Distributed JUnit

Bassim Aziz Safi

13. April 2005

Kurze Laufzeiten beim Testen fördern Akzeptanz und damit den Nutzen von Tests. In dieser Arbeit wird daher untersucht, in wie weit die Laufzeiten durch eine Verteilung der Testausführung auf mehrere Rechner verkürzt werden kann. Dazu wird ein Test-Framework namens Distributed JUnit erstellt und einer Performance-Evaluation unterzogen. Distributed JUnit, das ein Drop-In-Replacement des Java-Regressionstest-Frameworks JUnit darstellt, automatisiert das parallele Ausführen von Tests in einem Jini-Netzwerk.

Inhaltsverzeichnis

1. Einführung

1.1. Motivation

Das regelmäßige Durchführen von Regressionstests steigert nachweislich die Qualität von Software [17], bedeutet aber auch zusätzlichen Zeitaufwand. Bei umfangreichen Projekten kann dieser Aufwand so groß werden, dass die Ansicht entsteht, dass sich der Zeitaufwand des Testens nur dann lohnt, wenn genügend große Schritte in der Entwicklung zwischen den Testläufen getätigt werden. Das führt dann dazu, dass Tests sehr viel seltener als eigentlich empfehlenswert ausgeführt werden. Dadurch, dass große und damit komplexere Schritte bei der Entwicklung und der Wartung von Software unternommen werden, steigt das Risiko, dass Fehler erst später gefunden werden. Je später ein Fehler aber gefunden wird, desto größer ist üblicherweise der Aufwand, den Fehler zu beheben.

Letztendlich führen lange Testlaufzeiten und die daraus resultierenden Reaktionen zu langsameren Entwicklungszyklen, weniger Flexibilität bei der Entwicklung und einem Produkt mit mehr Fehlern als nötig.

Bedenkt man aktuellere Entwicklungen in der Softwaretechnik, wie zum Beispiel Paradigmen wie Extreme Programming, so erscheint die Relevanz kurzer Laufzeiten sogar noch viel höher. In Projekten, die das Extreme-Programming-Paradigma verfolgen, ist das regelmäßige Testen der zu entwickelnden Software einer der Grundpfeiler, auf die sich nahezu alle übergeordneten Entscheidungsprozesse stützen. Hier ist es ganz besonders wichtig, kurze Laufzeiten zu erreichen.

Man kann allgemein zusammenfassen, dass kurze Laufzeiten beim Testen Akzeptanz und damit Nutzen von Tests fördern, da diese dann öfter ausgeführt werden. Damit wird eine höhere Effizienz in der Entwicklung und bessere Software-Qualität erreicht.

Es existieren drei grundlegende Ansätze zur Laufzeitverkürzung: der Selektionsansatz, der Priorisierungsansatz und der Distributionsansatz. Bei dem Selektionsansatz und dem Priorisierungsansatz wird die Ausprägung ausgeführten Tests dahingehend verändert, dass entweder einzelne Tests bei der Ausführung übergangen, oder deren Reihenfolge verändert wird. Die Änderung der Reihenfolge stellt theoretisch keine Gefahr da, welche bei einer Auslassung eines Tests durchaus besteht (vgl. Abschnitt 2.2.1), die für eine sinnvolle Umstellung der Reihenfolge notwendigen Berechnungen sind aber sehr aufwendig (vgl. Abschnitt 2.2.2).

Da beim Distributionsansatz aber im Gegensatz zu den anderen Ansätzen die Testausprägung unangetastet bleibt, stellt der Distributionsansatz, sollte er eine genügende Effizienz aufweisen, ein vielversprechender Ansatz zur Laufzeitverkürzung von Tests dar.

1.2. Problemdefinition und Zielsetzung

In dieser Arbeit soll der Distributionsansatz untersucht, eine Implementation von Distributed JUnit vorgestellt und einer Performance-Evaluation unterzogen werden. Die Zielsetzung dabei ist zum einen, herauszufinden, welches Potential eine Verteilung von Tests im Hinblick auf

die Verkürzung der Laufzeit besitzt. Die Performance-Evaluation der Implementation soll darüber Aufschluss geben. Zum anderen soll ein Eindruck davon gewonnen werden, welche Anforderungen und Einschränkungen aus einer Verteilung der Testausführung resultieren. Aufgrund der Natur eines Netzwerks und den Möglichkeiten des Zusammenspiels der im Netzwerk verteilten Komponenten ergeben sich allgemeine Einschränkungen, die sich je nach gewählter Umsetzung der Verteilung mehr oder minder relativieren oder gar ganz umgehen lassen. Die überwiegende Menge der angesprochenen Problematiken sind jedoch eng mit den Eigenheiten der verwendeten Infrastruktur verknüpft.

Anhand der Implementation von Distributed JUnit kann konkret dargestellt werden, welche Vorzüge und Nachteile diesbezüglich die Verteilung der Testausführung mit sich bringt. Java-spezifische Einschränkungen, die sich aus der Java-Sprachspezifikation ergeben, insbesondere die zu beachtenden Details in Verbindung mit dem Serialisierungsmechanismus (vgl. 5.1.1), sind in diesem Fall eine Besonderheit dieser Implementation und gelten nicht unbedingt allgemein für den Distributionsansatz.

Die Implementation von Distributed JUnit soll dabei so gestaltet werden, dass sie anstelle des JUnit-Framework benutzt werden und dieses möglichst uneingeschränkt ersetzen kann, ein sogenanntes Drop-In-Replacement.

1.3. Aufbau der Arbeit

Zunächst wird ein Einblick in die Relevanz der Testlaufzeit und bereits existierende Lösungsansätze gegeben. Der Ansatz der Distribution wird dabei allgemein vorgestellt. Im Hauptteil der Arbeit wird das Konzept von Distributed JUnit, die Funktionsweise der einzelnen Komponenten und die Implementation behandelt. Dazu gehört auch die Performance-Evaluation. Schließlich werden die Ergebnisse kritisch betrachtet, zusammengefasst und durch einen Ausblick auf mögliche zukünftige Ergänzungen abgeschlossen.

2. Theoretische Grundlagen

2.1. Relevanz der Testlaufzeit

Gründe für regelmäßiges und ausgiebiges Testen

Software-Entwickler, die schon einmal mit JUnit als Regressionstest-Framework gearbeitet haben, wissen um die Vorteile des regelmäßigen Testen ihrer Anwendung. Das anfängliche Zweifeln, ob sich die zunächst als zusätzlicher Aufwand empfundene Arbeit des Erstellens und Durchführens von Tests lohnt, schlägt meist schnell in große Wertschätzung der nachweislichen Vorzüge dieser Herangehensweise um [16, 17]:

Nicht nur die Tatsache, dass durch das Testen viele Fehler schon in einem frühen Stadium der Entwicklung einer Anwendung gefunden und beseitigt werden können, vorallem auch das daraus entstehende Gefühl der Sicherheit führt zu einer konsequenteren Anwendung von modernen Ansätzen wie Refactoring [8] und anderen Techniken des Extreme Programming [2] und damit zu Quellcode, der weniger Fehler aufweist, besser erweiterbar und einfacher zu warten ist [20].

Stichwort Akzeptanz

Bei der Entwicklung von JUnit wurde als das Hauptziel festgelegt, das Framework so zu gestalten, dass Programmierer tatsächlich Tests schreiben. Es wurde also von Anfang an das Hauptaugenmerk auf eine gute Akzeptanz des Frameworks gelegt. Durch Verwendung bekannter Konzepte und größtmögliche Reduzierung des Aufwands beim Erlernen und Anwenden des Frameworks darf man dies auch als gelungen betrachten: Laut SourceForge wurde die momentan aktuelle Version 3.8.1 von JUnit bereits über eine halbe Millionen Mal heruntergeladen - mehr als alle anderen xUnit-Frameworks zusammen[1].

Inzwischen aber tritt durch die nun intensive Nutzung des Frameworks ein Umstand hervor, der sich äußerst nachteilig auf die Akzeptanz auswirken kann: der Faktor der Laufzeit.

Was die Laufzeit für die Akzeptanz bedeutet

Sobald die Laufzeit bei der Durchführung von Tests spürbar wird, ist ein schnelles Hin- und Herwechseln zwischen Programmieren und Testen nicht mehr ohne Bruch machbar.

> „This quick cycling between coding and testing is essential. With it, you find bugs immediately. Fixing them is usually simple and doesn't disrupt the flow of what you're doing. Quick cycling is what make programmers true believers in testing“ [18]

Bei umfangreicheren Tests kann deren Durchführung zum Teil weit über das hinausgehen, was subjektiv oder auch objektiv als erträglich empfunden wird:

[1] `http://sourceforge.net/project/showfiles.php?group_id=15278&package_id=12472`

> „Running all test cases in an existing test suite, however, can consume an inordinate amount of time. For example, one of our industrial collaborators reports that for one of its products that contains approximately 20,000 lines of code, running the entire test suite requires seven weeks.“ [10]

Es ist also von nicht zu unterschätzender Bedeutsamkeit, die Laufzeiten von Tests so gering wie möglich zu halten, da sonst die Akzeptanz leidet, damit die Verwendung nachlässt und hierdurch der Nutzen von Regressionstests erheblich veringert wird. Welche Möglichkeiten existieren nun aber und wie sind sie beschaffen?

2.2. Ansätze zur Laufzeitverkürzung

Unter den Ansätzen zur Beschleunigung der Testlaufzeiten sind drei grundsätzliche Herangehensweisen hervorzuheben: Selektion, Priorisierung und Distribution. In diesem Abschnitt werden diese Ansätze vorgestellt und ihre Eigenschaften beurteilt.

Notation

Es wird die von Rothermel und Harrold vorgestellte und von Kapfhammer erweiterte Notation zur Beschreibung von Regressionstestarten benutzt. Sie soll hier kurz zusammengefasst werden [12, 9].

Der Ausdruck $T = \{t_1, \ldots, t_n\}$ stellt eine Testsuite dar. P sei ein Programm, dass im Hinblick auf T korrekt ist. P' stellt eine modifizierte Version dieses Programms dar, beispielsweise das Ergebnis einer Refaktorisierung. Mit diesen Vereinbarungen lässt sich das Regressionstestproblem folgendermaßen notieren [11]:

Problem 1 (Regressionstest) Sei ein Programm P, dessen modifizierte Version P' und eine Testsuite T gegeben. Benutze T, um einen ausreichenden Beweis für die Korrektheit von P' zu erhalten.

Der folgende Algorithmus ist die Basis aller Regressionstest-Algorithmen und beschreibt die einzelnen Schritte, die für eine vollständigen Beweis der Korrektheit des getesteten Programms nötig sind.

Algorithmus 0 (Regressionstest)

1. Teste P' mit T um die Korrektheit von P' im Hinblick auf T zu ermitteln.
2. Falls nötig, erstelle T', einen Satz von neuen Tests für P'.
3. Teste P' mit T' um die Korrektheit von P' im Hinblick auf T' zu ermitteln.
4. Erstelle T'', eine neue Testsuite für P', aus T und T'.

In Schritt 1 wird die Modifikation von P mit dem Test getestet, der einen Beweis für die Korrektheit der ursprünglichen Version von P liefern kann. In Schritt 2 wird versucht, diejenigen Teile des Programms P' zu identifizieren, die noch nicht ausreichend getestet wurden und es wird versucht, die noch benötigten Tests zu erstellen. Stellt sich heraus, dass T

nicht ausreichend ist, um den Beweis der Korrektheit von P' zu liefern, weil beispielsweise neue Funktionalität in P' implementiert wurde, für die noch kein Test existiert, werden entsprechende Tests erstellt und Schritt 3 durchgeführt. In Schritt 4 schließlich werden alle für P' relevanten Tests in T'' vereint. Dabei kann es durchaus passieren, dass nicht alle Tests übernommen werden, beispielsweise wenn ein neu hinzugekommener Test aus T' einen bereits existierenden Test aus T ersetzt oder die Modifikation P' einen Test aus T überflüssig hat werden lassen. T'' muss allerdings in jedem Fall fähig sein, den vollständigen Beweis für die Korrektheit von P' zu liefern.

2.2.1. Selektion

Die Selektion versucht, den Aufwand des Testens dadurch zu vermindern, dass eine Untermenge T' von T gebildet und nur diese Untermenge beim Testen von P' ausgeführt wird. Dies unterscheidet die Selektion grundlegend vom Ansatz des herkömlichen Regressionstestens, der besagt, dass in jedem Testablauf sämtliche Tests durchgeführt werden. Der folgende Algorithmus von Rothermel und Harrold beschreibt den Ablauf eines Regressionstests mit Selektion:

Algorithmus 1 (Regressionstest mit Selektion)

1. Wähle $T' \subseteq T$, eine Untermenge von Tests, mit denen P' getestet wird.
2. Teste P' mit T' um die Korrektheit von P' im Hinblick auf T' zu ermitteln.
3. Falls nötig, erstelle T'', einen Satz von neuen Tests für P'.
4. Teste P' mit T'' um die Korrektheit von P' im Hinblick auf T'' zu ermitteln.
5. Erstelle T''', eine neue Testsuite für P', aus T, T' und T''.

Dadurch, dass nur diejenigen Tests überhaupt ausgeführt werden, die für die modifizierte Version P' relevant sind, kann die Gesamtlaufzeit eines Testdurchlaufs veringert werden.

Eine Schwierigkeit liegt in der hier in Schritt 1 des Algorithmus 1 erfolgenden Bestimmung der Untermenge der Tests, die ausgeführt werden sollen. Dies geschieht, wie bereits angesprochen, abhängig nach Relevanz der einzelnen Tests für P'.

Die Relevanz eines Tests ist abhängig von seiner eigenen Ausprägung und auch von der Ausprägung der Modifikation P' des Programms P. Für ihre Bestimmung wird der normalerweise der Quelltext des zu testenden Programms analysiert. [9]

Eine andere Schwierigkeit liegt darin, dass in manchen Fällen kein Test als irrelevant bezeichnet werden kann und somit eine Selektion keine Verminderung des Umfangs der Tests erreicht. Dies ist z.B. dann der Fall, wenn P' eine Portierung in eine andere Programmiersprache darstellt, prinzipiell also keine Änderung der Semantik des Programms erfolgt ist und daher alle Tests durchgeführt werden sollten, die vor der Portierung nötig waren, um die Korrektheit zu beweisen [9]. Eine Selektion, bei der Tests ausgelassen würden, könnte in diesem Fall also dazu führen, dass der Test nicht ausreichend für den Beweis der Korrektheit ist. Der ausschlaggebende Faktor für die Relevanz der selektierten Tests ist hierbei die Ausprägung von P'.

Sehr häufig bildet der Test selbst den ausschlaggebenden Faktor, so kommen bestimmte Tests wie z.B. User-Acceptance-Tests normalerweise für eine Auslassung nicht in Frage [12].

Zusammenfassend kann man behaupten, dass der Selektionsansatz durch die dargestellten Eigenschaften nicht immer effektiv zu einer Laufzeitverkürzung beitragen kann [12].

2.2.2. Priorisierung

Der von Elbaum et al. entwickelte Ansatz der Priorisierung verfolgt das Ziel, durch eine Permutation der einzelnen Tests in einer Testsuite die Wahrscheinlichkeit dafür zu erhöhen, dass gewisse Fehler im getesteten Programm schon früher während des Ablaufs des Tests gefunden werden. Wenn kritische Fehler dann schon früh während des Testens gefunden werden, kann der Test abgebrochen und somit die Laufzeit verkürzt werden.

Der folgende Algorithmus beschreibt den Ablauf eines Regressionstests mit Priorisierung [19]:

Algorithmus 2 (Regressionstest mir Priorisierung)

1. Erzeuge T_p, eine Permutation von T, so dass T_p eine bessere Fehlererkennungsrate als T hat.
2. Teste P' mit T_p um die Korrektheit von P' im Hinblick auf T_p zu ermitteln.
3. Falls nötig, erstelle T', einen Satz von neuen Tests für P'.
4. Teste P' mit T' um die Korrektheit von P' im Hinblick auf T' zu ermitteln.
5. Erstelle T'', eine neue Testsuite für P', aus T, T_p und T'.

In Schritt 1 des Algorithmus 2 erfolgt die Permutation der Tests innerhalb der Testsuite. Die Permutation erfolgt dabei nach verschiedenen Gesichtspunkten. Zum Beispiel kann versucht werden, schon früh im Testablauf einen möglichst großen Teil des Programmcodes abzudecken. Oder es kann versucht werden, vorallem solche Fehler, die kompetente Programmierer hauptsächlich machen, möglichst früh im Testablauf zu erkennen.

Diese Ziele können erreicht werden, indem man die Tests in der Testsuite dabei nach ihrem Fehleraufdeckungspotential für die jeweilig gesuchte Klasse von Fehlern anordnet, so dass diejenigen Tests, die für die gesuchte Klasse von Fehlern ein hohes Fehleraufdeckungspotential besitzen, zuerst ausgeführt werden.

Das Fehleraufdeckungspotential eines Tests lässt sich bestimmen mit Hilfe der Technik der Mutationsanalyse [19]. Dabei wird der Test, dessen Fehleraufdeckungspotential ermittelt werden soll, mit einer Anzahl speziell erstellter Mutationen eines Programms konfrontiert und es wird ermittelt, ob der Test die verschiedenen Fehler in den einzelnen Mutationen des Programms findet.

Diese Technik hat sich als effektiv erwiesen, ist aber sehr rechenintensiv. Daher kann es mitunter vorkommen, dass, wenn man die durch die Mutationsanalyse benötigte Rechenzeit miteinbezieht, die Anwendung des Priorisierungsansatzes sogar einen negativen Effekt auf die Laufzeitverkürzung hat [19].

Da beim Erstellen der Tests im Allgemeinen versucht wird, Abhängigkeiten von anderen Tests zu vermeiden, z.B. in Bezug auf deren Ausführungsreihenfolge, kann man behaupten, dass die Umordnung selbst üblicherweise keine negativen Nebeneffekte hat. Auch werden keine Tests ausgelassen. Daher stellt sich bei diesem Ansatz im Gegensatz zur Selektion nicht die Frage nach der bedenkenlosen Anwendbarkeit, sondern schon eher die Frage der Effizienz.

2.3. Der Distributionsansatz

Beim Ansatz der Distribution wird versucht, die Laufzeit einer Testdurchführung durch das parallelisierte Ausführen der einzelnen Tests auf mehreren Rechnern zu verkürzen.

Der folgende Algorithmus beschreibt den Ablauf eines Regressionstests mit Distribution:

Algorithmus 3 (Regressionstest mit Distribution)

1. Teste P' mit T um die Korrektheit von P' im Hinblick auf T zu ermitteln, unter Benutzung von m Rechnern.
2. Falls nötig, erstelle T', einen Satz von neuen Tests für P'.
3. Teste P' mit T' um die Korrektheit von P' im Hinblick auf T' zu ermitteln, unter Benutzung von m Rechnern.
4. Erstelle T'', eine neue Testsuite für P', aus T und T'.

Die angesprochenen Schwierigkeiten der zuvor behandelten Ansätze kommen hier nicht zu Tragen, da, adäquate Transparenz des Distributionsmechanismus vorausgesetzt, die Tests und deren Ausführung nicht verändert werden. Die Nutzung dieses Ansatzes bedeutet natürlich einen höheren Einsatz an Ressourcen wie z.B. Rechner und eine Netzwerk-Infrastruktur. Diese Ressourcen sind allerdings oftmals sowieso vorhanden und noch nicht ausgelastet. Die Möglichkeit zur Ausnutzung zuvor mehr oder weniger brachliegender Ressourcen verbunden mit dem Wegfall der Notwendigkeit potentiell nachteiliger Modifikationen des Tests lässt diesen Ansatz zunächst recht attraktiv erscheinen.

Viele Eigenschaften des Ansatzes der Distribution aber liegen in den Details der Umsetzung des Distributionsmechanismus verborgen und je nach Art und Ausprägung der Umsetzung sind diese Eigenschaften mehr oder weniger wünschenswert. Im folgenden Abschnitt soll dies näher untersucht werden.

2.3.1. Erwägungen zum Distributionsansatz

Um herauszufinden, wie die Umsetzung des Distributionsmechanismus am geeignetsten erfolgen sollte, hat Kapfhammer eine Anzahl Erwägungen aufgezählt, die einen geeigneten Distributionsmechanismus charakterisieren [12]. Diese Erwägungen und eigene Ergänzungen sollen nun vorgestellt und untersucht werden:

Erwägung 1 (Transparente und Automatisierte Distribution) Eine Distribution von n Tests auf m Rechner sollte so transparent und automatisiert wie möglich erfolgen und darf die ordentliche Ausführung der Tests nicht verhindern.

Erwägung 2 (Vermeidung von Testkontamination) Wenn n Tests auf m Rechner verteilt werden, darf dabei keine durch die gleichzeitige Ausführung zweier Tests hervorgerufene Testkontamination aufreten.

Erwägung 3 (Lastverteilung) Wenn n Tests auf m Rechnern verteilt werden, sollte dies auf eine Weise geschehen, die die Rechenlast angemessen auf die Rechner verteilt.

Erwägung 4 (Testintegrität) Eine Distribution von n Tests auf m Rechner darf die Korrektheit der Testergebnisse nicht beeinflussen.

Erwägung 5 (Ausführungskontrolle) Eine Distribution von n Tests auf m Rechner sollte von einem zentralen Ort initiiert, überwacht und ausgewertet werden können.

Erwägung 6 (Netzwerksicherheit) Die Sicherheit eines Rechners und der Netzwerk-Infrastruktur darf durch die Beteiligung an der Distribution nicht beeinträchtigt werden.

Erwägung 1 beschreibt, dass der Tester so wenig wie möglich mit Fragen der Verteilung bedacht werden sollte. Die Funktionsweise und Konfiguration sollte soweit wie möglich unabhängig und automatisiert erfolgen. Eine verteilte Ausführung stellt eine grundlegend andere Umgebung für ein Programm dar, in der keine gemeinsamen Adressräume existieren und der Speicherort eines Tests nichts mit dem Ausführungsort eines Tests zu tun hat. Diese Eigenschaften müssen durch entsprechende Maßnahmen behandelt werden, so dass Tests trotz völlig veränderter Ausführungsumgebung ordentlich durchgeführt werden können. Die Tatsache, dass ein Test verteilt ausgeführt werden könnte, sollte dabei einen möglichst geringen Einfluß auf dessen Gestaltung oder die dabei zur Verfügung stehenden Möglichkeiten haben.

Erwägung 2 behandelt das Problem der Testkontamination. Ein Test t_i kontaminiert einen anderen Test t_j, wenn t_i das Ergebnis von t_j bei gleichzeitige Ausführung der beiden Tests auf verschiedenen Rechnern beeinflussen kann. Beispiele dafür sind Tests, die gemeinsame Ressourcen oder Services benutzen und dabei deren Zustand verändern. Es spielt dabei keine Rolle, ob diese Veränderungen nach Abschluss des Tests wieder rückgängig gemacht werden oder gar als Vorbereitung für einen folgenden Test belassen werden, in beiden Fällen sind diese Tests voneinander abhängig, da sich ihre Testumgebungen zeitlich gesehen überschneiden und sie sich gegenseitig kontaminieren könnten, wenn die ursprüngliche Reihenfolge bzw. die Serialität nicht mehr gegeben ist.
Es gibt keine zufriedenstellenden Möglichkeiten, um allein anhand eines Tests, z.B. seines Quellcodes, festzustellen, ob dieser einen anderen Test kontaminieren könnte. Hierfür würden Meta-Daten benötigt, die Aufschluss über Abhängigkeiten mit anderen Tests geben und bei der Distribution berücksichtigt werden können.

Die sich aus der potentiellen Heterogenität eines Netzwerkes ergebende Frage der Lastverteilung wird in Erwägung 3 angesprochen. Eine angemessene Lastverteilung hängt von verschiedenen Parametern ab. Denkbar ist beispielsweise ein Lastverteilungsmechanismus, der die erlaubte Auslastung einzelner Rechner berücksichtigt, falls auf diesen parallel andere Prozesse ausgeführt werden sollen. Handelt es sich aber um ein dediziertes Netzwerk, sollte jeder Rechner im Netz entsprechend seiner Leistungsfähigkeit mit Rechenlast bedacht werden. Dies trägt zu einer optimalen Ausnutzung der vorhanden Ressourcen und damit zu einer optimaler Laufzeitverkürzung bei.

Erwägung 4 beschäftigt sich mit der Integrität eines Tests. Im Gegensatz zur Erwägung 2 richtet sich das Augenmerk hierbei nicht auf die Beeinflussung eines Tests oder dessen Ergebnisses durch einen anderen Tests, sondern ausschließlich auf die Beeinflussung durch den Distributionsprozess selbst. Mit einer Distribution sind Risiken wie Netzwerk- oder Rechner-

ausfall verbundenen und diesen muss mit geeigneten Maßnahmen begegnet werden, damit Ergebnisse nicht verfälscht oder gar verloren werden.

Die Notwendigkeit einer zentralen Bedienbarkeit wird in Erwägung 5 behandelt. Allein aus Gründen einer einfacheren Benutzung ist es notwendig, sämtliche Schritte der Durchführung eines Regressionstests von einer zentralen Stelle aus durchführen zu können. Das Verteilen der Tests und das Beziehen der Ergebnisse muss zudem auf eine wohldefinierte Art geschehen; es sollten so wenig Freiheiten wie möglich existieren, da diese nicht nur eine Gefahr für eine Verfälschung von Ergebnissen sondern auch eine Sicherheitsfrage darstellen. Daher ist es sinnvoll, all diese Tätigkeiten fest in das System zu integrieren und über eine zentrale Stelle zugänglich zu machen. Zudem muss ein solches System nicht zwangsläufig auf eine Benutzung in lokalen Netzwerken beschränkt bleiben, somit gibt es unter Umständen gar keine andere Möglichkeit als ein entfernter Zugriff.

Das wichtige Thema der Netzwerksicherheit wird in Erwägung 6 angesprochen. Jede Zugriffsmöglichkeit auf einen Rechner stellt eine potentielle Gefahrenquelle dar. Nicht umsonst werden in jedem Netzwerk umfangreiche Sicherheitsmaßnahmen getroffen. Die hierbei zu berücksichtigenden Aspekte lassen sich in folgende Kategorien einordnen:

Authentizität Die Identität eines Objekts im Netzwerk muss sichergestellt sein

Autorisation Es muss eine ordentliche Rechtevergabe und -durchsetzung erfolgen

Integrität Eine Datenübertragung muss störungsfrei verlaufen

Vertraulichkeit Geheime Daten müssen geschützt bleiben

Verfügbarkeit Ausfälle essentieller Komponenten müssen verhindert oder kompensiert werden

Noch dazu kommt in diesem Fall die Tatsache, dass eine Testdistribution aus der Sicht eines dabei verwendeten Rechners das Ausführen eines fremden Programms bedeutet. Der Distributionsmechanismus muss die Möglichkeit des Korrumpierens eines Rechners verhindern, darf aber eine ordentliche Testausführung nicht unmöglich machen.

2.4. Zusammenfassung

Während die Selektion durch das Treffen einer Auswahl der relevanten Einzeltests die Gesamtmenge der durchzuführenden Tests zu reduzieren versucht, verfolgt die Priorisierung das Ziel, durch Sortieren der Tests z.B. nach Fehleraufdeckungspotential möglichst früh während eines Testdurchlaufs wichtige Ergebnisse zu erlangen. Die Distribution nutzt mehr Ressourcen und verteilt die Ausführung auf mehrere Rechner, stellt aber auch hohe Anforderungen an die Implementation des Distributionsmechanismus.

Die beschriebenen Ansätze nutzen jeweils verschiedene Techniken auf verschiedenen Gebieten, die sich nicht überschneiden. Daher lassen sie sich auch kombinieren. Ein Algorithmus, der alle Ansätze vereint, kann in [12] nachgelesen werden.

3. Distributed JUnit

In diesem Kapitel wird Distributed JUnit vorgestellt, ein Drop-In-Replacement von JUnit, dass auf der Jini-Netzwerk-Technologie und dem ComputeFarm-Framework aufbaut und JUnit-Tests parallel auf prinzipiell beliebig vielen Rechnern ausführt.

Im Anschluss wird die Konzeption von Distributed JUnit dargestellt, wobei zunächst ein abstrakter Überblick gegeben wird. Die verwendeten Frameworks werden im weiteren Verlauf näher betrachtet. Schließlich werden die Implementationsdetails erläutert.

3.1. Konzeption

Distributed JUnit soll ein Drop-In-Replacement von JUnit darstellen. Es ist also nötig, das JUnit-Framework an relevanten Stellen umzuschreiben. Um herauszufinden, welche Stellen modifiziert werden müssen, ist ein genaues Verständnis der Abläufe innerhalb des JUnit-Frameworks nötig. Deren Untersuchung erfolgt im Abschnitt 3.2.

Durch die Aufgabenstellung sind einige Rahmenbedingungen bereits gegeben, z.B. die Festlegung auf Java als Implementationssprache. Daraus ergeben sich weitere Überlegungen, wie Distributed JUnit implementiert werden soll. Java bietet den Vorteil, dass sehr viele abstraktere Funktionen schon in Form von Klassenbibliotheken und Frameworks zur Verfügung stehen. Da Distributed JUnit vor diesem Kontext keine zu speziellen Anforderungen besitzt, um nicht mit Hilfe eines Compute-Server-Frameworks implementiert zu werden, stellt sich nun nur noch die Frage, welches der vielen existierenden Frameworks genutzt werden soll. Jedes Framework hat gewisse systeminhärente Einschränkungen. Die Features und Einschränkungen müssen daraufhin untersucht werden, ob sie die geplante Umsetzung von Distributed JUnit zulassen oder gar unmöglich machen. Die Auswahl kann erfolgen, indem Kriterien festgelegt und potentielle Frameworks anhand dieser Kriterien bewertet werden, was in Abschnitt 3.3 erläutert wird.

Ist die Entscheidung für ein Framework gefallen, muss untersucht werden, welche Konfigurationsschritte, Modifikationen und zusäzliche Klassen im Rahmen des Frameworks für die Umsetzung von Distributed JUnit nötig sind. Dazu ist eine genaue Untersuchung der Funktionsweise und Fähigkeiten des augewählten Frameworks notwendig, was in den Abschnitten 3.4 und 3.5 geschieht.

Schließlich wird in Abschnitt 3.6 auf die Details der Implementation eingegangen. Es werden notwendigen Modifikationen von JUnit und Details zur Konfiguration der Frameworks vorgestellt und die allgemein vorgestellten Funktionsprinzipien am Beispiel von Distributed JUnit konkretisiert.

3.2. JUnit

JUnit ist ein Regressionstest-Framework für die Programmiersprache Java und mittlerweile aus vielen Projekten nicht mehr wegzudenken. Die gute Funktionalität, der einfache Aufbau

des Frameworks und die Tatsache, dass nur ein Minimum an Einarbeitung nötig ist, haben zu einer schnellen und weiten Verbreitung von JUnit beigetragen und es zu einem unverzichtbaren Werkzeug gemacht.

In diesem Abschnitt werden Architektur und interne Funktionsweise beschrieben und es wird ein kurzer Überblick gegeben über die Anwendung von JUnit.

3.2.1. Architektur und Funktionsweise

Die Beschreibung orientiert sich an der Beschreibung der Architektur durch die Autoren des JUnit-Frameworks, Kent Beck und Erich Gamma [14].

Übersicht

Die Funktionalität von JUnit wird grob gesehen durch ein Interface und drei Klassen implementiert:

- `interface Test`
- `class TestCase`
- `class TestSuite` und
- `class TestResult`.

Ein Test setzt sich zusammen aus einem oder mehreren Testfällen, modelliert durch die Klasse `TestCase`. Innerhalb einer `TestSuite` können `TestCase`s oder auch weitere `TestSuite`s zusammengefasst werden. `TestCase` und `TestSuite` implementieren das Interface `Test`, welches eine Methode `run()` definiert, über dessen Aufruf ein Testdurchlauf gestartet wird. Das Testergebnis schließlich wird modelliert durch die Klasse `TestResult`, in welchem die eigentliche Testausführung stattfindet.

Weiterhin von Interesse ist das Benachrichtungssystem von JUnit, dessen Zweck die Protokollierung und Anzeige der während des Testablaufs geschehenden Ereignisse ist. Es besteht aus einem Interface und den drei `TestRunner`n:

- `interface TestListener`
- `class textui.TestRunner`, ein textueller TestRunner.
- `class awtui.TestRunner`, ein graphischer TestRunner, der auf dem AWT aufbaut.
- `class swingui.TestRunner`, ein graphischer TestRunner, der auf Swing aufbaut.

Die `TestRunner` implementieren das Interface `TestListener` und können sich als solche bei einem `TestResult` anmelden, dass diese über die Ergebnisse eines Tests informiert.

Es existieren eine Reihe weiterer Klassen und Interfaces, die Hilfsfunktionen übernehmen bzw. aus objekt-orientierten Gesichtpunkten sinnvoll sind, auf die aber nur dann eingegangen werden soll, falls dies zum Verständnis von Distributed JUnit beiträgt.

Im folgenden wird näher auf die einzelnen Klassen und deren wichtigsten Methoden eingegangen.

Ein einzelner Testfall

Es liegt nahe, einen Testfall als Java-Objekt zu modellieren; dies geschieht in JUnit durch die Klasse `TestCase`. Jeder Testfall hat einen Namen und kann ausgeführt werden. Dies orientiert sich am *Command*-Pattern [7], bei dem eine Operation wie hier ein Testdurchlauf in Form eines Objekts abgebildet wird und das Objekt eine Methode `execute()` besitzt, durch die die Operation angestossen werden kann.

Tests weisen im Allgemeinen eine gemeinsame Struktur auf. Üblicherweise werden zunächst die Objekte deklariert, die zum Testen der Funktionalität benötigt werden. Dann werden Methoden der Objekte ausgeführt und Ergebnisse erzeugt. Diese Ergebnisse werden dann überprüft und schließlich werden die Objekte wieder zerstört und der Ausgangszustand wird wiederhergestellt, damit nachfolgende Tests nicht beeinträchtig werden.

Diese Struktur ist in JUnit in Anlehnung an das *Template Methode*-Pattern [7] dahingehend umgesetzt, als dass für die einzelnen Schritte eines Testdurchlaufs einzelne Methoden vorgesehen sind, in denen die Umsetzung der einzelnen Schritte geschieht: die Methoden `setUp()`, `runTest()` und `tearDown()`. Sie werden in dieser Reihenfolge aufgerufen und stellen somit das Skelett eines Testdurchlaufs dar.

`setUp()` In der Methode `setUp()` werden die bereits erwähnten Objekte deklariert bzw. initialisiert, die zum Testen benötigt werden. Auch sind oft weitere Objekte bzw. primitive Datentypen nötig, die beispielsweise als Parameter bei Methodenaufrufen benutzt werden oder definierte Eingabewerte oder Ausgabewerte darstellen. Diesen Satz von Objekten und Variablen bezeichnet man als *Fixture* (engl. „Inventar“).

`runTest()` In der Methode `runTest()` werden Methoden der Objekte aus dem Fixture aufgerufen und Ergebnisse erzeugt. Um die Überprüfung selbst durchzuführen, wird ein Vergleich von den zurückgegebenen oder durch Seiteneffekte entstandenen Rückgabewerte mit den definierten Soll-Werten definiert. Dabei verlangt man beispielsweise Gleichheit oder andere Relationen. Die Definition dieser Anforderungen erfolgt in Form von *Assertions* (engl. Behauptungen). Sind diese erfüllt, ist der Test bestanden.

`tearDown()` Die Methode `tearDown()` ist schließlich dazu vorgesehen, etwaige, für das Testen benötigte und noch nicht abgeschlossene Vorgänge zu beenden, beispielsweise Netzwerkverbindungen abzubauen etc.

Das Testergebnis

Während der Durchführung eines Testfalls müssen auftretende Ereignisse in geeigneter Weise festgehalten werden. Dazu dient die Klasse `TestResult`. Sie besitzt Methoden, welche aufgerufen werden bei Start und Beendigung eines Tests und im Falle des Auftretens von Exceptions. Eine das `TestListener`-Interface implementierende Klasse wie zum Beispiel ein `TestRunner` kann sich über die `addListener()`-Methode bei einem `TestResult` anmelden und wird dann über alle auftretenden Ereignisse informiert.

`TestResult` implementiert das *Collecting Parameter*-Pattern [13]. Dieses Pattern ist geeignet, wenn Ergebnisse von mehreren Methoden zusammengefasst werden müssen und sieht vor, dass Ergebnis als Objekt zu modellieren, welches als Parameter beim Aufruf der Methoden

übergeben wird und über die im folgenden beschriebenen Methoden die einzelnen Ergebnisse notiert und an die angemeldeten Listerner weitergibt.

Beim Ausführen eines Tests wird eine Instanz von `TestResult` übergeben, die die eigentliche Ausführung des Tests in einem geschützten Bereich kapselt. Vor dem Begin des Tests wird eine Benachrichtigung an alle angemeldeten Listener gesendet. Innerhalb des Bereichs werden dann alle während der Testausführung auftretenden Fehler abgefangen und durch den Aufruf einer dem Fehler entsprechenden Protokollierungsmethode protokolliert. Bei Beendigung des Tests werden die Listener wiederum informiert. Die dazu verwendeten Methoden werden nun vorgestellt.

`startTest()` Die Methode `startTest()` wird zu Beginn der Ausführung eines Tests aufgerufen. Sie ruft dann die `startTest()`-Methode bei allen angemeldeten Listenern auf.

`endTest()` Die Methode `endTest()` wird zu Beendigung der Ausführung eines Tests aufgerufen. Sie ruft dann analog zu `startTest()` die `endTest()`-Methode bei allen angemeldeten Listenern auf.

`addFailure()` Die Methode `addFailure()` wird dann aufgerufen, wenn ein *failure*, also nicht bestandener Test aufgetreten ist. Sie ruft dann die `addFailure()`-Methode bei allen angemeldeten Listenern auf.

`addError()` Die Methode `addError()` wird dann aufgerufen, wenn ein *error*, also ein unvorhergesehene Fehler aufgetreten ist. Zu solchen unvorgesehenen Fehlern zählen beispielsweise eine `NullPointerException` oder `ArrayIndexOutOfBoundsException`. Analog zu `addFailure()` ruft diese Methode die Methode `addError` bei allen angemeldeten Listenern auf.

Die meisten Informationen über einen Testverlauf werden über das Listener-System schon während des Testablaufs weitergegeben. Das `TestResult` kann aber auch eingeschränkt erst nach Abschluss des Tests ausgewertet werden. Dazu besitzt es einige Methoden, mit denen man die Gesamtanzahl der ausgeführten Tests, die Anzahl der `failures` und `errors` und ähliches abfragen kann.

Das Zusammenfassen von Tests

Es ist möglich, Testfälle in einer Suite zusammenzufassen und ebenso können Suites selbst wieder in Suites zusammengefasst werden. Dies geschieht unter Anwendung des *Composite*-Patterns [7], welches Teil-Ganzes-Beziehungen in Form von Baumstrukturen modelliert. Das Interface `Test` stellt hier das gemeinsame Interface eines Blatts bzw. eines Teilbaums dar. Die bereits bekannte Klasse `TestCase` repräsentiert dabei ein Blatt des Baums und ein Verbund von Testfällen, also ein Teilbaum, wird realisiert durch die Klasse `TestSuite`.

Eine Testsuite besitzt eine Liste von Tests, die bei der Ausführung abgearbeitet wird. Es existieren mehrere Möglichkeiten, Tests zu einer Testsuite hinzuzufügen: zum einen manuell durch Aufruf der Methode `addTest()`, die einen `TestCase` oder eine `TestSuite` zu der Liste der Tests hinzufügt, zum anderen über die Methode `addTestSuite()`, die eine Klasse erwartet. Es werden dann per Reflection sämtliche definierten Testmethoden innerhalb der Klasse ausgelesen, in einer Suite vereint und diese hinzugefügt. Dazu muss die Benennung

der Testmethoden allerdings einer Konvention folgen, die vorsieht, dass jede automatisch hinzuzufügende Testmethode mit dem Präfix „test“ beginnen muss. Schließlich können durch Benutzung eines Konstruktors von `TestSuite`, der eine `TestCase`-Klasse als Parameter erwartet, alle in der `TestCase`-Klasse definierten Testmethoden in eine Suite umgewandelt werden. Dabei müssen die Testmethoden derselben, bereits angesprochenen Namenskonvention folgen.

Soweit gestaltet sich die Architektur und die interne Funktionsweise von JUnit. Abbildung 3.1 zeigt die wichtigsten Klassen von JUnit und ihre Beziehungen untereinander.

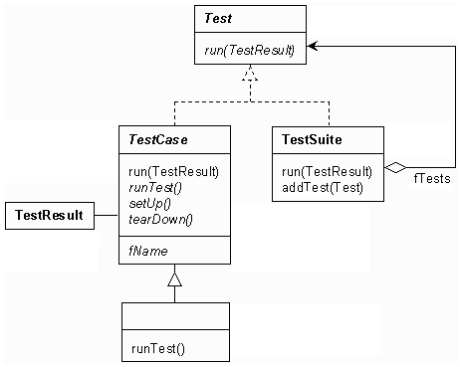

Abbildung 3.1.: Architektur von JUnit

Nun soll der Vollständigkeit halber noch kurz auf die Benutzung eingegangen werden, mehr Informationen sind unter [15] verfügbar.

3.2.2. Implementation eines Tests

Ein Test wird implementiert, indem eine Klasse erstellt wird, die von der Klasse `TestCase` abgeleitet ist. Die Superklasse stellt dann die Methoden `setUp()`, `runTest()` und `tearDown()`, die in der Klasse entsprechend der Testaufgaben überschrieben werden, und eine Reihe von Assertion-Methoden zur Verfügung, die innerhalb der `runTest()`-Methode für die Testimplementation benutzt werden können.

3.2.3. Ausführung eines Tests

Tests werden ausgeführt mit Hilfe einer der drei `TestRunner`, die in JUnit existieren. Die Unterschiede zwischen diesen liegen in der Art der Präsentation. Es existiert ein textueller `TestRunner` und zwei graphische `TestRunner`: einer basierend auf AWT, der andere basierend auf Swing. Sie sind alle von der abstrakten Klasse `BaseTestRunner` abgeleitet, in welcher der Großteil der von der Präsentationsart unabhängigen Funktionalität implementiert ist.

Jeder `TestRunner` besitzt eine `main()`-Methode und kann somit ausgeführt werden. Beim Start wird der auszuführende Test als Kommandozeilenparameter übergeben. Es wird dann ein `TestResult` erzeugt, der Test instantiiert und gestartet.

Die Klasse `BaseTestRunner` implementiert das `TestListener`-Interface und erfährt so von den Ergebnissen der Tests.

3.2.4. Ablauf eines Testdurchlaufs

In Abbildung 3.2 wird vereinfacht dargestellt, wie der Ablauf eines Tests erfolgt. Der in der Abbildung mit „TestCase_j“ bezeichnete `TestCase` steht für die einzelnen in der `TestSuite` enthaltenen Tests. Diese werden nacheinander ausgeführt und interagieren dabei jeweils mit demselben `TestResult`, welches alle Testergebnisse in sich vereint und über den Benachrichtigungsmechanismus den als Listener angemeldeten `TestRunner` über die Ergebnisse informiert. Dort werden sie dann dem Tester präsentiert.

3.2.5. Zusammenfassung

In diesem Abschnitt wurde Aufbau und interne Funktionsweise von JUnit aufgeführt. Im folgenden Abschnitt werden die hier vorgestellten Informationen benutzt, um die Anforderungen von Distributed JUnit an das Distributed-Systems-Framework zu definieren.

3.3. Framework-Auswahl

Während der Beschäftigung mit Distributed JUnit wurde klar, dass die Implementation eines Distributionsmechanismus keine triviale Aufgabe darstellt. Es existierten eine Vielzahl von theoretischen Möglichkeiten, die benötigten Funktionalitäten umzusetzen. Nach eingehender Untersuchung der zu implementierenden Funktionalitäten fiel die Entscheidung, ein geeignetes Framework für die anfallende Netzwerk-Kommunikation und die Aufgaben der Verteilung zu benutzen, um von der Vielzahl untergeordneter Abläufe abstrahieren zu können und von der Durchdachtheit eines Systems zu profitieren.

Auch hier existierten immer noch mehrere Kandidaten, so dass ein definierter Entscheidungsprozess vonnöten war. Dazu wurden zunächst, ausgehend von den Erwägungen in Abschnitt 2.3.1, eine Reihe Kriterien aufgestellt, nach denen die Auswahl erfolgen sollte und die Kandidaten anhand dieser Kriterien beurteilt.

3.3.1. Definition der Kriterien

Mit JUnit war der Kern der Funktionalitäten bereits gegeben und diese Tatsache hat einen großen Einfluss auf die Definition der Kriterien:

Das Framework sollte

- in Java implementiert sein.
- nicht-kommerziell/unter Open-Source-Lizenz vertrieben werden.
- einen ausreichend stabilen Entwicklungsstand erreicht haben.
- wirkungsvolle Sicherheitsmechanismen vorweisen können.
- Mechanismen zu Behandlung von Rechner- oder Netzwerkausfällen beinhalten.
- weitestgehend selbstkonfigurierend sein.
- zumindest eine simple Lastverteilung aufweisen.

- gut dokumentiert sein.

Mit der Aufstellung dieser Kriterien fiel der Großteil der in Frage kommenden Frameworks weg. Die verbliebenen Frameworks werden im Folgenden betrachtet.

3.3.2. Kandidaten

Da eine Nutzung von kommerziellen Produkten aufgrund der akademischen Natur der Arbeit von vorneherein ausgeschlossen war, wurden vorallem Angebote der Open-Source-Gemeinde untersucht. Das größte Portal für Open-Source-Software ist zweifelsohne SourceForge [5], auch die um die Java-Community angesiedelten Portale, die auf verteilte Anwendungen spezialisiert sind, wurden zu Rate gezogen. Die folgenden Frameworks kamen letztendlich in Frage:

- MyGrid[1]
- VGrid[2]
- DistrIT[3]
- ComputeFarm [21]

Tabelle 3.1 gibt einen Überblick über die Features der Kandidaten.

Kriterien	**Frameworks**			
	MyGrid	VGrid	DistrIT	ComputeFarm
in Java implementiert	nein (.NET/mono)	ja	ja	ja
Open Source	ja	ja	ja	ja
Entwicklungsstand	alpha	beta	stable	stable
Sicherheitsmechanismen	keine Aussage	Java Sandbox	Java Sandbox	Java Sandbox
recovery-/fault-tolerance	ja	keine Aussage	ja	ja
selbstkonfigurierend	teilweise	nein	ja	ja
lastverteilend	ja	ja	ja	ja

Tabelle 3.1.: Feature-Überblick

Nachfolgend sind einige Bemerkungen zu den Angaben in Tabelle 3.1 aufgelistet:

Sicherheitsmechanismen Dieser Punkt gibt an, ob Sicherheitsmechanismen zur Verfügung stehen. Bei MyGrid konnte keine Aussage getroffen werden, da dieses Thema nicht dokumentiert ist. *Java Sandbox* bezeichnet das Sicherheitssystem von Java: ein geschützter Bereich, in dem die Ausführung eines Programms gekapselt ist, vgl. Abschnitt 5.6.

recovery-/fault-tolerance Dieser Punkt gibt an, ob Funktionalitäten zur Behandlung von Rechner- oder Netzwerkausfall implementiert sind. Bei VGrid konnte keine Aussage getroffen werden, da dieses Thema nicht dokumentiert ist.

[1] `http://mygrid.sourceforge.net/`
[2] `http://vgrid.sourceforge.net/`
[3] `http://distrit.sourceforge.net/`

selbstkonfigurierend Dieser Punkt gibt an, ob die Komponenten des Frameworks automatisch konfiguriert werden, um miteinander kommunizieren zu können. Dies beinhaltet eine selbstständige Erkennung, selbstständiges Abrufen von Aufgaben und ähnliches. Bei MyGrid werden grundlegende Einstellungen automatisch erkannt, weitere Angaben müssen aber manuell gemacht werden bzw. deren automatische Konfiguration muss selbst implementiert werden.

lastverteilend Dieser Punkt gibt an, ob Mechanismen zur Lastverteilung implementiert sind.

Auf den ersten Blick hat es den Anschein, dass DistrIT und ComputeFarm eine gleichwertige Feature-Ausstattung besitzen. Tatsächlich stellen beide Frameworks annähernd die gleichen Funktionalitäten zur Verfügung, allerdings gibt es einen Unterschied, der letztendlich den Ausschlag gab: Das DistrIT-Framework implementiert seine Funktionalität proprietär mit Hilfe von Java RMI, wohingegen das ComputeFarm-Framework auf der Jini-Netzwerk-Technologie [3] von Sun aufsetzt und die in der Jini-API definierten Methoden für die Netzwerkkommunikation nutzt. Jini baut auf Java RMI auf und stellt eine Vielzahl an Funktionalitäten zur Verfügung, die insbesondere eventuelle zukünftige Erweiterungen von Distributed JUnit sehr vereinfachen.

Die Jini-Netzwerk-Technologie, auch Jini-Framework genannt, die die von dem ComputeFarm-Framework genutzten Funktionalitäten zur Verfügung stellt, soll nun vorgestellt werden.

3.4. Jini

Die Jini-Netzwerk-Technologie ist mehr als nur ein Framework, das einige Funktionalitäten zur Verfügung stellt. Die Jini-Corner F.A.Q.-Seite der Artima Developer Community [1] beantwortet die Frage, was Jini ist, mit der fogenden Antwort:

> „Jini is a set of APIs and runtime conventions that facilitate the building and deploying of distributed systems. Jini provides „plumbing“ that takes care of common but difficult parts of distributed systems.
>
> Jini consists of a programming model and a runtime infrastructure. By defining APIs and conventions that support leasing, distributed events, and distributed transactions, the programming model helps developers build distributed systems that are reliable, even though the underlying network is unreliable. The runtime infrastructure, which consists of network protocols and APIs that implement them, makes it easy to add, locate, access, and remove devices and services on the network.“[4]

Es existiert viel ausführliche Dokumentation zu Jini im Internet. Die Artima Developer Community [1] stellt einen guten Einstiegspunkt dar. Detailierte Informationen und sämtliche Spezifikationen finden sich im *Jini Specification and API Archive* [3] und *Jini.org* [4] ist das Community-Portal, das viele Projekte beherbergt, die die Jini-Netzwerk-Technologie verwenden.

Hier sollen nur die wichtigsten Informationen, die für das Verständnis des ComputeFarm-Frameworks und Distributed JUnit nötig sind, dargestellt werden.

[4] Quelle: `http://www.artima.com/jini/faq.html#whatis`

3.4.1. Allgemeines

Jini kann als *Umgebung für verteilte Systeme, das „network plug-and-play“* beherrscht, bezeichnet werden. Der zentrale Organisationsknotenpunkt in einem Jini-Netzwerk ist der Lookup-Service. Bei dem Lookup-Service können sich Jini-Services registrieren. Clients können diese Services über den Lookup-Service lokalisieren, mit ihnen in Kontakt treten und die angebotenen Dienste nutzen.

Die Methoden zum Auffinden des Lookup-Service, zum Registrieren eines Services beim Lookup-Service, zum Abrufen von Informationen über registrierte Services vom Lookup-Service durch einen Client und zur Kontaktaufnahme eines Clients mit einem Service sind allesamt über Protokolle und APIs definiert. Dadurch kann komplett von dem Netzwerk abstrahiert werden. Ein Client kann über die durch die Jini-Netzwerk-Technologie definierten Prinzipien Services im Netz auffinden und nutzen, ohne etwas über das zugrundeliegende Netzwerk wissen zu müssen. Die genaue Funktionsweise der vorgestellten Konzepte kann unter [3] nachgelesen werden.

Es werden nun die Services beschrieben, die für die Funktion des ComputeFarm-Frameworks im Netzwerk vorhanden sein müssen.

3.4.2. JavaSpaces-Service

Der JavaSpaces-Service stellt einen JavaSpace zur Verfügung. Ein JavaSpace ist eine Java-basierte Implementation eines Tupelraums [6]. Dieser stellt einen gemeinsam nutzbaren Behälter für Objekte dar. Der JavaSpace-Service bietet die Möglichkeit, Objekte in den JavaSpace hineinzuschreiben, Objekte zu lesen und Objekte wieder herauszunehmen. Dies kann nicht nur für Austausch von Objekten sondern auch für die Kommunikation zwischen beteiligten Komponenten genutzt werden.

Operationen auf einem JavaSpace können durch die Verwendung von Transaktionen gesichert werden:

3.4.3. TransactionManager-Service

Der TransactionManager-Service bietet Methoden, um Operationen, bei denen zwei oder mehr Objekte beteiligt sind, in einer Transaktion zu kapseln. Im Falle eines durch eine Transaktion gesicherten Zugriffs eines Clients auf einen JavaSpace stellt der JavaSpace-Service selbst das eine beteiligte Objekt und der Client des JavaSpace-Services das andere beteiligte Objekt dar.

Eine Transaktion kann entweder erfolgreich abgeschlossen werden oder aber misslingen, z.B. wenn der durchführende Client vor dem Abschluss der Transaktion eine Fehlfunktion erleidet oder die Netzwerkverbindung abbricht.

In dem Fall, dass die Transaktion erfolgreich abgeschlossen wird, werden die innerhalb der Transaktion durchgeführten Operationen allgemein übernommen; erst jetzt werden die Auswirkungen der durchgeführten Operationen auch für alle anderen auf den JavaSpace zugreifenden Clients sichtbar.

Sollte eine Transaktion aber misslingen, werden die innerhalb der Transaktion durchgeführten Operationen zurückgenommen.

3.4.4. Bemerkung zum „network plug-and-play“

Jini verlässt sich bei der Abstraktion von dem zugrundeliegenden Netzwerk darauf, das dieses Netzwerk korrekt konfiguriert ist. Eine Misskonfiguration entweder eines verwendeten Rechners oder eines Dienstes wie z.B. eines DNS-Dienstes kann dazu führen, dass die Funktionalität von Jini nicht mehr gewährleistet ist.

Dies schließt die Beschreibung der durch das ComputeFarm-Framework genutzten Services ab. Nun soll das ComputeFarm-Framework betrachtet werden.

3.5. ComputeFarm

ComputeFarm ist ein Compute-Server-Framework nach dem Replicated-Worker-Pattern. Es basiert auf einem JavaSpace und ist gut geeignet bei Berechnungsproblemen, die sich in kleinere Teilpobleme aufteilen lassen.

3.5.1. Funktionsprinzip

Die Funktionalitäten, auf denen das ComputeFarm-Framework aufbaut, im wesentlichen der JavaSpace und Transaktionen, sind als Services im Jini-Framework verfügbar.

Abbildung 3.3 zeigt das Funktionsprinzip des Replicated-Worker-Patterns[5].

Die zentrale Kommunikationsschnittstelle ist hierbei der Compute-Space, eine abstrakte Repräsentation eines JavaSpace (vgl. Abschnitt 3.4.2).

Ein Rechner, Master oder auch Server genannt, ist für die Eingabe von Aufträgen in und das Auslesen der Ergebnisse aus dem Compute-Space zuständig. Eine Reihe anderer Rechner, Worker genannt, beobachten den Compute-Space, entnehmen eingegangene Aufträge, bearbeiten diese und schreiben dessen Ergebnisse zurück in den Compute-Space. Jeder Zugriff auf den Compute-Space wird durch die Verwendung von Transaktionen gesichert (vgl. Abschnitt 3.4.3).

3.5.2. Natürliches Load Sharing

Das Funktionsprinzip des Replicated-Worker-Patterns mit dem Compute-Space sorgt für ein natürliches Load Sharing der anfallenden Last auf alle partizipierenden Worker. Jeder Worker arbeitet entsprechend seiner Leistungsfähigkeit einen Auftrag ab. Solange Aufträge im Compute-Space vorhanden sind, werden alle Worker gleichmäßig ausgelastet. Diese einfache aber wirkungsvolle Art der Lastverteilung ist für diese Zwecke mehr als ausreichend.

3.5.3. Frameworkaufbau

Das ComputeFarm-Framework definiert einige Klassen, von denen die beiden wichtigsten hier betrachtet werden sollen. Die Klasse `Job` stellt ein zu verteilendes Berechnungsproblem dar, das durch den Compute-Server abgearbeitet werden soll. Die Klasse `Task` stellt einen einzelnen Auftrag dar, der von einem Worker übernommen wird.

Von der Klasse `Job` muss abgeleitet werden und in dieser Ableitung müssen die Methoden `generateTasks()` und `collectResults()` überschrieben werden. In diesen Methoden wird

[5] Quelle: `http://www.jini.org/meetings/eighth/White/img2.html`

definiert, auf welche Art ein Job in einzelne `Task`s aufgeteilt wird und auf welche Weise die Ergebnisse der `Task`s eingeholt werden sollen. Sie werden während der Ausführung des Jobs aufgerufen, wobei eine Instanz des zur Verfügung stehenden `ComputeSpace` übergeben wird.

Von der Klasse `Task` muss ebenfalls abgeleitet werden. Die Klasse `Task` implementiert das *Command-Pattern* [7]. Es existiert auch nur die eine durch das *Command-Pattern* vorgeschriebene Methode `execute()`, welche überschrieben werden muss. In ihr wird definiert, was beim Ausführen des `Task`s geschehen soll.

Mit den nun beschriebenen Funktionalitäten kann Distributed JUnit implementiert werden.

3.6. Implementation

Nach der Beschreibung der Funktionsweisen der verwendeten Frameworks bleibt nun nicht mehr übrig, als das Zusammenfügen der einzelnen Teilfunktionen zu einem funktionierenden Ganzen.

3.6.1. Architektur

Abbildung 3.4 zeigt eine High-Level-Ansicht des Aufbaus von Distributed JUnit

3.6.2. Erstellte Klassen und deren Funktion

Distributed JUnit nutzt das ComputeFarm-Framework, um eine auszuführende Testsuite, deren `run()`-Methode von einem `TestRunner` aufgerufen wurde, in einen Job umzusetzen, diesen Job unter Zuhilfenahme des ComputeFarm-Frameworks und der von diesem benötigten Jini-Services ausführen zu lassen und die in dem Job gesammelten Testergebnisse anzuzeigen.

Dazu wurden die Klassen `TestJob` erstellt. Ihr Konstruktor nimmt eine `TestSuite` und ein `TestResult` entgegen. In der in dieser Klasse vorhandenen Methode `generateTasks()` werden nun die einzelnen in der Testsuite enthaltenen `TestCase`s extrahiert und in `Task`s umgesetzt. Diese `Task`s werden dann in den beim Aufruf der Methode übergebenen `ComputeSpace` hineingeschrieben, wobei eine Transaktion verwendet wird.

Analog dazu wurde dazu die Klasse `TestTask` erstellt. Ihr Konstruktor kann mit einer Instanz eines `TestCase` aufgerufen werden. In ihrer `execute()`-Methode wird die `run()`-Methode des enthaltene Tests aufgerufen.

Ein Worker, der den `ComputeSpace` beobachtet, liest den `TestTask` unter Verwendung einer Transaktion aus und führt ihn durch Aufruf der Methode `execute()` aus. Damit wird auf dem Worker der einzelne im `TestTask` enthaltene `TestCase` ausgeführt. Das dabei zurückgegebene `TestResult` wird von dem Worker wieder zurück in den `ComputeSpace` geschrieben. Erst wenn dieses erfolgreich geschehen ist, gilt die Transaktion als erfolgreich beendet.

Zurück in der Methode `collectResults()` der Klasse `TestJob` werden die Testergebnisse aus dem beim Aufruf der Methode übergebenen `ComputeSpace` wiederum unter Verwendung einer Transaktion ausgelesen.

Dabei wird jedes ausgelesene `TestResult` mit dem bei der Instantiierung des `TestJob`s übergebenen `TestResult` vereinigt, wobei für alle während der Testausführung auf dem Worker aufgetretenen und in dem zurückgegebenen `TestResult` protokollierten Ereignisse nun die entsprechenden Listener-Methoden aufgerufen werden (vgl. Abschnitt 3.2.1), was sicherstellt, dass ein als Listener angemeldeter `TestRunner` direkt nach dem Auslesen des Ergebnisses aus dem Compute-Space von dem Testergebnis unterichtet wird und seine Benutzeroberfläche entsprechend aktualisieren kann.

3.6.3. Modifikationen von JUnit

Um JUnit in Verbindung mit dem ComputeFarm-Framework nutzen zu können, mussten die gegebenen Einschränkungen des Frameworks auf den JUnit-Kern übertragen werden. Daher mussten einige Klassen modifiziert werden. Hauptsächlich musste Serialisierbarkeit hergestellt werden.

`TestCase` `TestCase` wurde dahingehend modifiziert, dass nun das Interface `Serializable` implementiert wird.

`TestResult` `TestResult` wurde ebenfalls dahingehen modifiziert, dass nun das Interface `Serializable` implementiert wird. Allerdings war hier das Hinzufügen der Serialisierung- bzw. Deserialisierungsmethoden `writeObject()` und `readObject()` notwendig.

Weiterhin wurde die Methode `addTestResult()` hinzugefügt. Diese wird dazu benötigt, die einzelnen durch die Worker erarbeiteten Testergebnisse zu einem Testergebnis hinzuzufügen, welches dem Tester präsentiert wird. Für diese Funktionalität war es allerding notwendig, eine Liste aller Tests zu pflegen, deren Ergebnisse sich in diesem Testergebnis befinden. Dazu wurde ein Vector `fRunnedTests` hinzugefügt, welcher über die Methode `getRunnedTests()` ausgelesen werden kann.

`TestSuite` `TestSuite` wurde am weitestgehenden modifiziert. Diese Klasse stellt die Schnittstelle zum Distributionsmechanismus dar und wurde daher in ihrer `run()`-Methode um die für die Kontaktaufnahme mit dem Jini-Netzwerk und dem Einspeisen der Tests in das System benötigten Befehle ergänzt.

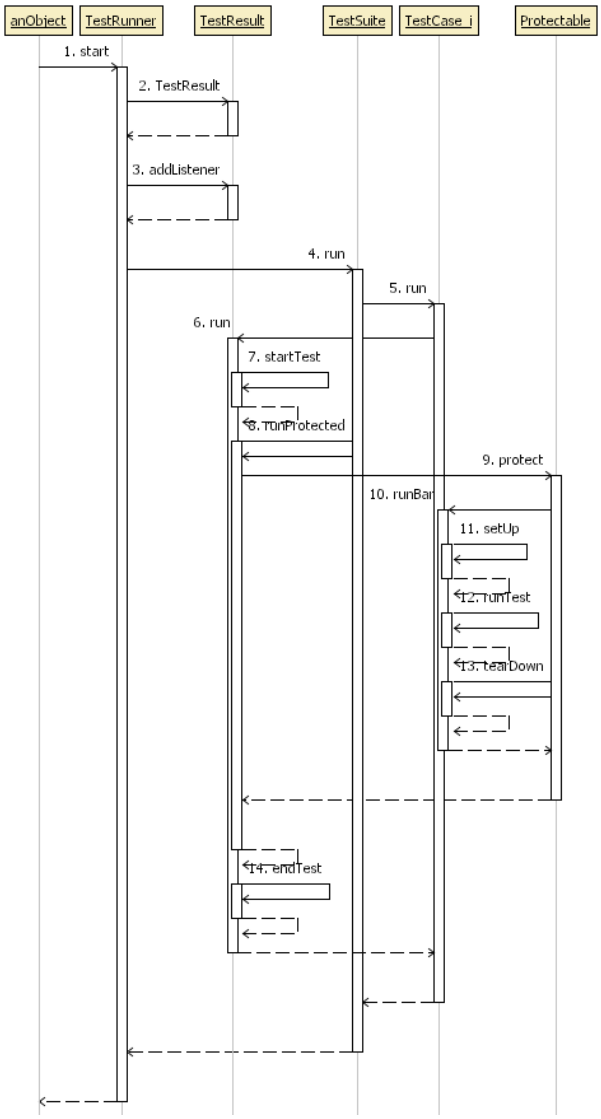

Abbildung 3.2.: Ablauf eines JUnit-Tests (vereinfacht)

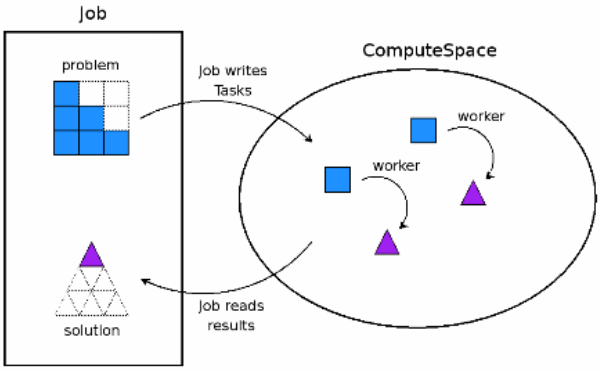

Abbildung 3.3.: Das Replicated-Worker-Pattern

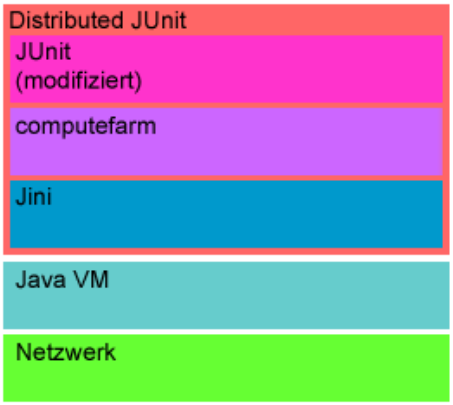

Abbildung 3.4.: Architektur von Distributed JUnit

4. Performance-Untersuchung

In diesem Kapitel wird die Performance von Distributed JUnit untersucht. Herausgefunden werden soll, in wie weit die Durchführung eines Tests durch den Distributionsmechanismus beschleunigt werden kann und in wie weit das Beschleunigungspotential von Art und Umfang des Tests abhängig ist. Ebenso ist die Ausprägung des nicht zu umgehenden zeitlichen Overheads, der bei der Verteilung durch die Kommunikation zwischen Server und Workern entsteht, von Interesse.

4.1. Bemerkungen

4.1.1. Verteilungsoverhead

Bei gegebener Netzwerk- und Rechnerinfrastruktur und der momentanen Implementation von Distributed JUnit ist der Verteilungsoverhead von den folgenden Parametern abhängig:

Testgröße Je größer ein Test, desto länger dauert dessen Beziehung über das Netzwerk und damit steigt der Verteilungsoverhead.

Laufzeit Je kürzer die Laufzeit eines Tests, desto höher ist der relative Anteil des Verteilungsoverheads an dessen Laufzeit und damit steigt der Verteilungsoverhead.

Da Tests z.T. sehr unterschiedliche Ausprägungen haben können, was Laufzeit und erzeugte Prozessorlast angeht, wurden in den Testsuites der Testreihen zwei verschieden geartete Tests verwendet:

PassTest Ein extrem kurzer, aus nur einer Anweisung bestehender Test, die zudem nur eine minimale Prozessorlast ezeugt.

RandomTest Ein Test, der Berechnungen durchführt, die eine hohe Prozessorlast erzeugt, wobei die Anzahl der Durchführung dieser Berechnungen zusätzlich zufällig variiert wird. Dadurch ist die Laufzeit zwar keine Konstante, dies wird aber durch eine Mittelwertbildung und zusätzlich durch die hohe Anzahl an Tests pro Testsuite relativiert.

Die Quelltexte der Tests können im Anhang A nachgeschlagen werden.

4.1.2. Messwertvollständigkeit

Ein verteilter Testdurchlauf unter der Verwendung einer Testsuite mit 46656 enthaltenen Einzeltests hätten eine Laufzeit von über 24h gehabt, daher wurden diese nur unverteilt ausgeführt.

4.2. Testkonfiguration

Die Testreihen wurden im Rechnerpool des Instituts für Software Engineering der Universität Hannover durchgeführt. Es standen acht identisch konfigurierte und mit identischer Hardware ausgestattete Testrechner zur Verfügung. Tabelle 4.1 listet deren Hardware- und Software-Konfiguration auf.

Prozessor:	Intel Pentium 4
Prozessorfrequenz:	2,6 GHz
Hauptspeicher:	1 GB
Betriebssystem:	Debian Linux Unstable, Kernel 2.6.7
Java-Runtime:	Sun Java Version 1.4.2_05, Hotspot Client
Jini Distribution:	Jini 2.0.1, JRMP transient
Distributed JUnit:	djunit Version 0.1 (JUnit 3.8.1, ComputeFarm 0.8.3)

Tabelle 4.1.: Konfiguration der Testrechner

4.3. Überblick über die durchgeführten Testreihen

Insgesamt wurden 410 Testläufe durchgeführt. 210 Testläufe wurden mit einer Testsuite durchgeführt, in der PassTest verwendet wurde. 200 Testläufe wurden mit einer Testsuite durchgeführt, in der der RandomTest verwendet wurde.

Die Testsuiten wurden in in der Anzahl der enthaltenen Einzeltests zwischen einem und max. 46656 identischen Tests variiert.

Zum Vergleich mit einer nicht-verteilten Ausführung wurde jede Testsuite unter Verwendung einer lokalen Simulation eines ComputeSpaces durchgeführt. Die lokale Simulation eines ComputeSpace ist extrem viel schneller als der während einer tatsächlichen Verteilung verwendete ComputeSpace und kommt der Ausführungsgeschwindigkeit eines Tests mittels dem originalen JUnit nahe genug, um aussagekräftige Ergebnisse zu erhalten.

Jede Testsuite wurde unter Verwendung von einem bis acht Workern ausgeführt, wobei auf jedem Rechner genau ein Worker gestartet wurde. Die Testsuiten, die den Randomtest verwendeten, wurden auch in einer Konfiguration mit 16 beteiligten Workern ausgeführt, wobei auf jedem Rechner jeweils zwei Worker gestartet wurden. Die Messwerte waren aber bis auf einen leicht geringeren Beschleunigungsfaktor identisch mit den Werten bei acht eingesetzten Workern, daher werden diese Werte nicht weiter beachtet.

Bei jedem Testlauf wurde die Laufzeit in Sekunden protokolliert. Jede Testsuite wurde zehn Mal unverändert duchrgeführt und aus den Ergebnissen wurde der Mittelwert gebildet. Dies ergibt 41 Messergebnisse.

4.4. Messergebnisse

Tabelle 4.2 zeigt die mittlere Laufzeit in Sekunden bei Verwendung des PassTests.

Tabelle 4.3 zeigt die mittlere Laufzeit in Sekunden bei Verwendung des RandomTests.

Zeit (sek.) / Anzahl Tests	Anzahl Worker 1 (lokal)	1 (remote)	4 (remote)	8 (remote)
1	1,0894	1,2551	1,3388	1,2643
4	1,0904	1,2493	1,334	1,2557
27	1,0939	2,372	2,4215	2,2506
256	1,0921	11,7414	10,3836	9,0782
3125	4,1005	98,3628	92,0131	82,0461
46656	47,0779			

Tabelle 4.2.: Mittlere Laufzeiten bei Verwendung des PassTests

Zeit (sek.) / Anzahl Tests	Anzahl Worker 1 (lokal)	4 (remote)	8 (remote)	16 (remote)
1	1,1937	1,9496	1,4651	1,7499
4	2,0924	1,9542	2,3609	2,4623
27	9,3073	3,9504	3,6857	3,8827
256	104,8122	25,2945	15,3748	16,4792
3125	1234,7725	291,9431	156,8665	158,921
46656				

Tabelle 4.3.: Mittlere Laufzeiten bei Verwendung des RandomTests

4.5. Auswertung

Aus den Messergebnissen lassen sich Metriken wie

- Beschleunigungsfaktor
- Verteilungsoverhead

ableiten. Jede Metrik lässt sich in Abhängigkeit von Anzahl der Einzeltests in der ausgeführten Testsuite und Workeranzahl darstellen.

4.5.1. PassTest

Für die Berechnung des Verteilungsoverhead wurde folgende Formel verwendet:

$$verteilungsoverhead = \frac{(laufzeit_{verteilt} - laufzeit_{unverteilt})}{laufzeit_{verteilt}} \tag{4.1}$$

Mit dieser Formel wurden die in Tabelle 4.4 aufgeführten Werte errechnet:

Der in Tabelle 4.4 und Abbildung 4.1 dargestellte Verteilungsoverhead ergibt sich aus dem Verhältnis der Differenz der Laufzeit bei unverteilter Ausführung und der Laufzeit bei verteilter Ausführung zu der Laufzeit bei verteilter Ausführung. Die Gesamtlaufzeit eines Testdurchlaufs setzt sich zusammen aus der Laufzeit des eigentlichen JUnit-Tests und der zusätzlichen Zeit, die auf den Verteilungsoverhead zurückzuführen ist. Da sämtliche verwendeten Rechner identische Performancespezifikationen besaßen, kann man davon ausgehen, dass die Laufzeit des eigentlichen JUnit-Tests immer gleich gewesen ist, daher muss die bei der verteilten Ausführung zusätzlich verstrichene Zeit allein dem Verteilungsoverhead zuzusprechen sein.

Verteilungsoverhead (%) Anzahl Tests	Anzahl Worker 1 (lokal)	4 (remote)	8 (remote)	
1	0%	13%	19%	14%
4	0%	13%	18%	13%
27	0%	54%	55%	51%
256	0%	91%	89%	88%
3125	0%	96%	96%	95%

Tabelle 4.4.: Verteilungsoverhead des PassTests in Prozent der Gesamtlaufzeit

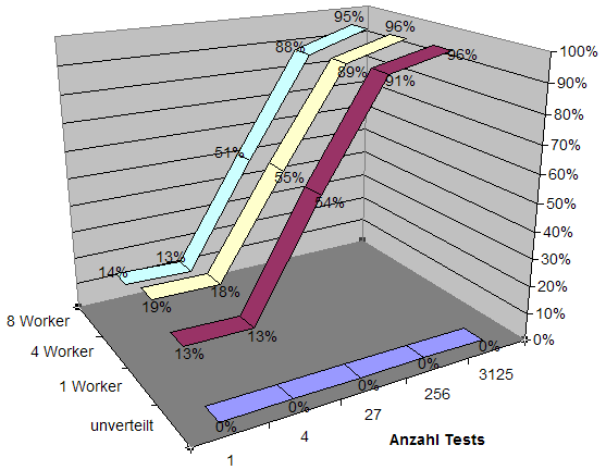

Abbildung 4.1.: Verteilungssoverhead des PassTests in Prozent der Gesamtlaufzeit

Es ist zu erkennen, dass der Verteilungsoverhead bei einer hohen Anzahl von sehr kurzen Einzeltests klar den hauptsächlichen Anteil an der Gesamtlaufzeit hat. Im extremen Fall beträgt er über 96% der Laufzeit. Gut sichtbar ist auch, dass der Verteilungsoverhead unabhängig von der Anzahl der verwendeten Worker ist, was darauf zurückzuführen ist, dass in der verwendeten Implementation von Distributed JUnit allein der Server die für eine Verteilung notwendigen Vorbereitungen durchführt.

4.5.2. RandomTest

Für die Berechnung des Beschleunigungsfaktors wurde folgenden Formel verwendet:

$$beschleunigungsfaktor = \frac{laufzeit_{unverteilt}}{laufzeit_{verteilt}} \tag{4.2}$$

Mit dieser Formel wurden die in Tabelle 4.5 aufgeführten Werte errechnet:

In Tabelle 4.5 und Abbildung 4.2 ist der Faktor der Beschleunigung der Testausführung dargestellt. Dieser ergibt sich aus dem Verhältnis der Laufzeit bei unverteilter Ausführung und der Laufzeit bei verteilter Ausführung. Es ist zu erkennen, dass der Beschleunigungsfaktor bei steigender Anzahl von Einzeltest zunächst recht steil und im weiteren Verlauf weniger steil ansteigt. Erwartungsgemäß wird diese Kurve bei immer weiter steigender Anzahl von Einzeltest immer flacher werden. Aus der Abbildung ist ebenfalls zu gut erkennen, dass der Beschleunigungsfaktor mit der Anzahl der bei der Verteilung eingesetzten Rechner korreliert.

Beschleunigungsfaktor Anzahl Tests	Anzahl Worker 1 (lokal)	4 (remote)	8 (remote)
1	1	0,612279442	0,814756672
4	1	1,070719476	0,886272184
27	1	2,356039895	2,525246222
256	1	4,143675503	6,817142337
3125	1	4,229506366	7,871503476

Tabelle 4.5.: Beschleunigungsfaktor des RandomTests

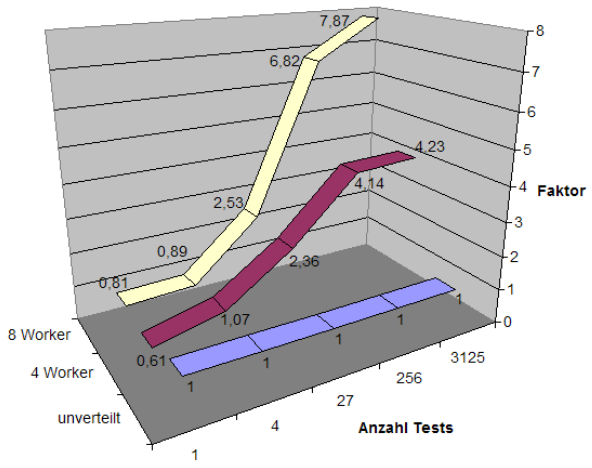

Abbildung 4.2.: Beschleunigungsfaktor des RandomTests in Abhängigkeit der Testanzahl

In Abbildung 4.3 wird die gemessene Skalierung dargestellt. Es wurden wiederum die Werte aus Tabelle 4.5 verwendet. Aufgrund der Begrenztheit der zur Verfügung stehenden Ressourcen kann leider nur ein kleiner Wertebereich erfasst werden, trotzdem ist zu erkennen, dass Distributed JUnit anfangs nahezu linear skaliert und dies bei weiter steigender Anzahl von Rechnern im weiteren Verlauf etwas abschwächt. Zu erkennen ist weiterhin, dass der Punkt der Abweichung von einer linearen Skalierung abhängig ist von der Anzahl der Einzeltests. Bei steigender Einzeltestanzahl tritt dieser Punkt später auf.

4.6. Zusammenfassung

Distributed JUnit verteilt Testsuiten mit sehr vielen und sehr kurzen Tests nur ungenügend performant. Der Verteilungsoverhead macht in diesem Fall weit über neun Zehntel der Gesamtlaufzeit der Testsuite aus. Dies könnte durch das Zusammenfassen mehrerer kleinerer Tests in einen `TestTask` verbessert werden.

Bei Testsuiten mit wenigen, aber großen Tests, die jeweils eine lange Laufzeit haben, liegt der Beschleunigungsfaktor von Distributed JUnit nahe am theoretischen Maximum.

Abschließend kann bemerkt werden, dass, je länger die Laufzeit eines Einzeltests ist, der verteilt ausgeführt wird, desto besser skaliert das System.

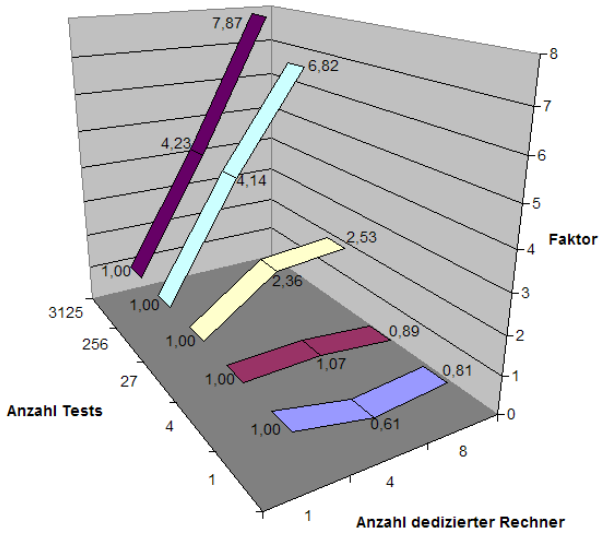

Abbildung 4.3.: Beschleunigungsfaktor des RandomTests in Abhängigkeit der Rechneranzahl

5. Diskussion

Die für den Distributionsansatz allgemein geltenden charakteristischen Problematiken liegen in der Umsetzung der aus Abschnitt 2.3.1 bekannten Erwägungen. In den folgenden Abschnitten wird daher nun auf die bei Distributed JUnit konkret vorliegenden Eigenheiten der Umsetzung dieser Erwägungen eingegangen.

5.1. Transparente und Automatisierte Distribution

Dieser Abschnitt behandelt die Ergebnisse von Distributed JUnit, die in den Bereich der Transparenz und der Automatisierung der Distribution fallen.

5.1.1. Transparenz

Absolute Transparenz ist dann erreicht, wenn für den Test nicht erkennbar ist, ob er verteilt oder nicht-verteilt ausgeführt wird. Dies wäre der Fall, wenn alle Funktionalitäten, die in der Ausführungumgebung eines verteilten Tests nicht mehr existieren, mit den noch verbliebenen Funktionalitäten nachgebildet werden.

Allgemeine Überlegungen zur Transparenz

Wie auch an den nachfolgenden Problematiken erkennbar, ist die Thematik der Transparenz höchst komplex. Eine vollständig transparente Verteilung kann höchstwahrscheinlich nur dann erreicht werden, wenn die Laufzeitumgebung selbst simuliert wird. Es ist immerhin fraglich, ob beim Distributionsansatz eine vollständige Transparenz überhaupt verlangt werden sollte. Solange die Einschränkungen bekannt sind, kann man schon beim Erstellen der Tests entsprechende Maßnahmen ergreifen.

Serialisierbarkeit von Tests

Ein großer Schwachpunkt der Umsetzung von Distributed JUnit liegt in der Notwendigkeit, dass ein Test, der verteilt ausgeführt werden soll, vollständig serialisierbar sein muss. Der Grund für diese Tatsache liegt in der Funktionsweise des Versendens eines Objekts über das Netzwerk. Ein Test ist aber nur dann vollständig serialisierbar, wenn alle im Test referenzierten Objekte ebenfalls serialisierbar sind, so dass bei der Deserialisierung der gesamte Objektbaum wiederhergestellt werden kann.

Die in einem Test referenzierten Objekte sind üblicherweise die Objekte, deren Funktionalitäten getestet werden sollen. Man muss davon ausgehen, dass nicht wenige Objekte nicht serialisierbar sind. Das Problem nicht serialisierbarer Tests kann nun auf zwei Arten angegangen werden.

Eine naive, aber recht einfach realisierbare Herangehensweise ist, dass während der Umsetzung eines Tests in einen `TestTask` festgestellt wird, ob der Test serialisierbar ist. Trifft

dies zu, kann normal fortgeschritten werden. Triff dies aber nicht zu, wird der `TestTask` entsprechend markiert. Alle auf diese Weise markierten `TestTask`s werden dann von einem lokal vorhandenen Worker abgearbeitet, so dass die Notwendigkeit einer Serialisierung überhaupt nicht mehr auftritt.

Der Nachteil dieser Herangehensweise ist, dass bei Testsuites, in denen bis auf wenige Ausnahmen kein Test serialisierbar ist, nur noch diese wenigen Ausnahmen überhaupt verteilt ausgeführt werden können. Die Effizienz einer solchen Implementation wäre in diesem Fall äußerst gering.

Die zweite Herangehensweise bedeutet, dass diejenigen in einem Test referenzierten Objekte, die nicht serialisierbar sind, auf eine andere Art zur Verfügung gestellt werden, wenn sie auf einem entfernten Rechner benötigt werden, um den Objektbaum eines Tests wiederherzustellen. Eine eventuelle Möglichkeit wäre die Umsetzung solcher Objekte in Jini-Services, die dann regulär über den Lookup-Service von Jini aufgefunden und genutzt werden können.

Ressourcendateien

Eine weitere Einschränkung von Distributed JUnit liegt darin, dass Ressourcendateien, die während der Ausführung eines Tests von der Festplatte nachgeladen werden sollen, üblicherweise nicht gefunden werden, da natürlich auf das lokale Dateisystem des ausführenden Workers zugegriffen wird und nicht auf das Dateisystem des Servers, in welchem der Test und alle benötigten Ressourcendateien gespeichert sind.

Eine Möglichkeit wäre, dass dafür gesorgt wird, dass benötigte Ressourcendateien auf das Dateisystem des ausführenden Rechners kopiert werden. Es ist allerdings problematisch, die benötigten Dateien zu determinieren. Theoretisch ist es zwar möglich, allein anhand eines als Quelltext oder Bytecode vorliegenden Tests schon vor dessen Ausführung zu bestimmen, ob in diesem Test auf Dateien zugegriffen wird, indem der Test daraufhin untersucht wird, ob in diesem Befehle verwendet werden, die eine IO-Operation nach sich ziehen, allerdings ist es fraglich, ob dies auch genügend performant durchgeführt werden kann.

Abschließende Bemerkung zur Transparenz

Distributed JUnit ist noch auf einer zu wenig fortgeschrittenen Entwicklungsstufe, um endgültige Behauptungen aufstellen zu können. Die hier angesprochenen Problematiken, die auf nicht genügende Transparenz zurückzuführen sind, stellen allerdings für viele Tests keine unüberwindbaren Einschränkungen dar. Man darf annehmen, dass selbst bei einer Umsetzung des naiven Ansatzes zur Behebung des Problems nicht-serialisierbarer Tests die Performance von Distributed JUnit genügend hoch ist, um den Distributionsansatz als ein effizientes Mittel zur Laufzeitverkürzung anzusehen.

5.1.2. Automatisierte Distribution

Die Installation und auch die Nutzung von Distributed JUnit stellt nur wenige Anforderungen an den Tester. Dank der Verwendung von Jini konnte der Konfigurationsaufwand für das Einrichten eines lauffähigen Systems minimiert werden. Dieser beschränkt sich im Optimalfall auf das Starten der benötigten Jini-Services und mindestens eines ComputeFarm-Clients. Ist das System einsatzfähig, wird ein Testdurchlauf vollständig automatisiert verteilt: In der Bedienung existiert praktisch kein Unterschied zwischen Distributed JUnit und JUnit.

5.2. Testkontamination

Distributed JUnit implementiert zur Zeit keinen Mechanismus zur Verhinderung von Testkontamination. Daher ist Distributed JUnit noch nicht für einen Einsatz anstelle JUnit zu empfehlen. Für die Verhinderung von Testkontamination sind, wie bereits in Abschnitt 2.3.1 erwähnt, Metadaten nötig, aus denen hervorgeht, welcher Test welchen anderen Test kontaminiert. Denkbar wäre eine Integration dieser Metadaten in Form einer statischen Methode, die bei der Umsetzung einer Testsuite in einen `TestJob` für jeden Test ausgeführt wird und so ein Abhängigkeitsgraph erstellt wird, der durch geeignete Methoden aufgelöst werden kann. Der Aufwand zur Implementation dieser Funktionalität dürfte übersichtlich sein.

5.3. Lastverteilung

Das Lastverteilungsverhalten von Distributed JUnit ist auf die Eigenschaft des natürlichen Load-Balancing des Replicated-Worker-Patterns zurückzuführen (vgl. Abschnitt 3.5.2). Während der Durchführung der Performance-Tests konnte eine äußerst gleichmäßige Belastung der Worker beobachtet werden, was aber auch darauf zurückzuführen ist, dass die Granularität der beim Testen verwendeten Testsuiten sehr hoch war.

Denkbar wäre natürlich eine Situation, in der ein besonders rechenintensiver und langer Test ausgerechnet von dem Worker auf dem leistungsärmsten Rechner ausgeführt wird und dadurch die Gesamtlaufzeit des Tests höher ausfällt, als wenn die Verteilung anders erfolgt wäre. Diese Situation könnte verhindert werden, indem Worker und Tests mit Attributen ausgestattet werden, die über Leistungsfähigkeit bzw. Komplexität Aufschluss geben und ein als leistungsarm definierter Worker nur Tests mit geringer Komplexität ausführt, solange solche noch vorhanden sind.

Im Allgemeinen ist das Lastverteilungsverhalten von Distributed JUnit im Rahmen normaler Anwendung mehr als ausreichend.

5.4. Testintegrität

Das ComputeFarm-Framework sorgt durch die Verwendung des durch Jini zur Verfügung gestellten Konzepts der Transaktion bei der Verteilung der `TestTasks` dafür, dass ein Test vollständig durchgeführt und das Testergebnis korrekt in den Computespace geschrieben wird, bevor der `TestTask` aus dem Computespace entfernt wird. Sollte ein Worker bei der Arbeit abstürzen oder die Netzwerkverbindung abbrechen, wird der `TestTask` von einem anderen Worker bearbeitet. Dadurch kann das Verlieren von einzelnen Testergebnissen effektiv verhindert und das Gesamtergebnis vor einer Verfälschung bewahrt werden.

5.5. Ausführungskontrolle

Ist das Distributed JUnit-System aktiv, lässt es sich komplett von einem Rechner aus bedienen. Die einzelnen Ergebnisse der auf den Workern durchgeführten Tests werden komplett selbstständig durch das System wieder auf dem Rechner in einem Gesamtergebnis vereint und dargestellt. Tests können mit jedem der drei `TestRunner` durchgeführt werden. Durch die Tatsache, dass Distributed JUnit interface-kompatibel mit JUnit ist, kann Distributed JU-

nit auch aus anderen Programmen heraus wie JUnit genutzt werden und käme damit sogar z.B. für eine Integration in eine Entwicklungsumgebung in Frage.

Das System kann sogar komplett von einem Rechner aus gestartet werden, allerdings nur unter der Voraussetzung, dass man auf alle in das System miteinzubeziehenden Rechner einen Zugriff hat, der es erlaubt, den ComputeFarm-Client zu starten, falls dieser schon installiert ist, bzw. einen Zugriff, der es erlaubt, den ComputeFarm-Client zu installieren und dann zu starten. Die Installation des ComputeFarm-Client beschränkt sich auf das Kopieren und Entpacken einer Archivdatei.

5.6. Netzwerksicherheit

Die Java Virtual Machine mit dem Sand-Box-Prinzip in Verbindung mit dem `SecurityManager` sorgen im allgemeinen für eine hohe Sicherheit beim Ausführen von Java-Applikationen. Nicht zuletzt ist dieses System für das unbedenkliche Ausführen von Java-Applets notwendig geworden und ist inzwischen sehr ausgereift. Zusätzlich dazu bietet Java natürlich verschlüsselte Verbindungen und die Nutzung von Zertifikaten. Jini profitiert in vollem Umfang von diesem umfassenden Sicherheitskonzept und bietet die Möglichkeit, Rechte in sehr feiner Auflösung zu gewähren bzw. unerlaubte Aktivitäten wirkungsvoll zu verhindern. Dabei können Rechte für Programme, Packages und sogar einzelne Klassen bzw. deren Instanzen bestimmt werden. Es existieren eine große Menge an Rechten, die explizit gesetzt werden müssen, bevor eine Aktion vom `SecurityManager` gewährt wird. Beispiele dafür sind die Rechte der Klassen

- `java.io.FilePermission`: read, write, execute, delete
- `java.net.SocketPermission`: accept, connect, listen, resolve

Jeder von Distributed JUnit genutzte Service kann durch das Editieren einer Konfigurationsdatei, *Policy*-Datei genannt, in seinen Sicherheitseinstellungen konfiguriert werden. Ebenso ist dies für den ComputeFarm-Client möglich. Dadurch kann Distributed JUnit entsprechend jeder Sicherheitsrichtlinie konfiguriert werden.

Da dies eine akademische Arbeit ist, wurde der Aspekt der Sicherheit nur in geringem Umfang betrachtet. Auch würde Distributed JUnit höchstwahrscheinlich hauptsächlich in nicht von aussen zugänglichen, nur von einer Arbeitsgruppe von Entwicklern genutzten, lokalen Netzen sein.

Dennoch bleibt der Aspekt der Sicherheit wichtig, vorallem wenn man bedenkt, dass Distributed JUnit theoretisch auch unter Benutzung anonymer Rechner im Internet ausgeführt werden könnte. In diesem Fall müsste Distributed JUnit wie ein Webservice betrachtet werden und es müsste größte Sorgfalt bei der Sicherheitskonfiguration an den Tag gelegt werden.

5.7. Performance

Während der Performance-Evalutation wurden einige Punkte sichtbar, die vielleicht nicht zwangsweise verbesserungsbedürftig aber immerhin verbesserungsfähig sind. Wie schon im Abschnitt 4.5.1 erwähnt, werden die zur Verfügung stehenden Worker erst bei der Abarbeitung der `TestTask`s verwendet. Die Umwandlung der in einer Testsuite enthaltenen einzelnen Testfälle in `TestTask`s geschieht zur Zeit komplett auf dem Server. Dieser Vorgang könnte

ebenfalls auf die zur Verfügung stehenden Worker verteilt werden und würde dadurch den Verteilungsoverhead bei Testsuiten mit sehr vielen, sehr kleinen Tests massiv senken.

Weiterhin könnte eine Zusammenlegung kleinerer Tests in einen `TestTask` den Verteilungsoverhead signifikant erniedrigen.

Es wurde in den Messungen eine nahe am theoretisch möglichen Maximum liegende Performancesteigerung bei zudem nahezu lineare Skalierung erreicht. Es ist diskussionswürdig, ob diese Ergebnisse auf die Künstlichkeit der bei der Messung verwendeten Tests zurückzuführen sind und ob bei echten Tests eine ähnliche Leistung erbracht werden würde. Dennoch darf man annehmen, dass im Rahmen normaler Anwendung die Performance zumindest zufriedenstellend, wenn nicht besser, ist.

6. Zusammenfassung und Ausblick

Kurze Laufzeiten beim Testen fördern Akzeptanz und damit Nutzen von Tests und führen letztendlich zu qualitativ besserer Software. Ziel dieser Arbeit war die Untersuchung des Distributionsansatzes auf sein Potential zur Laufzeitverkürzung von Tests und die Umsetzung dieses Ansatzes in der Erstellung von Distributed JUnit, einem auf JUnit basierendem Drop-In-Replacement des bekannten Regressionstest-Framework. Die dabei auftretenden Anforderungen und Einschränkungen sollten bewertet und eine Performance-Evaluation durchgeführt werden.

Es wurden zunächst die Selektion, die Priorisierung und schließlich die Distribution als Ansätze zur Laufzeitverkürzung von Tests vorgestellt. Der Distributionsansatz sieht vor, eine Laufzeitverkürzung durch die parallele Ausführung eines Tests auf mehreren Rechnern zu erreichen. Weiterhin wurden Erwägungen angeführt, die bei der Charakterisierung eines geeigneten Distributionsmechanismus berücksichtigt werden müssen.

Vor der Umsetzung wurde zunächst die Funktionsweise und der interne Aufbau von JUnit analysiert und dann eine Konzeption der Umsetzung des Distributionsmechanismus erstellt. Da ein Test üblicherweise aus vielen einzelnen Testfällen besteht, bietet sich das Replicated-Worker-Pattern als Grundlage für die Umsetzung an. Es wurde dann mit ComputeFarm ein Framework ausgewählt, dass unter der Benutzung dieses Modells einen Compute-Server zur Verfügung stellt. Der Distributionsansatz wurde dann umgesetzt in der Implementation von Distributed JUnit.

Distributed JUnit ist in der Lage, einen Test vollständig automatisiert auf partizipierende Rechner zu verteilen. Die Kontaktaufnahme der dabei beteiligten Rechnern findet ebenfalls völlig ohne Zutun des Testers statt und bedarf keiner Konfiguration.

Schließlich wurde eine Performance-Evaluation von Distributed JUnit durchgeführt. Es konnte gezeigt werden, dass die Implementation des Distributionsansatzes vorallem bei Tests mit langer Laufzeit einen Beschleunigungsfaktor aufweist, der nahe am theoretischen Maximum liegt und die Beschleunigung mit steigendem Ressourceneinsatz gut skaliert.

Die im Moment größten Nachteile von Distributed JUnit liegen klar in der durch die Notwendigkeit der Serialisierbarkeit von Tests eingeschränkten Verwendungsmöglichkeit und im Fehlen eines Mechanismus zur Verhinderung von Testkontamination. Diese Tatsache verhindert noch die unbedenkliche Nutzung von Distributed JUnit anstelle von JUnit. Dennoch konnte diese Implementation, obwohl sie sich noch im Anfangsstadium ihrer Entwicklung befindet, vor dem Hintergrund der Erwägungen aus Abschnitt 2.3.1 und den Ergebnissen der Performance-Evaluation aus Kapitel 4 zeigen, dass der Distributionsansatz in der Tat vielversprechend ist und eingehender untersucht werden sollte. Die hier zur Verfügung gestellte Implementation bietet hierfür eine gute Grundlage.

Literaturverzeichnis

[1] Artima Developer Community. http://www.artima.com/.

[2] Extreme Programming - A Gentle Introduction. http://www.extremeprogramming.org/.

[3] Jini Specifications and API Archive. http://java.sun.com/products/jini/.

[4] Jini.org - The Community Resource for Jini Technology. http://www.jini.org/.

[5] SourceForge.net. http://www.sourceforge.net/.

[6] Eric Freemen, Susanne Hupfer, Ken Arnold. *JavaSpaces: Principles, Patterns, and Practice*. Addison-Wesley, Inc., Reading, MA, 1999.

[7] Erich Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[8] Martin Fowler. Refactoring Home Page. http://www.refactoring.com/.

[9] G. Rothermel, M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[10] Gregg Rothermel, Roland H. Untch, Chengyun Chu, Mary Jean Harrold. Test Case Prioritization: An Empirical Study. In *Proceedings of the IEEE International Conference on Software Maintenance*, page 1, August 30-September 03 1999.

[11] G. Rothermel, M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the Sixteenth International Conference on Software Engingeering*, pages 201–210. IEEE Computer Society Press, May 1994.

[12] Gregory M. Kapfhammer. Automatically and Transparently Distributing the Execution of Regression Test Suites. In *Proceedings of the 18th International Conference on Testing Computer Software*, May 2001. http://cd.allegheny.edu:8080/gkapfham/17.

[13] Kent Beck. *SmallTalk Best Practice Patterns*. Prentice Hall, 1996.

[14] Kent Beck, Erich Gamma. JUnit - A Cook's Tour. http://junit.sourceforge.net/doc/cookstour/cookstour.htm.

[15] Kent Beck, Erich Gamma. JUnit, Testing Resources for Etxreme Programming. http://www.junit.org/.

[16] Kent Beck, Erich Gamma. Test-Infected: Programmers Love Writing Tests. *Java Report*, 1, 1998. http://junit.sourceforge.net/doc/testinfected/testing.htm.

[17] Laurie Williams, E. Michael Maximilien, Mladen Vouk. Test Driven Development as a Defect-Reduction Practice. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*. IEEE Computer Society Press, 2003.

[18] Brian Marrick. Testing For Programmers, 2000. `http://www.testing.com/writings/half-day-programmer.pdf`, 13.

[19] Sebastian Elbaum, Alexey G Malishevsky, G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, August 2000.

[20] Shine Technologies. Agile Methodologies Survey Results. `http://www.agilealliance.org/articles/articles/AgileSurvey2003.pdf`.

[21] Tom White. ComputeFarm. `http://computefarm.jini.org/`.

A. Test-Quelltexte

A.1. PassTest.java

```
package djunit;

import junit.framework.*;

public class PassTest extends TestCase {

public void testTrue() {
assertTrue(true);
}
}
```

A.2. RandomTest.java

```
package djunit;

import junit.framework.*;

public class RandomTest extends TestCase {

public RandomTest(String test) {
super(test);
}

/**
 * The fixture set up called before every test method.
 */
protected void setUp() throws Exception {
}

/**
 * The fixture clean up called after every test method.
 */
protected void tearDown() throws Exception {
}

public void testRandom() throws Exception {
int rndA = (int) (Math.random() * 10);
```

```
int rndB = 0;
while (rndA != rndB) {

double x = 1.0, y, z;
for (int i= 0; i < 100000; ++i) {
x = Math.sin((x*i%35)*1.13);
y = Math.log(x+10.0);
z = Math.sqrt(x+y);
}

rndB = (int) (Math.random() * 10);
}

assertEquals(rndA, rndB);
}
}
```

A.3. Durchführung eines Tests mit Distributed JUnit

Unter `http://user-btmurphy.jini.org/` findet sich eine Anleitung, wie man ein Jini-System installiert und startet.
Unter `http://computefarm.jini.org/installation.html` findet sich eine kleine Anleitung, wie man einen ComputeFarm-Client startet.

B. Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 13. April 2005

_____

Eingegangen am (Datum/Stempel):_____