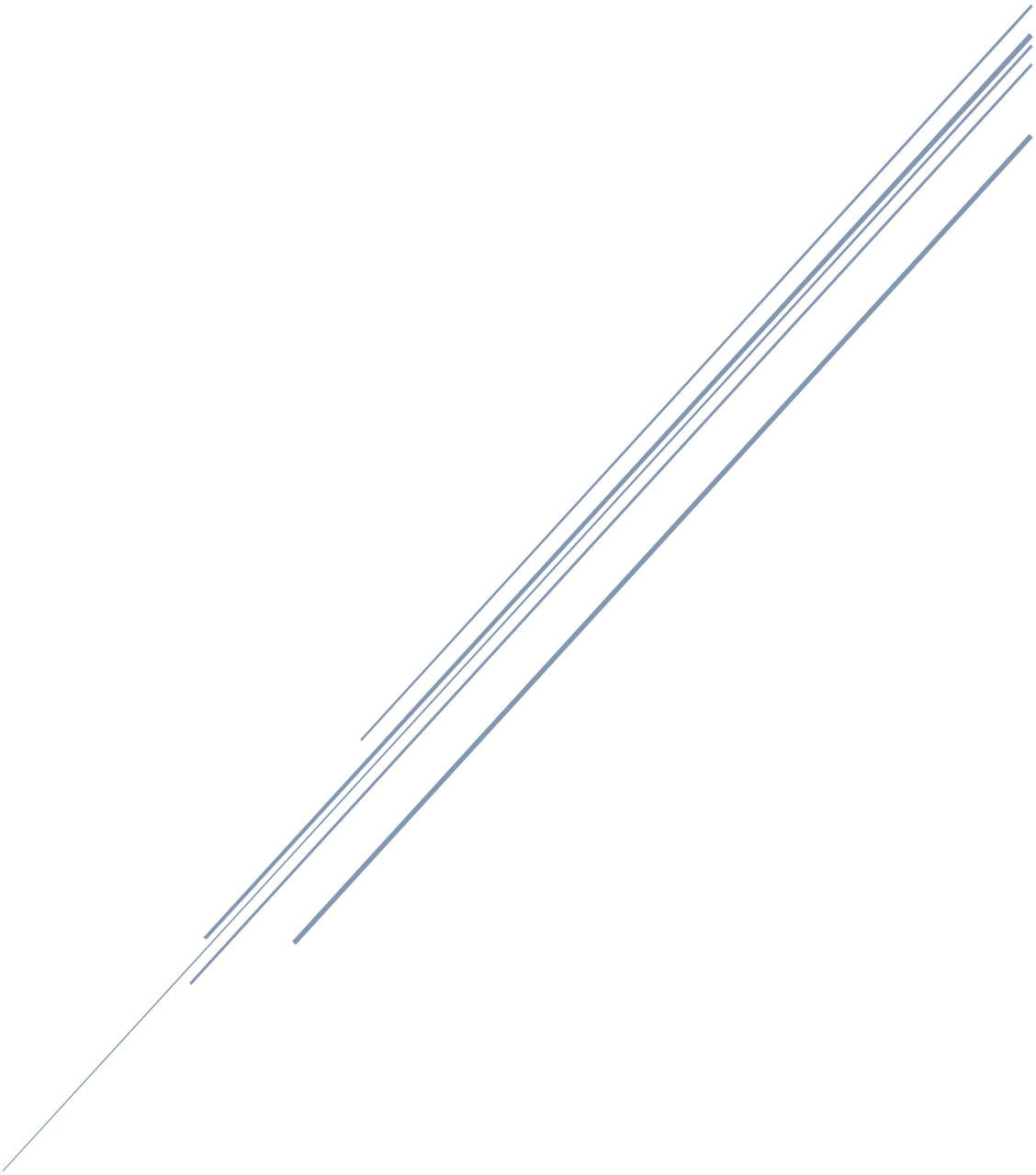
PI-CALCULATOR

DESYS-4AHIT



Patrick Malik & Thomas Taschner

08.01.2015

Contents

Aufgabenstellung 2
Distributed PI Calculator 2
Gruppenarbeit 3
Benotungskriterien 3
Quellen 3
Designüberlegung 3
Requirementsanalyse 6
Arbeitsaufteilung 6
Zeiteinteilung 7
Arbeitsdurchführung/Lessons Learned 8
Testbericht 8

Patrick Malik & Thomas Taschner
4AHIT

Aufgabenstellung

Distributed PI Calculator

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

```
1 // Calculator.java
2 public interface Calculator {
3         public BigDecimal pi (int anzahl_nachkommastellen);
4 }
```

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 kann, unter dem das CalculatorImpl-Objekt beim Load-Balancer gespeichert wird. Damit soll der Server nun seine exportierte Instanz an den Balancer übergeben, 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 Berechung von pi entsprechend gross, sodass eine Anfrage lang genug dauert um feststellen zu können, dass der Balancierer tatsächlich mehrere Anfragen parallel bearbeitet.

Gruppenarbeit

Die Arbeit ist als 2er-Gruppe zu lösen und über das Netzwerk zu testen! Nur localhost bzw. lokale Testzyklen sind unzulässig und werden mit 6 Minuspunkten benotet!

Benotungskriterien

o 12 Punkte: Java RMI Implementierung (siehe Punkt 1)
o 12 Punkte: Implementierung des Balancers (siehe Punkt 2)

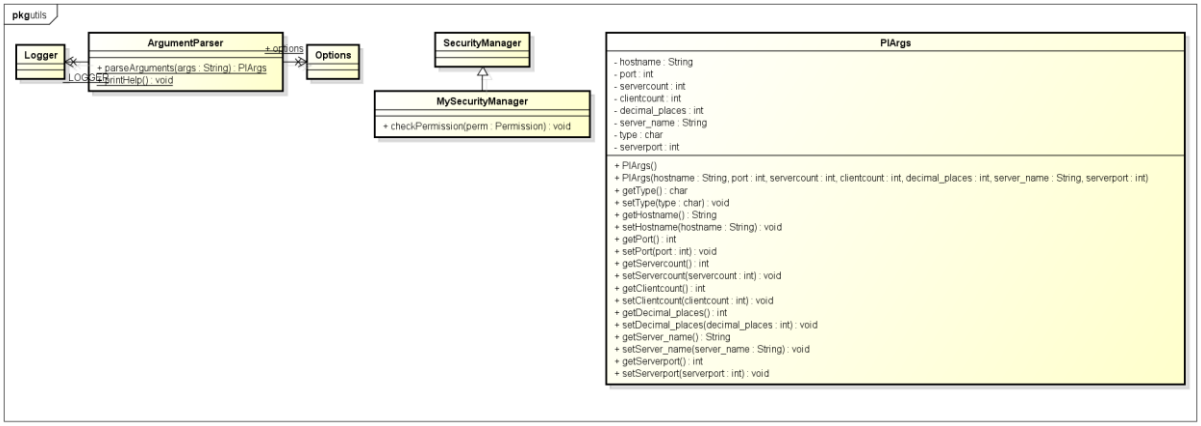
o davon 6 Punkte: Balancer

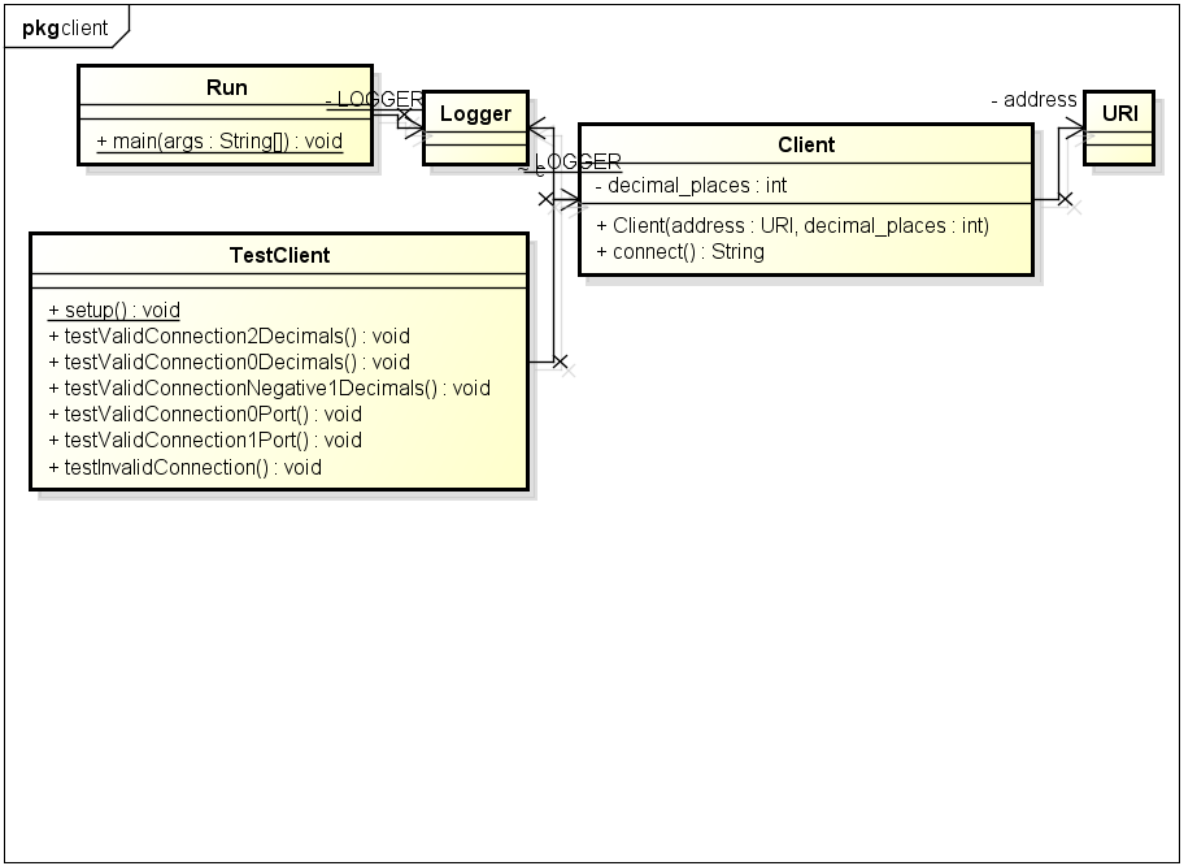
o davon 2 Punkte: Parameter - Name des Objekts
o davon 2 Punkte: Listing der Server (dyn. Hinzufügen und Entfernen)
o davon 2 Punkte: Testprotokoll mit sinnvollen Werten für Serverpoolgröße und Iterationen

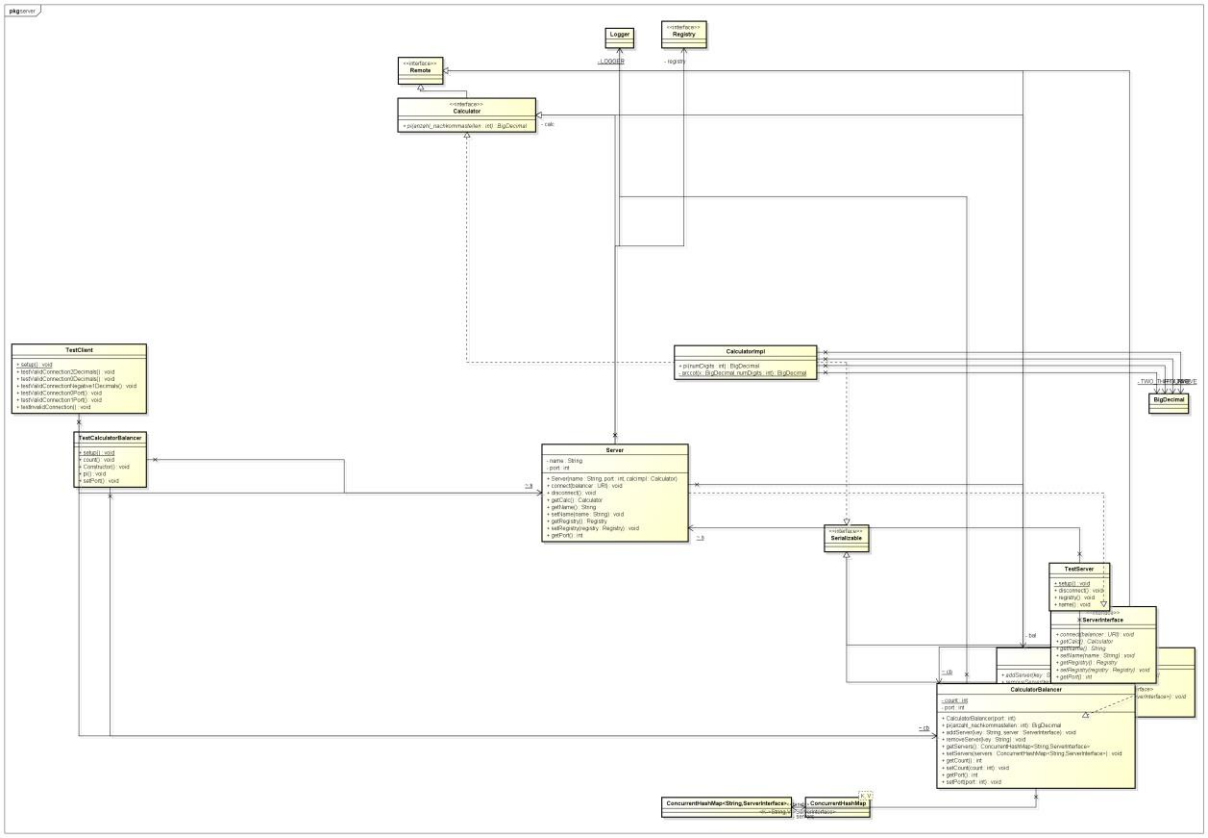
Quellen

An Overview of RMI Applications, Oracle Online Resource, http://docs.oracle.com/javase/tutorial/rmi/overview.html (last viewed 28.11.2014)

Designüberlegung







Client

Der Client bekommt die URI vom Balancer und die Dezimalstellen für PI, überprüft die Richtigkeit der Parameter und setzt die nötigen Attribute.

Mit connect() wird sich mithilfe der URI auf den Balancer verbunden, der Client holt sich den Calculator und führt pi aus.

Run

Run setzt zuallererst den SecurityManager auf unseren eigenen, verarbeitet dann die Argumente, erstellt mithilfe derer einen Client, Server oder Balancer, somit sparen wir uns das Erstellen der 3 jars.

BalancerInterface

Dieses Interface ist dazu da, den Balancer exportieren zu können, es schreibt dem Balancer die notwendigen Methoden vor.

Calculator

Dieses Interface wurde uns bei der Aufgabenstellung geliefert.

CalculatorBalancer

Hierbei handelt es sich um das Herzstück des ganzen Programms, hier wird eine ConcurrentHashMap initialisiert um die anmeldenden Server zu registrieren. Er implementiert zudem das Calculator Interface um für den Client wie ein Rechner auszusehen, er binded sich selbst in seine registry und vergibt in pi via round robin die server die sich um den client kümmern.

CalculatorImpl

Diese Implementierung wurde auf

http://stackoverflow.com/questions/8370290/generating-pi-to-nth-digit-java

gefunden, leicht verändert und sonst so verwendet.

Server

Wird auf einem bestimmten Host und Port mit einem bestimmten Namen gestartet. Kann PI mit beliebiger Genauigkeit berechnen. Es wird eine lokale Registry erstellt, exportiert sich dann auf seine eigene Registry.

Beim Verbinden verbindet sich dieser mit einem Balancer, lädt die Reigstry, sucht nach einem Rechendienst und wird der Balancer Serverliste hinzugefügt.

ServerInterface:

Besitzt Methoden zum Herstellen einer Verbindung mit dem Balancer und diverse Getter- und Setter Methoden für die Attribute der Klasse.

ArgumentParser:

Klasse, die die beim Ausführen mitgegebenen Parameter genau analysiert, erkennt, welche Connection gewünscht ist und den User auf Fehler hinweist.

Bei Bedarf kann auch eine Hilfe angezeigt werden.

MySecurityManager

Da uns der Logger zu Beginn mit dem SecurityManager gemeinsam Probleme gemachthat, haben wir einen eigenen implementiert um den Logger seine Klassen laden zu lassen.

PIArgs

Eine Klasse die Alle Argumente sammelt um sie für den parser als Speicher verwenden zu können. Der Parser speichert alle seine Daten in das Objekt und des weiteren kann dann eines der 3 Objekte erstellt werden.

Requirementsanalyse

- Aufgabe
  - Interpretation
- Aufgabe 1: Serverprogramm & CalculatorImpl
  - Implementieren des Calculator Interfaces, erzeugen eines CalculatorImpl-Objekts und an die registry des Servers zu binden
- Aufgabe 1: Client
  - Verbinden zur registry des Servers, Aufrufen der Methode pi() via RMI auf dem Calculator Objekt.
  - So gestalten, dass für Aufgabe 2 nichts mehr geändert werden muss.

Aufgabe 2

- Implementierung des Balancers ohne den Client ändern zu müssen
  - Alles was in der registry des Balancers liegt, sollte ein Objekt mit Namen „Calculator“ sein, wie vom Client erwartet, dieses sollte auch Calculator implementieren.
- Name des Servers sollte anzugeben sein
  - Wird mit den anderen Argumenten geparsed und in den Server gespeichert.
- Abfragen welche Server derzeit aktuell sind
  - Wir verwenden eine concurrentCollection um alle Server einzutragen, die Server tragen sich ein, wenn sie connecten, tragen sich bei disconnect aus und bei abstürzen eines Servers wird die Anfrage weitergeleitet und der Server aus der Collection gelöscht
- Threadsichere Anwendung
  - Der einzige Punkt in unserem Code der gesichert werden muss, ist die Collection auf die die Server beim Registrieren zugreifen und diese ist concurrent.
- Warten um sicherzugehen, dass Server gestartet sind
  - Unsere Server fahren bei der Initialisierung alles hoch, erzeugen ihre registry und erledigen alles was nötig ist, erst danach connecten sie sich mit dem Balancer und sind somit bei Registrierung arbeitsfähig.
- Listing der Server mit dynamischen hinzufügen und löschen
  - Mithilfe einer Collection wird das hinzufügen und Löschen der Server vom Balancer gehandhabt.

Arbeitsaufteilung

Aktion	**Person**
Designen der Klassen	Taschner,Malik
Schreiben der Dokumentation	Taschner, Malik
Schreiben der Kommentare	Taschner
Testen (JUnit)	Taschner, Malik
Testen (im Netzwerk, nicht automatisiert)	Taschner, Malik

Implementierung Logger	Malik
Implementierung CalculatorImpl	Taschner
Implementierung Client	Taschner, Malik
Implementierung Run	Taschner, Malik
Implementierung BalancerInterface	Taschner, Malik
Implementierung Server	Taschner, Malik
Implementierung ServerInterface	Taschner, Malik
Implementierung ArgumentParser	Malik
Implementierung MySecurityManager	Malik
Implementierung PIArgs	Malik
ExceptionHandling	Taschner

Zeiteinteilung

Patrick Malik

Aktion		**Geschätzt**	**Tatsächlich**
Designen		30min	45min
Dokumentieren		30min	30min
Testen (JUnit)		60min	60min
Testen (im Netzwerk, nicht automatisiert)		10min	30min
Implementierung Logger		15min	20min
Implementierung Client		30min	15min
Implementierung Run		20min	60min
Implementierung BalancerInterface		2min	2min
Implementierung Server		60min	60min
Implementierung ServerInterface		2min	2min
Implementierung ArgumentParser		30min	30min
Implementierung MySecurityManager		2min	2min
Implementierung PIArgs		5min	5min
	Gesamt	296min	361min

Thomas Taschner

Aktion		**Geschätzt**	**Tatsächlich**
Designen		30min	45min
Dokumentieren		30min	30min
Schreiben der Kommentare		60min	120min
Testen (JUnit)		120min	120min
Testen (im Netzwerk, nicht automatisiert)		10min	30min
Implementierung CalculatorImpl		30min	10min
Implementierung Client		30min	15min
Implementierung Run		20min	60min
Implementierung BalancerInterface		2min	2min
Implementierung Server		60min	60min
Implementierung ServerInterface		2min	2min
ExceptionHandling		20min	45min
	Gesamt	414min	539min

Arbeitsdurchführung/Lessons Learned

- Umgang mit Java RMI

- Werfen von Exceptions ist besser, als sie alle aufzufangen

- Erste Praxisschritte bezüglich Pair Programming

- Erneuter Umgang mit der Apache CLI Commons

- Erneuter Umgang mit Log4J

- Kennenlernen von @BeforeClass in JUnit

- Kennenlernen von SecurityManagern (allerdings nicht sonderlich viel)

Wie den Lessons Learned zu entnehmen ist, haben wir die meisten Klassen gemeinsam geschrieben (über Teamviewer, Bildschirmübertragung,…). An manchen Klassen hat einer mehr als der Andere gearbeitet, jedoch auch umgekehrt. Auffallend ist vielleicht das besonder Thomas Taschner viel kommentiert hat, das liegt allerdings einer 8stündigen Zugfahrt ohne Internet zugrunde.

Testbericht

Beim Testen sind wir unter anderem darauf gestoßen, da das handlen von Exceptions in den eigenen Klassen nicht sonderlich von Vorteil ist, das Testen solcher Bereiche stellt sich oft als Herausforderung dar, zudem schlägt sich das auch auf die Code-Coverage nieder.

Client: 67.5%

Run: 15.6%

CalculatorBalancer: 78.9%

Server: 57.9%

ArgumentParser: 94.7%

MySecurityManager: 100%

PIArgs: 100%

Besonders beim Client und der Run wird verhältnismäßig viel ExceptionHandling betrieben.