ALP IV: Übung 7

Tutor: Christoph Beuck

Lisa Dohrmann (4130066), Adrian Neumann (4140810), Naja v. Schumde (4127652)

24. Juni 2008

Aufgabe 1

Um das Master-Worker verhalten in Java zu implementieren, benötigt man die Klassen Worker, Master, Solution und Problem. Dabei werden alle Klassen mit `abstract` versehen, denn diese Klassen stellen lediglich das Grundgerüst dar, jedes spezielle Problem muss aber natürlich individuell angepasst werden.

```
1  package alpiv.uebung7;
2
3  import java.util.ArrayList;
4
5  public abstract class Master<P extends Problem, S extends Solution>
       extends Thread{
6
7    protected ThreadBag<S> solutions;
8    protected ThreadBag<P> problems;
9    protected ArrayList<S> sortedSolutions;
10
11   public Master(ThreadBag<S> s, ThreadBag<P> p, P prob) {
12     this.solutions = s;
13     this.problems = p;
14     sortedSolutions = new ArrayList<S>();
15     problems.add(prob);
16   }
17
18   /**
19    * Solange wie wir noch nicht fertig sind, soll immer schön kombiniert
          werden.
20    */
21   public void run() {
22     while (!S.isSolved) {
23       try {
24         S sol = solutions.remove();
25         System.out.println("Hole Lösung");
26         combine(sol);
27       } catch (InterruptedException e) {
28         e.printStackTrace();
29       }
30     }
31     finishing();
32   }
33
34   /**
35    * Kombiniert die neue Lösung mit den anderen, so dass alles in die
          richtige Reihenfolge kommt.
36    * Dafür gibts die sortedSolution ArrayList, in der sie sortiert sind
37    * @param solution
38    */
39   public abstract void combine(S solution);
40
41   /**
42    * Wenn noch irgendwas ganz am Ende, nachdem alle Lösungen vorhandne
          sind, ausgeführt werden soll
```

```
43      */
44    public abstract void finishing();
45  }
```

Der Master hat zugriff auf die bereits gelösten Probleme und die noch offenen, welche in ThreadBag-Objekten verwaltet werden. Ganz zu beginn wird das Ursprungsproblem zu `problems` hinzugefügt, dann beginnt in `run()` die eigentliche Arbeit. Solange das ganze noch nicht gelöst ist, werden Lösungen aus `solutions` genommen und per `combine(sol)` mit der bereits vorhandenen Teillösung kombiniert. Es kann sein, dass noch zum Schluss irgendwelche aufräumarbeit o.ä. verrichten werden muss, daher gibt es noch die Methode `finishing()` die zum Abschluss von `run()` aufgerufen wird.
Da das Kombinieren und Abschließen problemspezifisch ist, sind die Methoden `abstract`, so dass jede Unterklasse sie implementieren muss.

```
1   package alpiv.uebung7;
2
3   public abstract class Worker<P extends Problem, S extends Solution>
        extends Thread {
4
5     protected ThreadBag<S> solutions;
6     protected ThreadBag<P> problems;
7
8     public Worker(ThreadBag<S> s, ThreadBag<P> p) {
9       this.solutions = s;
10      this.problems = p;
11    }
12
13    /**
14     * Problem holen, und entscheiden ob wir es direkt lösen können, oder ob
            es halbiert werden muss
15     */
16    public void run() {
17      while(!S.isSolved) { // solange das Problem nicht gelöst ist, arbeiten
            ...
18        try {
19          P prob = problems.remove();
20          if(prob.getIsSimple())
21            solve(prob);
22          else
23            divide(prob);
24        } catch (InterruptedException e) {
25          e.printStackTrace();
26        }
27      }
28    }
29
30    /**
31     * Problem lösen
32     * @param problem
33     */
34    public abstract void solve(P problem);
35
36    /**
37     * Problem in zwei oder mehr Teilprobleme spalten und wieder problems
            hinzufügen
38     * @param problem
39     */
40    public abstract void divide(P problem);
41  }
```

Der Aufbau der Worker ist ähnlich zu dem des Masters. Hier werden solange das Problem noch nicht gelöst ist, in der `run()`-Methode immer neue Probleme aus `problems` geholt, dann entschieden, b sie klein genug sind, dass man sie direkt lösen kann, oder ob sie noch weiter geteilt werden müssen. Genauso wie beim Master sind hier `solve(P problem)` und `divide(P problem)` problemabhängig und daher wieder `abstract`.

```
1   package alpiv.uebung7;
```

```
2
3  public abstract class Problem {
4
5    protected boolean isSimple;
6
7    public boolean getIsSimple() {
8      return isSimple;
9    }
10 }
11
12 package alpiv.uebung7;
13
14 public abstract class Solution {
15
16   /**
17    * Gibt an, ob das Gesamtproblem schon gelöst ist.
18    */
19   public static boolean isSolved = false;
20
21 }
```

Problem und Solution sind nur Grundgerüste, die lediglich `isSimple` bzw. `isSolved` bereitstellen. Alle übrigen Dinge müssen sich selber ausgedacht werden.

Aufgabe 2

Als erstes muss man sich natürlich überlegen, wie genau ein Problem definiert wird. Und zwar ist das in diesem Fall das bestimmte Intervall an Zeilen, die durch einen Worker gerendert werden sollen. Implementiert wird also die Klasse `ProblemRaytrace`, die von `Problem` erbt. Wir passen sie nun so an, dass sie über zwei Objektvariablen `windowSize` und `windowStart` verfügt, die eben dieses Fenster definieren.

```
1    public ProblemRaytrace(int winSize, int winStart) {
2      this.windowSize = winSize;
3      this.windowStart = winStart;
4      if(windowSize <= 40)
5        isSimple = true;
6    }
```

Das spannendste ist auch der Konstruktor, in dem nun noch getestet werden kann, ob ein Problem schon einfach genug ist.
Analog dazu kann man sich nun die `SolutionRaytrace`-Klasse überlegen. In dieser Klasse wird neben eben dieser Fenstergröße wie in der Problemklasse die Ausgabe des Renderns gespeichert; dafür braucht man einmal den `ByteArrayOutputStream` und das entsprechende `RendererFrontend`-Objekt. In der Klasse passiert auch nix weiter, als den Zugriff auf die Objektvariablen zu gewährleisten.
Als nächstes folgt die Erläuterung der `MasterRaytrace`-Klasse. Wie bereits in erstens erwähnt, muss `combine` und `finishing` selbst implementiert werden.

```
1  public void combine(SolutionRaytrace sol) {
2    System.out.println("Combine");
3    int solStart = sol.getWindowStart();
4    int lines = 0;
5
6    if (sortedSolutions.size() == 0) // Liste ist noch leer, also einfach
       rein damit
7      sortedSolutions.add(sol);
8    else { // ansonsten suchen wir die richtige Position ...
9      for (int i = 0; i < sortedSolutions.size(); i++) {
10       SolutionRaytrace solI = sortedSolutions.get(i);
11       if (solI.getWindowStart() + solI.getWindowsSize() <= solStart)
12         continue;
13       else {
14         System.out.println("neue␣Lösung␣einfügen");
```

```
15          sortedSolutions.add(i, sol);
16          solI = sol;
17          break;
18        }
19      }
20    }
21
22    // Jetzt noch ausrechnen, wie viele Lines schon komplett sind
23    for (int i = 0; i < sortedSolutions.size(); i++) {
24      lines += sortedSolutions.get(i).getWindowsSize();
25    }
26    System.out.println(this.getName() + ": Momentan fertige Lines: " + lines
        );
27    if (lines == 800) { // bei 800 sind wir fertig!
28      System.out.println("Fertig!");
29      SolutionRaytrace.isSolved = true;
30    }
31  }
```

Wir bekommen die nächste Lösung übergeben und schauen nun nach, ob es die erste Lösung ist, die vorliegt, oder ob wir darauf achten müssen, an welche Stelle wir die Lösung in unsere bereits sortierten Lösungen einfügen müssen. Die bereits sortierten Lösungen werden in `sortedSolutions` gehalten. Wir sortieren aufsteigend nach dem Startpunkt des Renderfensters. Zusätzlich überprüfen wir, ob wir nach dem Einsortieren nun alle Lösungen beisammen haben, wenn das der Fall ist, dann können wir nämlich aufhören und setzen daher `SolutionRaytrac.isSolved` auf `true`. Die Methode `finishing()` ist nun dafür zuständig, alle `BufferedArrayOutputStream`s der Lösungen richtig in die Ausgabedatei zu schreiben.
Jetzt zu der `WorkerRaytrace`-Klasse ...

```
1  public void solve(ProblemRaytrace prob) {
2    ByteArrayOutputStream baos = new ByteArrayOutputStream();
3    RendererFrontend rf= new RendererFrontend(baos);
4    rf.setWindowStrip(prob.getWindowStart(), prob.getWindowStart() + prob.
       getWindowSize());
5      rf.render();
6    // add to solutions
7    SolutionRaytrace sr = new SolutionRaytrace(prob.getWindowSize(), prob.
       getWindowStart(),baos, rf);
8    System.out.println(this.getName() + ": Problem gelöst für " + prob.
       getWindowStart() + " Größe " + prob.getWindowSize());
9    solutions.add(sr);
10 }
11
12 public void divide(ProblemRaytrace prob) {
13   int newWinSize = prob.getWindowSize() / 2;
14   System.out.println(this. getName() + ": Neue WinSize: " + newWinSize);
15   ProblemRaytrace prob1 = new ProblemRaytrace(newWinSize, prob.
       getWindowStart());
16   ProblemRaytrace prob2 = new ProblemRaytrace(newWinSize, prob.
       getWindowStart() + newWinSize);
17   problems.add(prob1);
18   problems.add(prob2);
19 }
```

Hier wieder am interessantesten die `solve` und `divide` Methoden. Wir holen uns also zunächst wie erwähnt in `run()` das nächste Problem. Wenn es die richtige Größe hat, wird es per `solve()` gelöst. Es wird also ein neues `RendererFrontend` Objekt erstellt mit der passenden Fenstergröße und dann gerendert. Das fertige packen wir dann einfach in eine neue `SolutionRaytrace`. Das Telen geht noch einfacher. Man teilt einfach die Fenstergröße durch zwei und bastelt sich mit den neuen Werten zwei neue Probleme, die wieder in den Problempool kommen.

Aufgabe 3