf die Anwendung gehe ich später noch genauer ein. Mit `sleep(long millis)` hat man zusätzlich die Möglichkeit, einen Thread für eine bestimmte Zeit zu suspendieren. Der vierte Zustand ist erreicht, wenn der Thread beendet ist. Dies geschieht dadurch, dass auch die Methode `run()` beendet wurde.

Das eben beschriebene habe ich noch einmal in einer Skizze zusammengefasst:



Als Beispiel habe ich das bekannte Problem der "dinierenden Philosophen" mit Hilfe von Threads implementiert. Dabei sitzen fünf Philosophen, die abwechselnd denken und essen, an einem runden Tisch. Zwischen den Philosophen liegt jeweils ein Stäbchen. Zum Essen benötigen die Philosophen jeweils zwei Stäbchen.

Das Programm besteht aus drei Klassen:

Main: Diese Klasse startet den Prozess der dinierenden Philosophen. Dafür erzeugt sie jeweils 5 Stäbchen-Objekte und 5 Philosophen-Objekte. Außerdem startet sie die Threads, die Philosophen dürfen also anfangen, ihre Handlungen auszuführen.

```
public class Main{

    private Staebchen staebchen1 = new Staebchen();
      ...
    private Staebchen staebchen5 = new Staebchen();
    Philosoph p1 = new Philosoph(staebchen5, staebchen1);
      ...
    Philosoph p5 = new Philosoph(staebchen4, staebchen5);
```

```
    public Main(){
        p1.start();
            ...
        p5.start()
    }
    public static void main(String args[]){
        Main start = new Main();
    }
 }
```

Philosoph: Diese Klasse stellt den eigentlichen Philosophen dar. Da die Klasse Philosoph von `Thread` abgeleitet ist, muss die Methode `run()` überschrieben werden. In der `run()` Methode ist die Reihenfolge der Aktionen des Philosophen festgelegt, die ich auf das Nehmen und Hinlegen des Stäbchens beschränkt habe. Dabei nehmen die Philosophen immer zuerst das linke Stäbchen und legen zuerst das rechte Stäbchen wieder zurück.

```
public class Philosoph extends Thread {
    private Staebchen linkesStaebchen, rechtesStaebchen;

    public Philosoph(Staebchen staebchen1, Staebchen staebchen2){
        linkesStaebchen = staebchen1;
        rechtesStaebchen = staebchen2;
    }
    public void run(){
        while(true){
            //Philosoph denkt nach
            linkesStaebchen.nehmen() ;
            rechtesStaebchen.nehmen();
            //Philosoph isst
            rechtesStaebchen.hinlegen();
            linkesStaebchen.hinlegen();
        }
    }
}
```

Stäbchen: Diese Klasse stellt die Stäbchen dar. Ihre Aufgabe ist es, die Vergabe der Stäbchen zu kontrollieren. Ist das Stäbchen frei, ist also die boolesche Variable `frei` gleich `true`, und ein Philosoph nimmt dieses Stäbchen, wird `frei` auf `false` gesetzt. Wenn jetzt also ein weiterer Philosoph dieses Stäbchen nehmen möchte, muss er warten. Realisiert habe ich dieses mithilfe von zwei synchronized-Methoden, `nehmen()` und `hinlegen()`.

```
public class Staebchen{
    private boolean frei;

    public Staebchen(){
        frei = true;// anfangs ist das Stäbchen frei
    }
    synchronized void nehmen(){
        if (!frei){
            try{this.wait();}
            catch(InterruptedException e){}
        }
```

```
        else
            frei = false;
    }
 synchronized void hinlegen(){
        frei = true;
        this.notify();
    }
    public boolean status(){
        return frei;
    }
}
```

Während die Klassen Main und Philosoph leicht verständlich sind, möchte ich noch einmal genauer auf die Methoden `nehmen()` und `hinlegen()` aus der Klasse Staebchen eingehen. Hierbei handelt es sich nämlich um synchronized-Methoden.

Zur Synchronisation von nebenläufigen Prozessen wird in Java ein Monitor-ähnliches Konzept verwendet. Dabei enthält jedes Objekt eine Sperre (lock), die in Anspruch genommen werden kann, indem man Methoden oder Blöcke als synchronized kennzeichnet. Da jede Klasse ohne explizite Vererbungsanweisung von Object abgeleitet ist, besitzt auch die Klasse Stäbchen diese Sperre. Deswegen kann nur ein Thread (bzw. ein Philosoph) zurzeit die Methode `nehmen()` oder `hinlegen()` in `Staebchen` ausführen. In dem Beispiel reicht es allerdings nicht aus zu verhindern, dass zwei Philosophen gleichzeitig ein Stäbchen nehmen. Auch nachdem ein Philosoph das Stäbchen x genommen hat, darf kein anderer Philosoph dieses Stäbchen x nehmen. Deswegen müssen wir zusätzlich noch die Methoden `wait()` und `notify()` verwenden. Wie bereits erwähnt sind auch diese Bestandteil der Klasse `Object`, und zusätzlich zu der erwähnten Sperre besitzt jedes Objekt auch eine Prozesswarteliste. Dabei handelt es sich um eine (möglicherweise leere) Menge von Threads, die vom Scheduler unterbrochen wurden und auf ein Ereignis warten, um fortgesetzt werden zu können.

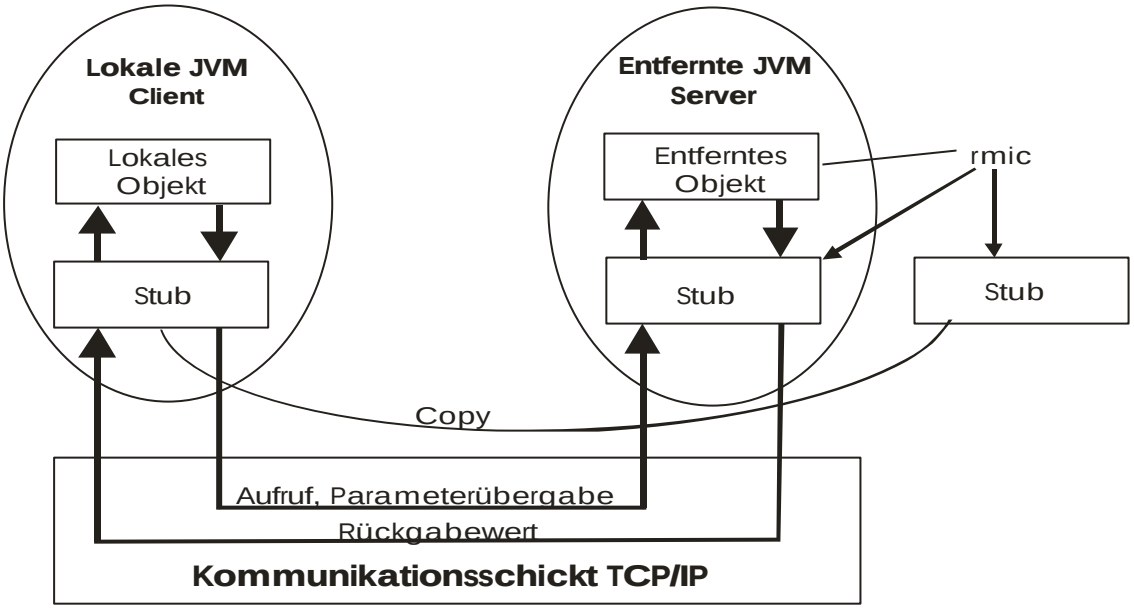
Sowohl `wait()` als auch `notify()` dürfen nur aufgerufen werden, wenn das Objekt bereits gesperrt ist, also nur innerhalb eines synchronized-Blocks oder einer synchronized-Methode für dieses Objekt. Sei o ein Objekt. Dann bewirkt ein Aufruf von `o.wait()`, dass die bereits erzeugte Sperre freigegeben wird und dass der Prozess, der den Aufruf von `wait()` verursachte, in die Warteliste des Objekts gestellt wird. Dadurch wird er unterbrochen, d.h. er befindet sich im suspendierten Zustand. Ein Aufruf von `o.notify()` entfernt einen (beliebigen) Prozess aus der Warteliste des Objekts o, stellt die aufgehobenen Sperren wieder her und führt ihn dem normalen Scheduling zu.

Wie man im Beispiel erkennt, muss bei der Benutzung von `wait()` eine `InterruptException` abgefangen werden. Der Grund dafür liegt darin, dass die Klasse Thread die Methode `interrupt()` enthält. Diese Methode bewirkt bei einem suspendierten Thread, dass dieser wieder aktiviert wird. Da ich diese Methode aber nicht verwende, habe ich die `catch`-Anweisung leer gelassen.

Kapitel 1.2: Remote Message Invocation (RMI)

RMI steht als Abkürzung für Remote Method Invocation und kann als die Weiterentwicklung von Remote Procedure Calls (RPCs) für Java gesehen werden. RMI ist ein Java-Mechanismus, der es erlaubt, Methoden von Objekten aufzurufen, die nicht auf derselben Java Virtuel Machine laufen.

Auf eine genaue Beschreibung der Funktionsweise von RMI möchte ich an dieser Stelle verzichten und auf die Literatur (siehe z.B. [1] oder [9]) verweisen. Die folgende Abbildung zeigt allerdings grob die Vorgehensweise:



Als Beispiel für RMI möchte ich im Folgenden einen Geldautomaten vorstellen, an dem man den Kontostand abfragen und Geld abheben kann:

Zunächst muss dafür ein Interface festgelegt werden, das von java.rmi.Remote abgeleitet ist. Dieses Interface muss von dem Server implementiert werden.

```
import java.rmi.*;
public interface Geldautomat extends Remote {
   public void auszahlen( long betrag )
         throws RemoteException;
   public long kontoStand()
         throws RemoteException;
}
```

Der Server sollte als Unterklasse von java.rmi.server.UnicastRemoteObject abgeleitet werden. Wichtige Aufgabe des Servers ist es, den angebotenen Dienst - d.h. Methodenaufrufe an das Objekt - in der Registry einzutragen, damit entfernte Clients diesen Dienst identifizieren können. Dies erfolgt in der `main`-Methode:

```
import java.rmi.*;
import java.rmi.server.*;
public class GeldautomatServer extends UnicastRemoteObject
      implements Geldautomat {
   private int wert = 150;
   public GeldautomatServer() throws RemoteException { super(); }
   public long kontoStand() throws RemoteException { return wert; }
```

```
   public void auszahlen( long betrag )
         throws RemoteException {
      if (betrag <= wert) wert -= betrag;
   }
   public static void main( String args[] ) {
      try {
         GeldautomatServer s = new GeldautomatServer();
         Naming.rebind( "rmi:///Geldautomat", s);
      }
      catch (Exception e) {
      System.out.println ("GeldautomatServer exception: " + e);
      }
   }
}
```

Der Client ermittelt eine Referenz auf das entfernte Objekt und kann dann die gewünschten Methoden aufrufen. Es wird davon ausgegangen, dass Server und Client in verschiedenen JVM auf einem Rechner laufen.

```
import java.rmi.*;
public class GeldautomatClient {
   public static void main( String args[] ) {
      Geldautomat s;
      try {
          s = (Geldautomat) Naming.lookup( rmi://localhost/Geldautomat" );
          System.out.println( s.kontoStand() );
          s.auszahlen( 100 );
          s.auszahlen( 100 );
      }
      catch (Exception e) {
      System.out.println ("GeldautomatClient exception: " + e);
    }
   }
}
```

Nach der Übersetzung des Servers wird auf der Serverseite das Programm

```
rmic GeldautomatServer.class
```

aufgerufen, um damit die Stub- und Skeleton-Klassen zu erzeugen. Es entstehen die Dateien `GeldautomatServer_Stub.class` und `GeldautomatServer_Skel.class`. Die Stub-Klasse muss dann zum Client kopiert werden.

Soll das Beispiel ausgeführt werden, so muss vor dem Server zunächst die Registry gestartet werden. Serverseitig müssen folgende Aufrufe auf Kommandoebene erfolgen:

```
rmiregistry
java GeldautomatServer
```

Danach kann der Client mit

```
java GeldautomatClient
```

ausgeführt werden.

Kapitel 2: Debugging

Zusätzlich zu den in der sequentiellen Programmierung auftretenden Fehlern gibt es bei der nebenläufigen Programmierung neue Fehler, der bekannteste davon ist der Deadlock. Daher möchte ich am Beispiel der dinierenden Philosophen aufzeigen, wie weit uns ein Debugger behilflich sein kann, diesen Fehler zu analysieren. Zunächst werde ich dabei den jdb, einen kommandozeilenorientierten Debugger verwenden, um später noch einen weiteren Debugger vorzustellen, den JSwat, der im Gegensatz zum jdb eine graphisch Oberfläche besitzt. Ferner werde ich an dem Beispiel des Geldautomaten aus Kapitel 1.2 zeigen, wie einfach man mit beiden Debuggern Remote Objekte untersuchen kann.

Kapitel 2.1: jdb

Der Java-Debugger jdb war von Anfang an festes Bestandteil des JDK. Es gab mit dem ursprünglichen jdb allerdings einige Probleme, weil der Debugger denselben Java-Interpreter benutzt hat wie das zu debuggende Programm. Dieses hatte zur Folge, dass das Programm unter dem jdb in seiner Ausführung beeinflusst wurde. Um dieses zukünftig zu verhindern, wurde die Java Platform Debugger Architecture (JPDA) entwickelt. Die JPDA besteht aus den folgenden drei Komponenten:

- dem Java Virtual Machine Debug Interface (JVMDI),
- dem Java Debug Wire Protocol (JDWP)
- und dem Java Debug Interface (JDI).

und ermöglicht es selber Debugger in Java zu programmieren. Dabei soll der jdb von Sun eine Beispielimplementierung sein. Mit dem JDK erhält man daher nicht nur den Debugger jdb selbst sondern auch seinen Quellcode. Man findet ihn im Verzeichnis demo/jpda unter example.jar, wobei sich die Main-Klasse im package tty befindet und ebenfalls TTY heißt. Weiterhin findet man im package gui die Klasse GUI, die die Main-Methode für den graphischen Debugger javadt von Sun enthält. Da dieser aber nur ein sehr einfacher graphischer Debugger ist und ebenfalls nur eine Beispielimplementierung sein soll, habe ich ihn nicht weiter betrachtet.

Bevor man mit einer jdb-Sitzung beginnt, sollte man die zu debuggende Klasse mit der Option -g kompilieren. Diese bewirkt, dass zusätzliche Informationen (insbesondere die Zeilennummern und die Definitionen lokaler Variablen) in die erzeugten class Bytecodes aufgenommen werden, um damit das Debuggen dieser Klassen zu erleichtern. Um den Debugger zu starten, ruft man jdb mit der Main-Klasse als Argument auf, im Philosophen-Beispiel also:

```
jdb Main
```

Da jdb ebenfalls ein Java-Programm ist, wurde nun ein Java-Interpreter für jdb benutzt. Mit dem Kommando `run` startet man einen zweiten Java-Interpreter, der die Klasse Main ausführt. Nun kann man z.B. mit dem Kommando `threads` sich alle aktuellen Threads anzeigen lassen. Die Ausgabe sieht wie folgt aus:

```
Group system:
  (java.lang.ref.Reference$ReferenceHandler)0xf6 Reference Handler cond. waiting
  (java.lang.ref.Finalizer$FinalizerThread)0xf5  Finalizer         cond. waiting
  (java.lang.Thread)0xf4                         Signal Dispatcher running
Group main:
  (java.util.logging.LogManager$Cleaner)0x13e    Thread-0          unknown
  (Philosoph)0x12f                               Thread-1          cond. waiting
  (Philosoph)0x130                               Thread-2          cond. waiting
  (Philosoph)0x131                               Thread-3          cond. waiting
  (Philosoph)0x132                               Thread-4          cond. waiting
  (Philosoph)0x133                               Thread-5          cond. waiting
  (java.lang.Thread)0x134                        DestroyJavaVM     running
```

Dabei sind die Threads nach Gruppen unterteilt. Wir interessieren uns hier aber nur für die Gruppe main. In der ersten Spalte steht die Thread ID, in der nächsten Spalte der Thread-Name und in der letzten Spalte der Thread-Zustand. Da alle Threads den Zustand `cond. waiting` haben, befinden wir uns bereits im Deadlock.

Nun interessieren wir uns aber dafür, wie wir in den Deadlock gekommen sind. Um die Threads genauer zu untersuchen, suspendiere ich alle Threads mit dem Kommando `suspend`. Mit dem Befehl `thread 0x12f` wähle ich den Thread-1 als aktuellen Thread aus. Hier kann ich mir nun mit `where` den Stack ausgeben lassen. Die Ausgabe sieht wie folgt aus:

```
[1] java.lang.Object.wait (native method)
[2] java.lang.Object.wait (Object.java:426)
[3] Staebchen.nehmen (Staebchen.java:28)
[4] Philosoph.run (Philosoph.java:40)
```

Für die übrigen Threads sehen die Stacks gleich aus. Somit wissen wir genau, an welcher Stelle die Prozesse stehen geblieben sind, nämlich nachdem sie das linke Stäbchen genommen haben. Leider bleibt uns aber die Reihenfolge verborgen, in der die Philosophen die Stäbchen aufgenommen haben. Hier bietet sich die Möglichkeit an, am Anfang der `run()`- Methode in der Klasse Philosoph einen Breakpoint zu setzen und dann im Einzelschrittmodus weiterzumachen. Dabei kann man das Scheduling selber bestimmen, indem man alle Threads bis auf einen suspendiert.

Für das Setzen von Breakpoints bietet der jdb zwei Alternativen an:

- `stop at Main:22` (setzt den Breakpoint auf die erste Instruktion für Zeile 22 im Quellcode der Klasse Main)
- `stop in Main.method` (setzt den Breakpoint auf den Beginn der Methode method)

Mit dem Kommando `clear Main:22` entfernt man den in Main auf Zeile 22 gesetzten Breakpoint. Gibt man nur `clear` ein, erhält man eine Liste aller gesetzten Breakpoints. Mit `step` und `next` hat man die Möglichkeit das Programm im Einzelschrittmodus auszuführen. Der Unterschied zwischen den beiden Methoden liegt im Methodenaufruf. `step` bietet die Möglichkeit eine aufgerufene Methode ebenfalls im Einzelschrittmodus ausführen zu lassen, während `next` die ganze Methode ausführt.

Weiterhin ist noch zu erwähnen, dass der jdb natürlich noch wesentlich mehr Funktionen bietet. Daher möchte ich hier noch einmal die wichtigsten Kommandos aufführen. Eine ausführliche Liste erhält man, indem man in einer jdb-Sitzung `help` eingibt.

Basis-Kommandos des jdb:

`help` Das wichtigste jdb-Kommando, man erhält eine Liste der zur Verfügung stehenden Kommandos mit einer kurzen Beschreibung.

`run` Nachdem man jdb gestartet hat und notwendige Breakpoints gesetzt hat, kann man mit diesem Kommando die Ausführung der Applikation starten.

`cont` Setzt die Ausführung nach einem Haltepunkt oder einer Ausnahme fort.

`print` Zeigt den Inhalt der Variablen an, die als Argument übergeben wurde. print verwendet dazu die Methode toString, die in allen Objekten implementiert ist. Damit eine Variable angezeigt werden kann, muss sie an der Aufrufstelle sichtbar sein.

`dump` Zeigt alle Informationen über ein Objekt an, sofern dieses an der Aufrufstelle sichtbar ist.

`threads` Zeigt alle aktuellen Threads auf, wobei jeweils der Name und der aktuelle Status ausgegeben werden, wie auch der Index, den man für weitere Kommandos benötigt

`thread` Wird verwendet, um einen Thread zu selektieren und zwar mit Hilfe der Indizes, die man mit dem threads Kommando erhält

`where` Zeigt die aktuelle Methodenaufruf-Hierarchie an.

`quit` Beendet den Debugger und die Ausführung des Programms

Eine wichtige Eigenschaft des jdb ist außerdem, dass man eine bereits laufende Java-Applikation debuggen kann. Dafür muss man das zu debuggende Programm (z.B. GeldautomatServer) wie folgt starten:

```
java –Xdebug
     -Xrunjdwp:transport=dt_socket,server=y,suspend=n GeldautomatServer
```

Man erhält daraufhin einen Port, hier 1253:

```
Listening for transport dt_socket at address: 1253
```

Den jdb starte ich dann wie folgt:

```
jdb -connect com.sun.jdi.SocketAttach:port=1253,hostname=localhost
```

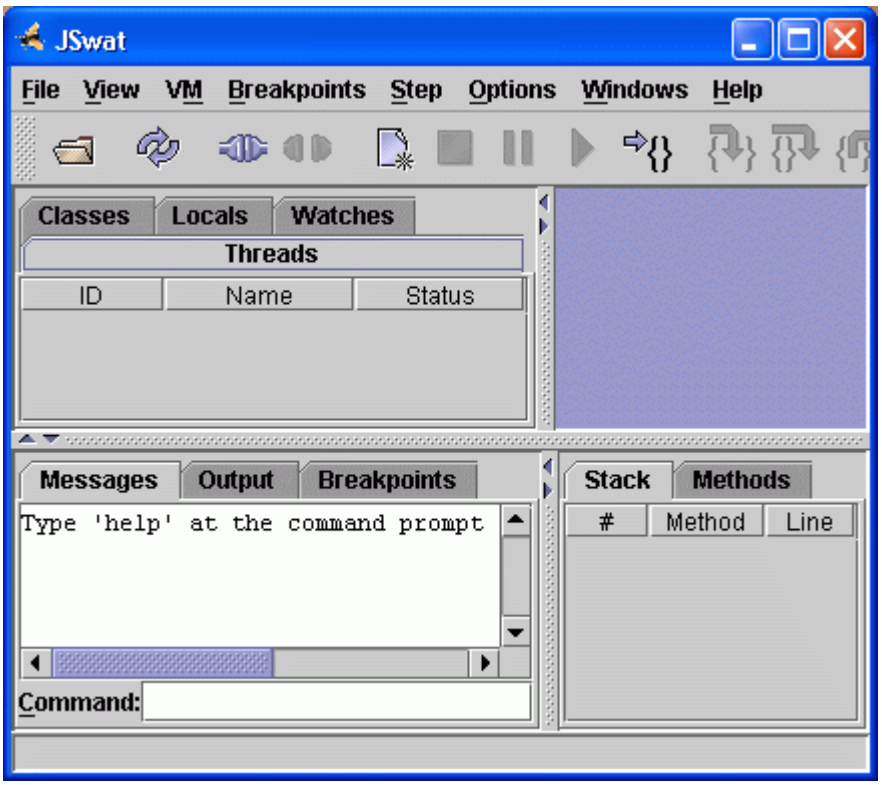
Das Beispiel des Geldautomaten werde ich im Folgenden mit dem Programm JSwat debuggen, das im Grunde die gleichen Funktionen hat wie der jdb, durch seine graphische Oberfläche allerdings viel komfortabler und einfacher zu bedienen ist.

Kapitel 2.2: JSwat

JSwat ist ein in Java geschriebener graphischer Debugger für Java-Programme, der wie der jdb die Java Platform Debugger Architektur (JPDA) verwendet. JSwat wurde bereits 1999 von Nathan Fiedler entwickelt und ständig aktualisiert. Für diesen Seminarvortrag habe ich die Version JSwat 2.16 verwendet, für die man das JDK 1.4 benötigt. JSwat ist im Netz frei verfügbar (siehe Literaturverzeichnis [5]). Beim Starten des Programms ist zu beachten, dass der Debugger die Java Platform Debugger Architektur verwendet und damit eine weitere Angabe darüber benötigt, wo die tool.jar Datei zu finden ist. Daher sollte man JSwat mit folgendem Kommando starten:

```
java –Djava.ext.dirs=<Java_Home>/lib –jar <JSwat_Home>/jswat.jar
```

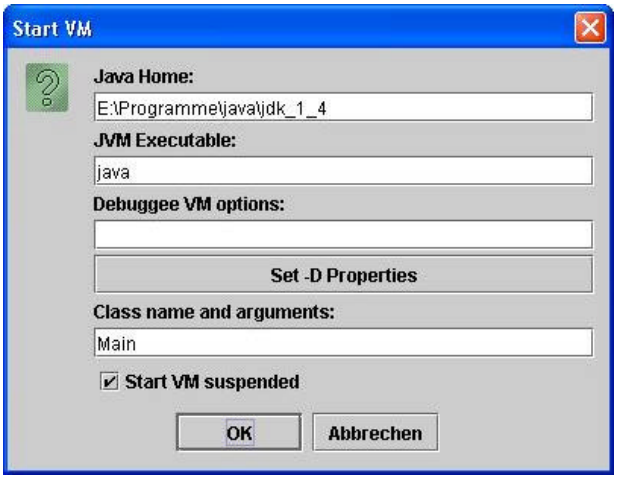
Dabei sollte <Java_Home> das Verzeichnis enthalten, in dem das JDK installiert ist, und <JSwat_Home> ist das Verzeichnis, das die jswat.jar Datei enthält. Nach dem Ausführen des Kommandos erhält man folgendes Fenster:



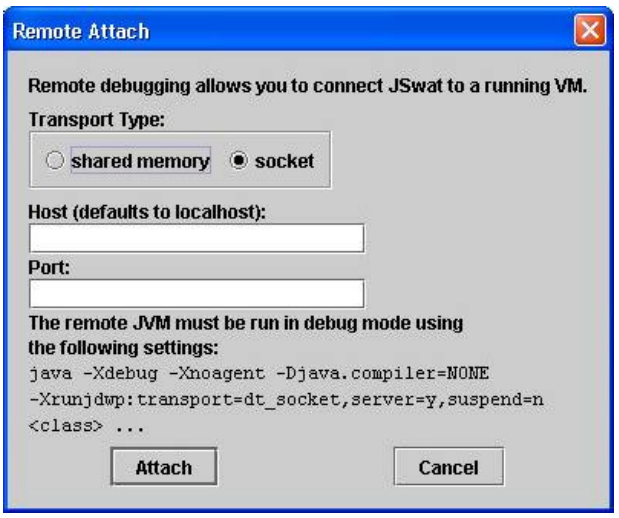
Wie man hier erkennt, ist JSwat viergeteilt. Links oben befinden sich Informationen über Threads, Klassen usw. während man sich rechts daneben den Sourcecode anzeigen lassen kann. Links unten sieht man einerseits Nachrichten über das Debuggen selber, wobei man auch hier Kommandos eingeben kann. Damit man weiß, wie die Befehle auszusehen haben, erhält man eine ausführliche Hilfe, wenn man das Kommando `help` eingibt. Unter dem Reiter "Output" sieht man die Programmausgaben, die sonst in der Konsole erscheinen. Wartet das Programm auf Benutzereingaben, so müssen diese auch hier eingegeben werden. Unter dem dritten Reiter "Breakpoints" sieht man schließlich, welche Haltepunkte gesetzt wurden. Rechts unten befindet sich der Stack-Frame und eine Liste aller definierten Methoden. Die einzelnen Teile kann man in ihrer Größe beliebig verändern.

Bevor man mit einer Sitzung beginnt, muß man noch den Classpath einstellen. Dies geschieht unter Options -> Set Classpath. Zum Glück werden diese Einstellungen auch gespeichert, so dass man sie in der nächsten Sitzung wieder verwenden kann. Dann kann man den Quellcode laden, der in der rechten oberen Ecke angezeigt wird.

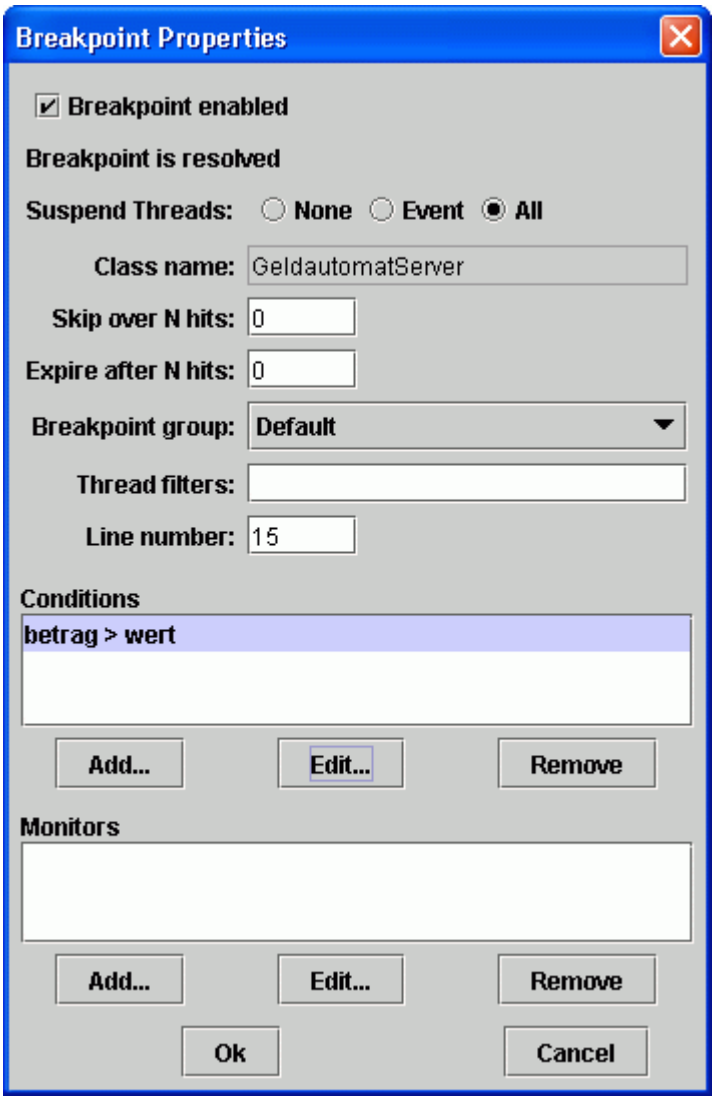
Den Java-Interpreter kann man entweder durch Mausklick auf das entsprechende Icon in der oberen Zeile oder unter VM -> Start VM starten. Auch eine Kommandoeingabe ist möglich.
Man erhält folgendes Fenster:



Im Folgenden möchte ich das Beispiel des Geldautomaten behandeln. Dafür habe ich den GeldautomatenServer wie in Kapitel 3.1 beschrieben gestartet und starte den Debugger mit dem Befehl Remote Attach, den man ebenfalls unter VM findet. Hier muss dann nur noch der Port eingegeben werden.



Bevor ich den Client von einer anderen Konsole aus starte, setze ich in der Methode `auszahlen(long betrag)` des GeldautomatenServer einen Breakpoint in die erste Zeile. Für diesen Breakpoint können wir einige Eigenschaften einstellen:



Man kann bestimmen, ob die Threads beim erreichen des Breakpoints suspendiert werden sollen oder nicht. Unter "Skip over N hits" kann man eingeben, wie oft der Breakpoint ignoriert werden soll, während "Expire after N hits" bedeutet, dass der Breakpoint nach N Mal nicht mehr genutzt wird. Ist die Zahl unter Skip größer als die Zahl unter Expire, wird der Breakpoint also ignoriert.
Für das Beispiel des Geldautomaten habe ich eine Bedingung für den Breakpoint eingegeben. Ich interessiere mich nämlich nur für den Fall, dass jemand mehr Geld abheben möchte, als auf dem Konto vorhanden ist, d.h. der Breakpoint ist nur gültig, wenn `wert > betrag` ist. Das Feld Conditions darf allgemein nur boolesche Ausdrücke enthalten. Wird dieser Ausdruck zu true ausgewertet, wird das Programm angehalten, ansonsten wird der Breakpoint ignoriert. Die Syntax dieser Ausdrücke ist gleich der Syntax in Java, allerdings mit folgenden Ausnahmen:

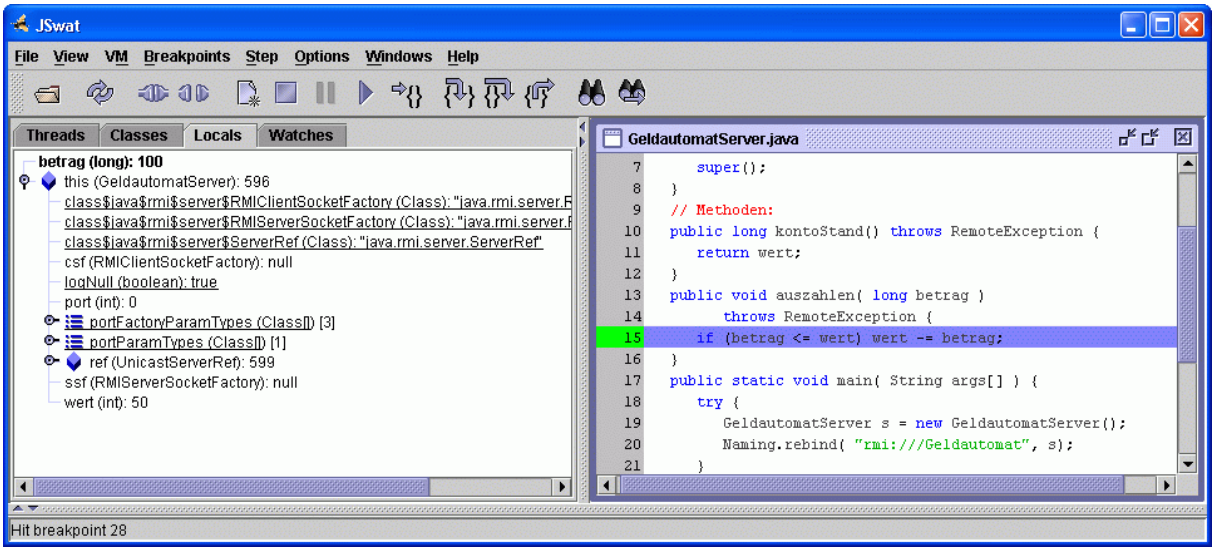
- Bedingungsoperatoren sind (noch) nicht erlaubt
- "Type casting" wird (noch) nicht unterstützt
- Das Schlüsselwort "super" darf (noch) nicht benutzt werden
- Ausdrücke wie "abc".length() werden nicht unterstützt

Anwendbare Operatoren sind:

- - (unär), + (unär), !, ~
- *, /, %
- +, -
- <<, >>, >>>
- <, >, <=, >=
- ==, !=
- &
- ^
- |
- &&
- ||

Es ist auch möglich für einen Breakpoint "Monitors" einzugeben. Ein "Monitor" ist dabei eine Aktion, die ausgeführt wird, wenn der Breakpoint erreicht wird. Zurzeit wird nur ein Typ von Monitoren unterstützt, nämlich der sogenannte "Command Monitor". Dabei ist die Syntax gleich der Syntax, die in der Kommandozeile von JSwat benutzt wird.

Zurück zum Beispiel: Versucht der Client das zweite Mal Geld abzuheben, ist der Breakpoint erreicht, und der Server unterbricht seine Ausführung. Hier habe ich jetzt die Möglichkeit, die Variablen zu untersuchen und evtl. zu ändern.



Ich gebe also in die Befehlszeile `set betrag = 50` ein, und damit kann unser Client 50 Euros abheben, die er sonst nicht bekommen hätte. Nun kann man das Server-Programm entweder weiter im Einzelschrittmodus verfolgen oder mit `resume` wieder normal ablaufen lassen.
Alternativ hätte man das Kommando "`set betrag = 50`" auch als Monitor zum Breakpoint eingeben können.

Kapitel 3: Testumgebungen

Kapitel 3.1: Allgemeine Tests

Eine Art von Tests, die wir zeitnah zur Programmierung erstellen und nach jeder Programmmodifikation ausführen wollen, sind Unit Tests. Unit Tests sind in den überwiegenden Fällen White-Box Tests, das heißt, wir testen das Programm, indem wir auch die innere Programmstruktur berücksichtigen. Um einen Unit Test durchzuführen benötigen wir Testwerte bzw. Testfälle. Dabei ist es nicht immer einfach gute Testfälle zu konstruieren. Wichtig ist dabei, dass man positive, Grenz- und negative Tests durchführt. Bei einem Programm zur Berechnung der Wurzel wählt man z.B. als Testwerte 2, 0 und -2. Bei dem ersten Test erwartet man auf jeden Fall ein vernünftiges Ergebnis, beim zweiten Fall testet man sein Programm auf Grenzfälle und beim dritten Test überprüft man, ob das Programm einen nicht gültigen Wert auch ablehnt.
Natürlich ist man auch daran interessiert, dass das Programm richtig rechnet, also sollte man in einem Test Abfragen einbauen, ob das richtige Ergebnis geliefert wird. Dies ist vor allem dann wichtig, wenn man die Berechnung nicht mehr von Hand nachvollziehen kann.
Dabei gibt es natürlich verschiedene Möglichkeiten die Tests durchzuführen. Eine Möglichkeit wäre, sich die Ergebnisse von Tests in der Konsole durch „System.out.println(String)“ ausgeben zu lassen. Dies hat aber den Nachteil, dass man alle diese Nachrichten nach erfolgreichem Testen wieder aus dem Programm entfernen muss. Tritt später an anderer Stelle erneut ein Fehler auf, muss man sie evtl. nachträglich wieder einfügen. Eine Alternative dazu bietet Java ab der Version 1.4 mit der Klasse Logger. Die einfachste Benutzung sieht wie folgt aus:

```
Logger logger = Logger.getLogger(„global“);
```

Dann können wir unsere `System.out.println(String)`-Befehle durch `logger.info(String)` ersetzen. Der Vorteil dabei ist, dass mit dem Aufruf `logger.setLevel(Level.OFF)` die Ausgabe der Nachrichten ausstellen kann.

Um das Durchführen von Tests zu vereinfachen gibt es aber auch Testumgebungen. Eine dieser Testumgebungen ist JUnit, die ich im Folgenden näher beschreiben möchte.

Kapitel 3.2: JUnit

JUnit ist ein kleines, mächtiges Java-Programm zum Schreiben und Ausführen automatisierter Unit Tests. Die Software ist frei und im Kern von Kent Beck und Erich Gamma geschrieben. JUnit ist als kostenloser Download im Internet verfügbar (siehe Literaturverzeichnis [6]).
Für meinen Seminarvortrag habe ich JUnit in der Version 3.8.1 getestet. Die vollständige JUnit-Distribution besteht zur Zeit aus einem ZIP-Archiv, in dem neben dem Test-Framework (in junit.jar) auch seine Quellen (in src.jar), seine Tests, einige Beispiele, die Javadoc-Dokumentation, die FAQs, ein "Kochbuch" und zwei Artikel beiliegen.

Um eine Klasse mit JUnit zu testen, definieren wir uns eine weitere Klasse, die so genannte Testklasse. Diese Testklasse muss von junit.framework.testCase abgeleitet werden. Weiterhin muss diese Klasse eine main-Methode enthalten, die JUnit aufruft. Dabei können wir 3 verschiedene Arten unterscheiden. Zunächst gibt es bei JUnit eine textbasierte Testumgebung. Der Aufruf dafür mit

```
junit.textui.TestRunner.run(Testklasse.class);
```

Die nächsten beiden Möglichkeiten rufen beide eine graphische Testumgebung auf, wobei die eine auf AWT und die andere auf SWING aufbaut. Die Aufrufe sehen wie folgt aus:

```
junit.awtui.TestRunner.run(Testklasse.class);
junit.swingui.TestRunner.run(Testklasse.class);
```

Bevor man die Testklasse kompiliert und ausführt, sollte man daran denken, den Classpath auf das junit.jar Verzeichnis zu setzen.

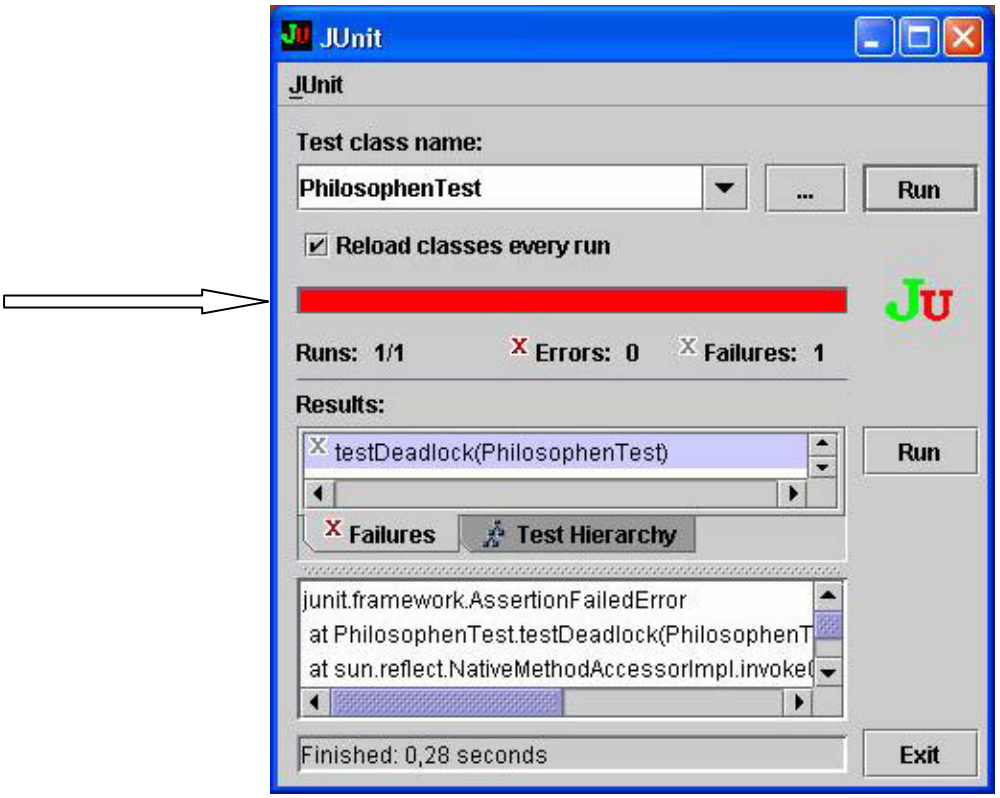
Des Weiteren muss die Testklasse mindestens eine Methode besitzen, deren Name mit "test" anfängt. Diese Methode soll die eigentlichen Tests für unser Programm enthalten. JUnit stellt für diese Tests so genannte "assert"-Methoden aus der Klasse Assert zur Verfügung. Da TestCase von Assert abgeleitet ist, erben wir diese Methoden auch in unserer Testklasse. Die wichtigsten Methoden sind:

- `assertTrue(boolean condition)`
  verifiziert, ob eine logische Bedingung wahr ist
- `assertFalse(boolean condition)`
  verifiziert, ob eine logische Bedingung falsch ist
- `assertEquals(Object expected, Object actual)`
  verifiziert, ob zwei Objekte gleich sind. Der Vergleich der Objekte erfolgt dabei über die equals-Methode
- `assertNull(Object object)`
  sichert zu, dass eine Objektreferenz null ist
- `assertNotNull(Object object)`
  sichert zu, dass eine Objektreferenz nicht null ist

Als Beispiel habe ich wieder das Problem der dinierenden Philosophen genommen. Hierzu möchte ich eine Testmethode schreiben, die überprüfen soll, ob mein Programm einen Deadlock besitzt. In einem Deadlock befindet man sich, wenn kein Thread mehr aktiv ist. Leider gibt es bei der Klasse Thread keine Abfrage, ob ein Thread suspendiert ist oder nicht. Daher muss man sich überlegen, wann dieser Fall eintreten könnte. In dem Programm der dinierenden Philosophen tritt er ein, wenn jeder Philosoph bereits ein Stäbchen besitzt und auf das zweite wartet. Daher habe ich in der Klasse Philosoph eine boolesche Variable `moechteZweitesStaebchenNehmen` eingeführt. Ist diese Variable für alle Philosophen `true`, befinden wir uns im Deadlock. Damit sieht meine Testklasse wie folgt aus:

```
import junit.framework.*;
public class PhilosophenTest extends TestCase{
     public void testDeadlock(){
          Main main = new Main();
          Philosoph p1 = main.p1;
          …
          Philosoph p5 = main.p5;
          int i;
          while(true){
               boolean b1 = p1.moechteZweitesStaebchenNehmen;
               …
               boolean b5 = p5.moechteZweitesStaebchenNehmen;
               boolean evtlDeadlock = b1 & b2 & b3 & b4 & b5;
               assertFalse(evtlDeadlock);
          }
     }
     public static void main(String args[]){
          junit.swingui.TestRunner.run(PhilosophenTest.class);
     }
}
```

JUnit liefert das folgende Ergebnis:



Der Balken, der sich in der Höhe des Pfeils befindet, verfärbt sich rot, und wir erhalten einen Hinweis, dass unser Test fehlgeschlagen ist. Also muss das Programm noch einmal abgeändert werden. Ich erweitere nun die Methode run() der Klasse Philosoph, so dass die Philosophen das erste Stäbchen wieder zurücklegen, wenn das zweite nicht frei ist:

```
public void run(){
   while(true){
      //Philosoph denkt nach
      linkesStaebchen.nehmen();
     while(!rechtesStaebchen.status()){
         linkesStaebchen.hinlegen();
         linkesStaebchen.nehmen();
      }
      moechteZweitesStaebchenNehmen=true;
      rechtesStaebchen.nehmen();
      moechteZweitesStaebchenNehmen=false;
      //Philosoph isst
      rechtesStaebchen.hinlegen();
      linkesStaebchen.hinlegen();
   }
}
```

Lassen wir JUnit nun erneut laufen, erhalten wir keinen roten Balken. Normalerweise verfärbt sich der Balken grün, wenn die Tests richtig verlaufen. Da sich dieser Test aber in einer Endlosschleife befindet, kann die Methode testDeadlock nicht beendet werden. Wir können aber davon ausgehen, dass sich dieses Programm nicht im Deadlock befindet, solange der Balken sich nicht rot färbt. Damit können wir allerdings nicht verifizieren, dass unser Programm keinen Deadlock besitzt.

Hat man mehrere Testklassen implementiert, ist es auch möglich, die Tests in einer so genannten `TestSuit` zusammenzufassen. Damit kann man die Tests gemeinsam ausführen und muss nicht jeden Test einzeln starten. Die Syntax der TestSuit sieht wie folgt aus:

```
import junit.framework.*;
public class AllTests {
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(Testklasse1.class);
        ...
        suite.addTestSuite(TestklasseN.class);
        return suite;
    }
    public static void main(String[] args) {
        junit.swingui.TestRunner.run(AllTests.class);
    }
}
```

Etwas verwunderlich ist vielleicht, dass die Methode `suite()` ein Objekt vom Typ `Test` zurückgibt. Dies lässt sich aber dadurch erklären, dass `Test` eine Schnittstelle ist, die sowohl von `TestSuite` wie auch von `TestCase` implementiert wird.

Aufbauend auf JUnit gibt es eine ganze Reihe von Projekten, die seine Funktionalität erweitern oder mittels JUnit auch weitergehende Aufgaben lösen. Da diese aber so zahlreich sind, dass ich sie hier nicht alle aufführen kann, verweise ich an dieser Stelle auf die Literatur [7].

Die Integration in verschiedene IDEs wird ebenfalls unterstützt. Beides - Erweiterung und Integration - sind Stärken von Open-Source-Projekten. Bei kommerziellen Produkten ist das zwar nicht ausgeschlossen, oftmals aber vom Willen der Firma abhängig, zumindest Schnittstellen zu veröffentlichen.

Zusammenfassung

JUnit ist ein Tool, mit dem man seine eigene Arbeit ständig kontrollieren kann. Dabei ist aber die Empfehlung der Entwickler von JUnit zu beachten, die besagt, dass man die Tests vor dem eigentlichen Programm schreiben sollte. JUnit kann uns dann dabei unterstützen von Anfang an richtig zu programmieren, wodurch JUnit für verteilte Programmierung genauso hilfreich ist wie für nicht verteilte Systeme. Taucht in unserem Programm allerdings ein Fehler auf, den wir nicht weiter analysieren können, weil wir von JUnit nur die Zeilenangabe erhalten, empfehle ich, zusätzlich einen Debugger zu verwenden. Dabei ist es jedem selbst überlassen, ob er eher einen kommandozeilenorientierten Debugger wie jdb oder einen graphischen Debugger wie JSwat bevorzugt. Wie man an dem Beispiel des Geldautomaten erkennt, ist es kein Problem, verteilte Systeme zu debuggen. Dies beruht auf der von Sun entwickelten Java Platform Debugger Architecture (JPDA), die von beiden von mir vorgestellten Debuggern benutzt wird.
Bei der Analyse von Deadlocks dagegen, würde man sich mehr Unterstützung eines Debuggers wünschen. Es wäre zum Beispiel hilfreich, wenn man nach dem Erreichen eines Deadlocks das zuletzt ausgeführte Scheduling noch einmal zurückverfolgen könnte.

An dieser Stelle möchte ich mich noch bei meinem Betreuer, Herrn Frank Huch, herzlich für seine Unterstützung bedanken.

Literaturverzeichnis

[1] Marko Boger, Java in verteilten Systemen

[2] Ralph Steyer, Java 2

[3] Guido Krüger, Handbuch der Java-Programmierung

[4] http://java.sun.com/products/jpda/doc/soljdb.html

[5] http://www.bluemarsh.com/java/jswat

[6] http://www.junit.org

[7] http://www.junit.org/news/extension/index.htm

[8] Frank Westphal, Testgetriebene Entwicklung

[9] http://www.inf.fh-dortmund.de/personen/professoren/achilles/Vorlesung/pg/Vorlesung/RMI.html