

RMI – PI CALCULATOR

SYT / DEZSYS07



8. Januar 2015

Erceg, Kritzl

4AHITT

Inhalt

[1. Aufgabenstellung 2](#_Toc408413085)

[2. Requirementanalyse 3](#_Toc408413086)

[3. detaillierte Arbeitsaufteilung mit Aufwandsabschätzung 4](#_Toc408413087)

[3.1 Aufwandabschätzung 4](#_Toc408413088)

[2.2 Arbeitsaufteilung für die Implementierung des Programms 4](#_Toc408413089)

[2.2.1 Package „control“ 4](#_Toc408413090)

[2.2.2 Package „algorithm“ 4](#_Toc408413091)

[2.2.3 Package „components“ 4](#_Toc408413092)

[2.2.4 Package „service“ 4](#_Toc408413093)

[4. anschließende Endzeitaufteilung 5](#_Toc408413094)

[4.1 Erceg 5](#_Toc408413095)

[4.2 Kritzl 5](#_Toc408413096)

[4.3 Gesamtsumme 5](#_Toc408413097)

[5. Designüberlegung 6](#_Toc408413098)

[5.1 Abbildung 6](#_Toc408413099)

[5.2 Überlegungen zur Struktur 7](#_Toc408413100)

[5.2.1 Package „control“ 7](#_Toc408413101)

[5.2.2 Package „algorithm“ 8](#_Toc408413102)

[5.2.3 Package „components“ 8](#_Toc408413103)

[5.2.4 Package „service“ 9](#_Toc408413104)

[6. Arbeitsdurchführung 10](#_Toc408413105)

[7. Testbericht 11](#_Toc408413106)

[8. Lessons learned 12](#_Toc408413107)

[9. Quellenangaben 13](#_Toc408413108)

*Github-Link:* [*https://github.com/mkritzl-tgm/DezSys07-PI\_Calculator*](https://github.com/mkritzl-tgm/DezSys07-PI_Calculator)

*Github-Tag: erceg\_kritzl\_dezsys07\_v1*

# 1. Aufgabenstellung



Als Dienst soll hier die beliebig genaue Bestimmung von pi betrachtet werden. Der Dienst stellt folgendes Interface bereit:

***// Calculator.java***

**public interface Calculator {**

**public** [**BigDecimal**](http://www.google.com/search?hl=en&q=allinurl%3Abigdecimal+java.sun.com&btnI=I%27m%20Feeling%20Lucky) **pi (int anzahl\_nachkommastellen);**

**}**

Ihre Aufgabe ist es nun, zunächst mittels Java-RMI die direkte Kommunikation zwischen Klient und Dienst zu ermöglichen und in einem zweiten Schritt den Balancierer zu implementieren und zwischen Klient(en) und Dienst(e) zu schalten. Gehen Sie dazu folgendermassen vor:

1. Entwicklen Sie ein Serverprogramm, das eine CalculatorImpl-Instanz erzeugt und beim RMI-Namensdienst registriert. Entwicklen Sie ein Klientenprogramm, das eine Referenz auf das Calculator-Objekt beim Namensdienst erfragt und damit pi bestimmt. Testen Sie die neu entwickelten Komponenten.
2. Implementieren Sie nun den Balancierer, indem Sie eine Klasse CalculatorBalancer von Calculator ableiten und die Methode pi() entsprechend implementieren. Dadurch verhält sich der Balancierer aus Sicht der Klienten genauso wie der Server, d.h. das Klientenprogramm muss nicht verändert werden. Entwickeln Sie ein Balanciererprogramm, das eine CalculatorBalancer-Instanz erzeugt und unter dem vom Klienten erwarteten Namen beim Namensdienst registriert. Hier ein paar Details und Hinweise:
   * Da mehrere Serverprogramme gleichzeitig gestartet werden, sollten Sie das Serverprogramm so erweitern, dass man beim Start auf der Kommandozeile den Namen angeben ksnn, unter dem das CalculatorImpl-Objekt beim Namensdienst registriert wird. dieses nun seine exportierte Instanz an den Balancierer übergibt, ohne es in die Registry zu schreiben. Verwenden Sie dabei ein eigenes Interface des Balancers, welches in die Registry gebinded wird, um den Servern das Anmelden zu ermöglichen.
   * Das Balancierer-Programm sollte nun den Namensdienst in festgelegten Abständen abfragen um herauszufinden, ob neue Server Implementierungen zur Verfügung stehen.
   * Java-RMI verwendet intern mehrere Threads, um gleichzeitig eintreffende Methodenaufrufe parallel abarbeiten zu können. Das ist einerseits von Vorteil, da der Balancierer dadurch mehrere eintreffende Aufrufe parallel bearbeiten kann, andererseits müssen dadurch im Balancierer änderbare Objekte durch Verwendung von synchronized vor dem gleichzeitigen Zugriff in mehreren Threads geschützt werden.
   * Beachten Sie, dass nach dem Starten eines Servers eine gewisse Zeit vergeht, bis der Server das CalculatorImpl-Objekt erzeugt und beim Namensdienst registriert hat sich beim Balancer meldet. D.h. Sie müssen im Balancierer zwischen Start eines Servers und Abfragen des Namensdienstes einige Sekunden warten.

Testen Sie das entwickelte System, indem Sie den Balancierer mit verschiedenen Serverpoolgrössen starten und mehrere Klienten gleichzeitig Anfragen stellen lassen. Wählen Sie die Anzahl der Iterationen bei der Berechnung von pi entsprechend groß, sodass eine Anfrage lang genug dauert um feststellen zu können, dass der Balancierer tatsächlich mehrere Anfragen parallel bearbeitet.

# 2. Requirementanalyse

|  |  |  |  |
| --- | --- | --- | --- |
| Arbeitspaket | Zuständige Person | Geschätzte Zeit | Erledigt |
| **Client** | Erceg | 60 min |  |
| **Server** | Erceg | 60 min |  |
| **Balancer** |  | 450 min |  |
| Balancier-Algorithmus | Erceg | 210 min |  |
| Service (Registry) | Kritzl | 240 min |  |
| **CLI** | Kritzl | 120 min |  |
| **Calculator-Algorithmus** | Erceg | 30 min |  |
| **Gesamt** |  | 720 min |  |

# 3. detaillierte Arbeitsaufteilung mit Aufwandsabschätzung

## 3.1 Aufwandabschätzung

|  |  |
| --- | --- |
| Teilaufgabe | benötigte Gesamtzeit |
| UML-Diagramm erstellen | 240 Minuten (4 Stunden) |
| Implementierung des Programms inkl. JavaDoc | 720 Minuten (12 Stunden) |
| Testen des Programms | 300 Minuten (5 Stunden) |
| Protokoll schreiben | 180 Minuten (3 Stunden) |
|  |  |
| *Gesamt* | **1440 Minuten (24 Stunden)** |

## 2.2 Arbeitsaufteilung für die Implementierung des Programms

### 2.2.1 Package „control“

|  |  |  |
| --- | --- | --- |
| Klassen/Interfaces | Erceg | Kritzl |
| CLI |  | x |
| Input |  | x |
| Main |  | x |

### 2.2.2 Package „algorithm“

|  |  |  |
| --- | --- | --- |
| Klassen/Interfaces | Erceg | Kritzl |
| BalancerAlgorithm | x |  |
| CalculatorAlgorithm | x |  |
| SequenceAlgorithm | x |  |

### 2.2.3 Package „components“

|  |  |  |
| --- | --- | --- |
| Klassen/Interfaces | Erceg | Kritzl |
| Balancer |  | x |
| Calculator | x |  |
| Client | x |  |
| Server | x |  |

### 2.2.4 Package „service“

|  |  |  |
| --- | --- | --- |
| Klassen/Interfaces | Erceg | Kritzl |
| CalcService |  | x |
| Service |  | x |

# 4. anschließende Endzeitaufteilung

## 4.1 Erceg

|  |  |  |
| --- | --- | --- |
| Arbeit | Datum | Zeit in Minuten |
| UML | 12.12.2014 | 180 Minuten |
| UML fertiggestellt | 06.01.2015 | 30 Minuten |
| Protokoll | 07.01.2015 |  |
|  |  |  |
| *Gesamt* | *08.01.2015* | **Minuten ( h min)** |

## 4.2 Kritzl

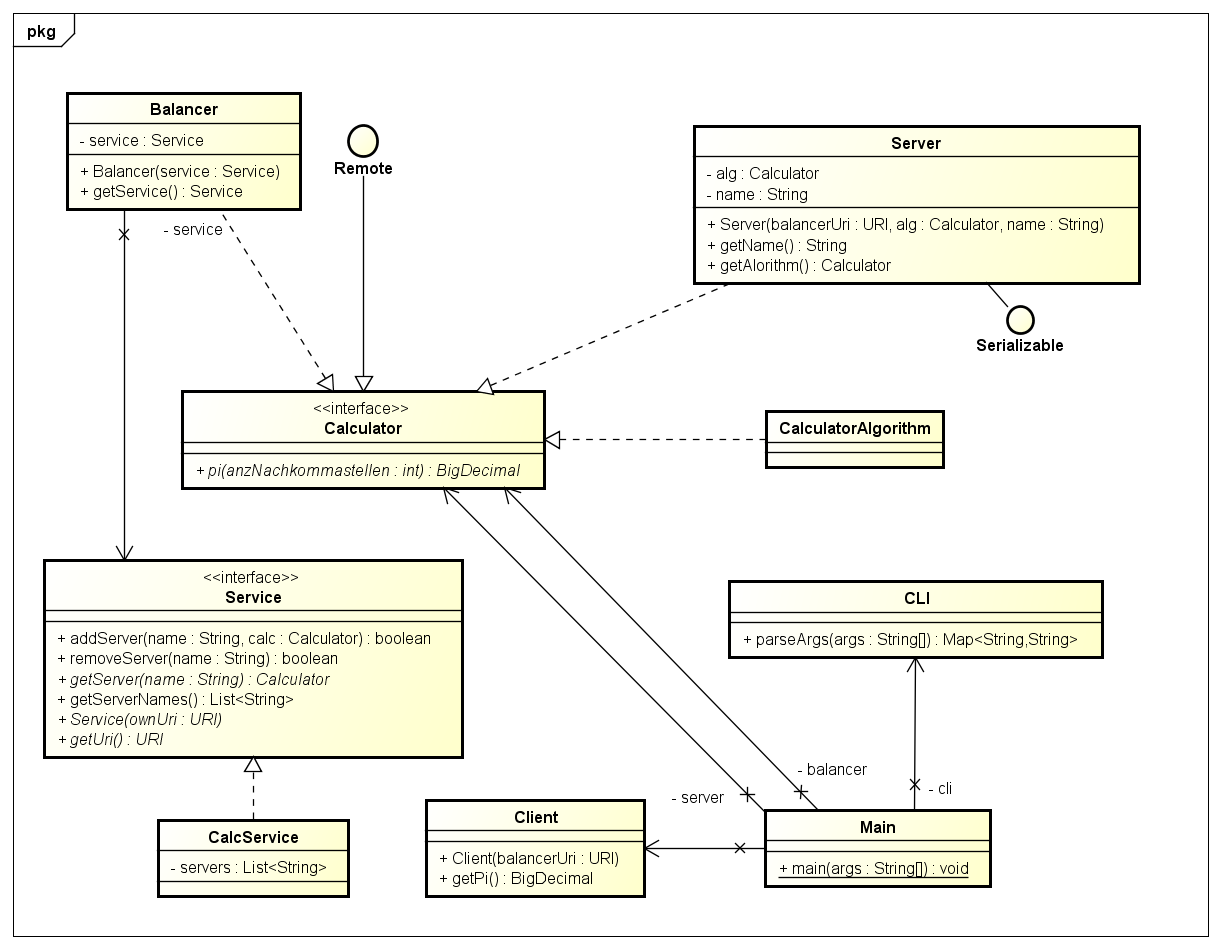
|  |  |  |
| --- | --- | --- |
| Arbeit | Datum | Zeit in Minuten |
| UML | 12.12.2014 | 180 Minuten |
| UML | 04.01.2015 | 30 Minuten |
| UML fertiggestellt | 06.01.2015 | 30 Minuten |
| Protokoll | 07.01.2015 |  |
|  |  |  |
| *Gesamt* | *08.01.2015* | **Minuten (h min)** |

## 4.3 Gesamtsumme

Insgesamt haben wir für diese Übung **Stunden** benötigt. Geschätzt wurden 24 Stunden, daher lag unsere Einschätzung ziemlich daneben.

# 5. Designüberlegung

## 5.1 Abbildung

Das UML-Diagramm wurde mit dem Programm „Astah“ erstellt.

## 5.2 Überlegungen zur Struktur

Wir haben uns überlegt, unser Programm in 4 Packages unterzuordnen:

**1.) control**

In diesem Package befinden sich die Klassen, die mit der Eingabe und der Verwaltung dieser Eingaben beschäftigt sind.

**2.) algorithm**

Dazu zählen die zwei Algorithmen zur Bestimmung des verarbeitenden Servers und der Berechnung von PI.

**3.) components**

Hier wird mit den 3 Klassen „Balancer“, „Client“ und „Server“ die Verbindung zwischen den jeweiligen Komponenten definiert.

**4.) service**

Die Klasse, die sich in diesem Package befindet, kümmert sich um die Verwaltung der verfügbaren Server. Die Server können dabei zu einer Registry hinzugefügt oder entfernt werden.

### 5.2.1 Package „control“

**Klassen, die im Package enthalten sind:**

1.) Input (Interface)

* beinhaltet die Methode „parseArgs“, mit der die jeweiligen vom Benutzer eingegebenen Argumente getrennt werden

2.) CLI

* implementiert das Interface Input

3.) Main

* initialisiert die benötigten Objekte abhängig durch die Eingabe des Benutzers
* betreibt das Exception-Handling

### 5.2.2 Package „algorithm“

**Klassen, die im Package enthalten sind:**

1.) BalancerAlgorithm (Interface)

* schreibt die Methode „getServerName“ vor, die den nächsten zur Verfügung stehenden Server ermittelt

2.) SequenceAlgorithm

* implementiert das Interface BalancerAlgorithm

3.) CalculatorAlgorithm

* implementiert das Interface Calculator (siehe 5.2.3 – Punkt 4)
* dient zur Berechnung von PI

### 5.2.3 Package „components“

**Klassen, die im Package enthalten sind:**

1.) Client

* bittet den Balancer für die Berechnung von PI mit den angegebenen Nachkommastellen

2.) Server

* berechnet für den Client PI und gibt das Ergebnis dem Balancer zurück

3.) Balancer

* stellt die Verbindung zwischen Client und Server dar
* verwendet den Balancer-Algorithmus und den Service

4.) Calculator (Interface)

* schreibt die Methode „pi“ vor, welcher als Parameter die Anzahl der Nachkommastellen übergeben wird und als Ergebnis ein BigDecimal zurückgibt

### 5.2.4 Package „service“

**Klassen, die im Package enthalten sind:**

1.) Service (Interface)

* bietet Methoden zum Hinzufügen und Entfernen von bestimmten Servern an
* das Erhalten aller Servernamen in einer Liste geschieht durch die Methode „getServerNames“
* mittels der Methode „getServer“ kann ein bestimmter Server, dessen Name im Parameter angegeben wird, zurückgegeben werden
* mittels der Methode „getURI“ kann die URI des Balancers ausgelesen werden

2.) CalcService

* implementiert das Interface Service

# 6. Arbeitsdurchführung

# 7. Testbericht

# 8. Lessons learned

# 9. Quellenangaben