Adaptive und nebenläufige Ausführung von JUnit Testfällen

Entwurf und prototypische Implementierung eines Frameworks

Studienarbeit

Vorgelegt am: 31. Mai 2005

Von: Sven Pecher

Matrikelnummer: 16414

Betreuer (TUHH): Prof. Dr. Ralf Möller

Betreuer (IBM): Ralf Dürig

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne unzulässige fremde Hilfe angefertigt habe. Alle verwendeten Quellen und Hilfsmittel sind angegeben.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Hamburg, den 31.Mai 2005
Sven Pecher

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Übersicht

Diese Studienarbeit beschreibt den Entwurf und die prototypische Implementierung eines Frameworks, das den flexiblen Einsatz von JUnit Testfällen in den entscheidenden Phasen der Softwareentwicklung ermöglicht.

Die Arbeit ist ein fünf Kapitel aufgeteilt:

Kapitel 1 führt in das Thema dieser Arbeit ein, gibt die Vorgehensweise an und formuliert die Anforderungen an das Framework.

Kapitel 2 stellt die Architektur des JUnit Testframeworks vor, erläutert die von JUnit vorgesehenen Erweiterungspunkte und analysiert die Lücken zwischen den Anforderungen und der von JUnit mitgebrachten Funktionalität.

Kapitel 3 gibt einen Einblick in das Java Threading Framework. Dazu werden sowohl die Thread Klasse als auch die Java Threadpools eingehend untersucht.

Kapitel 4 beschreibt die Kernaspekte des Frameworkentwurfs. Von besonderer Bedeutung ist die Umsetzung der vorgesehenen Erweiterungspunkte: Einbindung JUnit Testfälle, Scheduling und Testergebnisverarbeitung, sowie der Entwurf eines flexiblen Konfigurationskonzeptes.

Kapitel 5 stellt eine Beispielimplementierung der drei Erweiterungspunkte vor und erläutert ein in sich geschlossenes Einsatzszenario. Abschließend wird eine Bewertung vorgenommen und ein Ausblick auf mögliche Erweiterungen gegeben.

1 Einleitung

Dieses Kapitel führt in das Thema dieser Arbeit ein, gibt die Vorgehensweise an und formuliert die Anforderungen an das Framework.

1.1 Motivation

Die Erstellung und Ausführung von Tests nimmt im gesamten Lebenszyklus eines Softwareprojekts eine wichtige Rolle ein. Abbildung 1 zeigt zentrale Testdomänen auf, die in vielen Softwareentwicklungsprozessen definiert sind. Modultests (Unit Tests) werden vom Entwickler erstellt und überprüfen das isolierte Verhalten einer Programmeinheit (Unit), im Falle von Java ist dies in der Regel eine Klasse. Interaktionstests testen das korrekte Zusammenspiel mehrerer, bereits getesteter Units und fallen ebenfalls in den Aufgabenbereich des Entwicklers. Kommt Java zum Einsatz, so wird für Modul- und Integrationstests in vielen Fällen das JUnit Framework[1] verwendet.

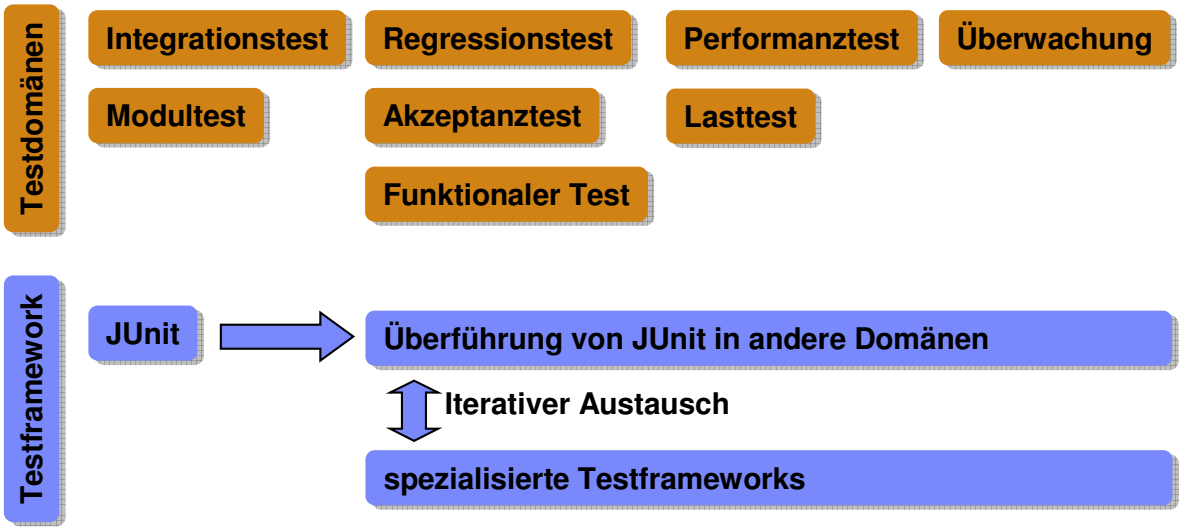

Abbildung 1 - Übersicht Testdomänen

Die funktionalen Anforderungen werden durch ein dediziertes Testteam im Rahmen von funktionalen Tests und Akzeptanzstests überprüft, als Testframework könnte hier z.B.

[1] Vgl. [JUnit]

der Rational Functional Tester[2] Verwendung finden. Durch Performanz- und Lasttests werden nicht funktionale Anforderungen, wie z.B. Antwortzeiten überprüft. Für diesen Bereich gibt es spezialisierte Werkzeuge wie z.B. JMeter[3] oder den Rational Performance Tester[4]. In der Regel wird ein großes Softwaresystem zur Laufzeit durch Monitoringtools, wie z.B. Nagios[5], überwacht. Solche Werkzeuge überprüfen gleichsam funktionale und nicht funktionale Anforderungen. Den in diesem Absatz genannten Programmen ist der große Funktionsumfang in ihrer jeweiligen Domäne gemein. Allerdings stellen sich für den praktischen Einsatz oft mehrere der folgenden Schwierigkeiten ein:

- Es entstehen unter Umständen hohe Lizenzkosten (Ausnahme JMeter).
- Es sind spezielle Kenntnisse für den Einsatz notwendig.
- Häufig sind aufwendige und zeitraubende Installationen notwendig.

Modultests mit JUnit bringen diese Risiken nicht mit sich, es entstehen keine Lizenzkosten, die Entwickler beherrschen das Tool und eine Installation entfällt. Auf den ersten Blick scheint auch eine Überführung von vorhandenen JUnit Modultestdefinitionen in anderen Testdomänen nicht ausgeschlossen, dies wäre sogar sehr effizient. Könnte also JUnit nicht auch in den anderen Testbereichen eingesetzt werden?

Die in Tabelle 1 dargestellten Hauptunterschiede zwischen JUnit Modultests und Testfällen aus anderen Domänen lassen nicht erwarten, dass JUnit von sich aus über entsprechende Fähigkeiten verfügt. Die offene Architektur könnte aber eine gute Basis für Erweiterungen bereitstellen. Es ist sicher nicht realistisch, durch den Einsatz von JUnit alle anderen Tools ersetzen zu können, ein gewinnbringendes Seite an Seite ist aber gutvorstellbar.

Modultest Bereich	**Andere Bereiche**

[2] Der Rational Functional Tester ist ein mächtiges Tool zur Erstellung und Durchführung funktionaler Testfälle (vgl. [RAFUNC]).

[3] JMeter führt Performanztests auf statischen und dynamischen Ressourcen, wie z.B. Servlets, Datenbanken oder Java Objekten, durch (vgl. [JMETER]).

[4] Der Rational Performance Tester ist ein Programmpaket zum Last- und Performanztesten von Web Applikationen (vgl. [RAPERF]).

[5] Nagios ist ein Netzwerküberwachungstool, das zahlreiche Netzdienste überwachen kann. Durch eine Schnittstelle zum Einbinden von Skripten kann nahezu jede Funktionalität kontrolliert werden(vgl. [NAGIOS05]).

• Stets serielle, vollständige Ausführung aller Testfälle. • Die einzelnen Testfälle sind kurz und isoliert. • Die Testergebnisse sind reproduzierbar.	• Keine fixe Reihenfolge und Frequenz in der Ausführung. • Die einzelnen Testfälle sind in der Regel nicht von der Umgebung isoliert. • Durch Abhängigkeiten sind die Testergebnisse nicht zwingend reproduzierbar.

Tabelle 1 – Charakteristika, Modultest Bereich gegenüber den anderen Bereichen

1.2 Ziel dieser Arbeit

Ziel dieser Arbeit ist der Entwurf und die prototypische Umsetzung einer offenen Umgebung, die eine flexible Basis für die Ausführung und Auswertung von JUnit Testfällen auch außerhalb der klassischen Unit Test Domäne bereitstellt. Wie in Abbildung 2 dargestellt soll die Ablaufumgebung auf dem JUnit Framework basieren. Die konkrete Funktionalität wird flexibel durch die Konfiguration der drei Erweiterungspunkte Scheduling, Testfälle und Ergebnisverarbeitung in die Umgebung eingebracht. In den folgenden Abschnitten werden die generellen Kernanforderungen der Ablaufumgebung und der Erweiterungspunkte herausgearbeitet.

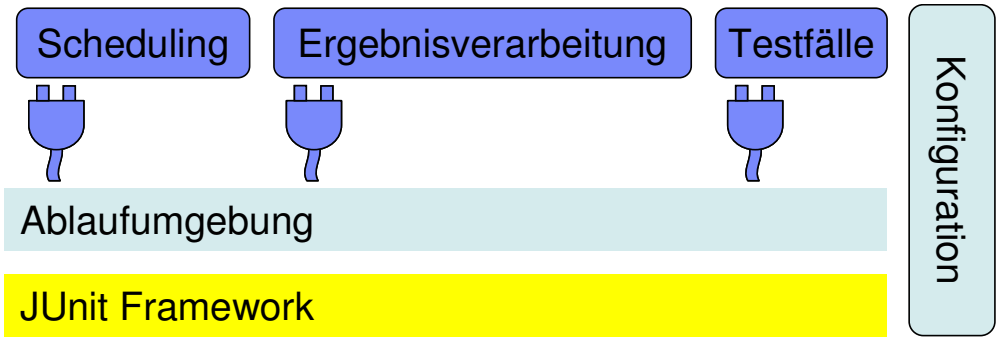

Abbildung 2 - Architekturübersicht

1.2.1 Integration von JUnit Testfällen

Bestehende JUnit Testdefinitionen sollen unverändert mit geringem Aufwand in die Umgebung integriert werden können. Um dies zu erreichen, muss die Granularität existenter Testdefinitionen aufrecht erhalten werden, d.h. neben der Einbindung einzelner Testfälle ist auch die Möglichkeit der Integration gesamter Suiten vorzusehen. Das JUnit Framework sollte ohne Modifikationen eingebunden werden.

1.2.2 Steuerung der Testfälle

Die Umgebung soll ermöglichen den Zeitpunkt, das Intervall und die Reihenfolge der Testausführung flexibel und dynamisch zu bestimmen. Es sind z.B. folgende Strategien denkbar:

- Einmaliger Durchlauf aller Tests.
- Regelmäßig wiederholte Ausführung der Tests (→ Überwachung).
- Dynamische, adaptive Teststeuerung (→ Funktionaler Test).
- Parallele Ausführung der Tests (→ Lasttests).

1.2.3 Verarbeitung der Testresultate

Die Umgebung soll eine Schnittstelle für die flexible Darstellung, Auswertung und Verarbeitung der Testergebnisse bieten, die alle relevanten Daten über den Testablauf bereitstellt. Durch eine Implementierung dieser Schnittstelle sollen folgende Funktionalitäten umgesetzt werden können:

- Benutzer Schnittstelle zur Information über den Testablauf.
- Erstellung von Testberichten in verschiedenen Formaten.
- Auswertung der Testergebnisse und Berechnung von Durchschnittswerten.
- Versenden von Benachrichtigungen und Auslösung von Aktionen.
- Anbindung externer Systeme.

1.2.4 Konfiguration

Alle Charakteristika der Umgebung müssen über eine zentrale Konfiguration bestimmt werden, die zur Laufzeit nicht veränderbar sein muss. Damit Konfigurationsfehler allerdings nicht erst zur Laufzeit bemerkt werden, muss die Konfiguration vor Start des Testablaufes verifiziert werden können. Die einzelnen Ausprägungen der Erweiterungspunkte müssen parametrisierbar sein.

1.2.5 Nicht funktionale Anforderungen

Die Umgebung sollte grundsätzlich für die gleichzeitige parallele Verarbeitung einer vieler Testfälle geeignet sein, genau lässt sich diese Anforderung allerdings nicht quantifizieren. Es sollen sowohl kurz als auch lang laufende Tests zuverlässig durchgeführt werden, entsprechend sind sollten die Schedulingintervalle kurz gehalten werden können.

2 Untersuchung des JUnit Frameworks

Dieses Kapitel stellt die Architektur des JUnit Testframeworks vor, erläutert die von JUnit vorgesehenen Erweiterungspunkte und analysiert die Lücken zwischen den Anforderungen und der von JUnit mitgebrachten Funktionalität.

2.1 Grundsätzliches

JUnit ist ein Framework zur Automatisierung von Java Modultests. Kent Beck und Erich Gamma beschreiben in [BECGAM_B] das zentrale Designziel von JUnit mit *„… the number one goal is to write a framework within which we have some glimmer of hope that developers will actually write tests. The framework has to use familiar tools, so there is little new to learn. It has to require no more work than absolutely necessary to write a new test“*. Entsprechend dieser Vorgabe erfolgt die Testspezifikation ausschließlich in Java, die Ein- und Ausgangsdaten werden implizit im Code definiert. Die Entwickler können somit ihre Testfälle in gewohnter Sprache und Umgebung erstellen.

Anwendungs- und Testcode werden in separaten Klassen gepflegt. Die Granularität in der ein Test spezifiziert, ausgeführt und verifiziert wird, ist der Testfall. Die Ausführung und Verifikation einzelner Testfälle sind voneinander unabhängig.

2.1.1 Erstellung von Testfällen

Wie in [LINFRO03] beschrieben wird ein Testfall durch Java Code repräsentiert, der die Test- und Verifikationsroutinen enthält. Testfälle, die eine ähnliche Struktur besitzen, können eine gemeinsame Fixture nutzen. Eine solche Fixture ermöglicht es, mehreren Testfällen, sich eine gemeinsame Umgebung zu teilen, z.B. könnte eine von allen Testfällen benötigte Objektstruktur zentral aufgebaut werden. Damit nicht jeder Testfall einzeln ausgeführt werden muss, können Testfälle zu Suiten zusammengefasst werden. Eine Suite kann sowohl Testfälle als auch weitere Suiten enthalten.

Die Verifikation einzelner Testbedingungen wird durch diverse Assert Methoden unterstützt, diese erlauben komfortable Überprüfungen auf vielen gängigen Java Datenty-

pen[6]. Die Definition einer Suite, einer Fixture oder eines Testfalles erfolgt rein programmatisch, somit sind in der Regel sind keine Konfigurationsdateien anzupassen.

2.1.2 Ausführung der Testfälle

Ein Testfall oder eine Suite kann durch einen direkten Aufruf[7] oder mittels eines Testrunners ausgeführt werden. Die Testrunner führen die einzelnen Testfälle seriell aus. Falls innerhalb eines Testfalles eine Bedingung nicht erfüllt ist oder ein unerwarteter Fehler auftritt, wird die Ausführung des Testfalles abgebrochen und bei den folgenden Testfällen fortgesetzt. Nach Abarbeitung aller Testfälle werden die Testresultate angezeigt, der Swing basierte Testrunner stellt z.B. folgende Informationen bereit:

- Erfolg / Misserfolg (roter oder grüner Balken).
- Anzahl erfolgreich durchgeführter Testfälle.
- Anzahl nicht erfüllter Bedingungen und unerwarteter Fehler.
- Hierarchische Übersicht über die ausgeführten Testfälle.
- Detaillierte Information über Testfehler (Stack Trace).
- Testlaufzeit.

Ziel, der im JUnit Framework enthaltenen Testrunner, ist es, die Testresultate möglichst kompakt zu präsentieren und die wiederholte Ausführung der Testfälle zu unterstützen. Dies manifestiert sich z.B. in dem roten oder grünen Indikatorbalken, aber auch in der Möglichkeit, die Testklassen ohne Neustart des Testrunners vor jedem Testlauf erneut zu laden.

Die Integration von JUnit in diverse Entwicklungsumgebungen ist weit vorangeschritten, so verfügt z.B. Eclipse über einen eigenen Testrunner, der sich sehr gut in den bestehenden Workflow einbettet. Ein Wizard, der die Erstellung neuer Testfälle unterstützt, steht zur Verfügung. Ebenso ist JUnit gut in das Java Build Werkzeug Ant[8] integriert, ein spezieller Task erlaubt die Einbindung in entsprechende Skripte.

---

[6] Beispielsweise int, short , long, double, float, String, byte, char oder boolean.

[7] Jeder JUnit Test verfügt über eine run() Methode, die den Test durchführt und ein Resultat zurückliefert.

[8] Vgl. [ANT05].

2.2 Architektur des JUnit Frameworks

Der Aufbau des JUnit Frameworks ist stark durch die Verwendung von Design Patterns geprägt, aus diesem Grund wird für das Verständnis des folgenden Abschnitts die Kenntnis der verwendeten Pattern[9] vorausgesetzt.

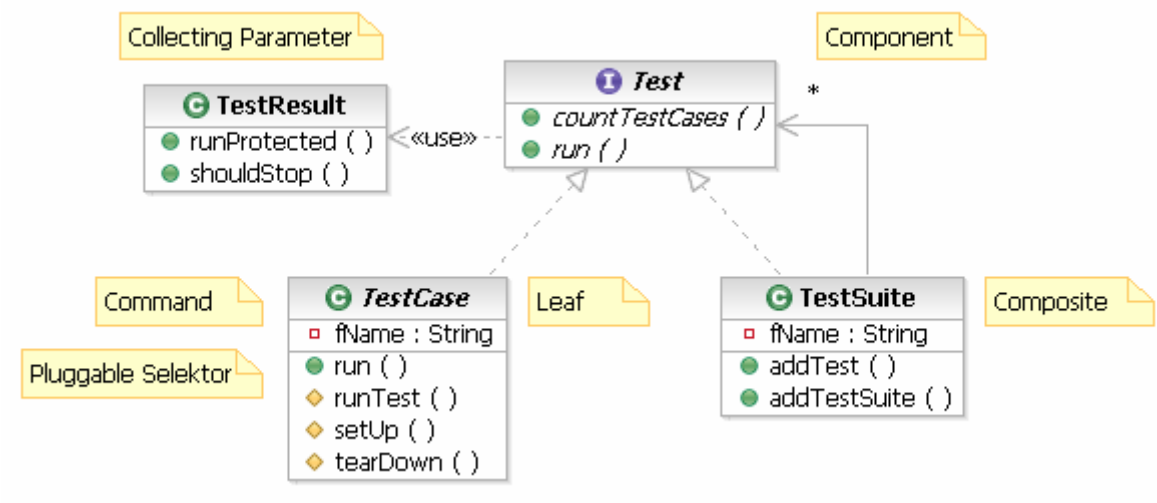

Abbildung 3 - Klassendiagramm JUnit

2.2.1 Aufbau eines Testfalles

Wie in [BECGAM_A] geschildert, wird jeder Testfall zur Laufzeit durch die Instanz einer Subklasse von `TestCase` repräsentiert. Angelehnt an das *Command Pattern* ist in jeder Testklasse eine Methode zur Ausführung des Tests enthalten. Die Methoden zum Auf- und Abbau einer Fixture werden entsprechend des *Template Method Patterns* vorgegeben.

Damit mehrere Testfälle die gleiche Fixture nutzen können, könnte jeder Testfall als Subklasse einer `TestCase` Instanz, die die Auf- und Abbau Methoden enthält, implementiert werden. Daraus würden jedoch sehr viele Klassen resultieren, die viel unnötigen Code zur Klassendefinition erfordern. Aus diesem Grund erlaubt JUnit, entsprechend dem *Pluggable Selector Pattern,* die Testfälle als Methoden einer einzigen Subklasse von `TestCase` zu implementieren. Die Auswahl des Testfalles erfolgt durch einen entsprechenden Parameter im Konstruktor. Erst zur Laufzeit wird dann die ausgewählte Methode mittels Reflektion ausgeführt.

---

[9] Vgl. [GHJV94].

2.2.2 Überprüfung von Testbedingungen

Alle Asserts sind öffentliche statische Methoden der Assert Klasse. Damit in den Testfällen komfortabel auf diese Methoden zugegriffen werden kann, sind alle Testfälle als eine Subklasse von Assert implementiert. Falls eine überprüfte Bedingung nicht erfüllt ist, wird von der jeweiligen Assert-Methode ein spezieller Fehler geworfen, der neben dem Stack Trace noch eine individuelle Meldung enthalten kann, die bei Aufruf der Prüfmethode übergeben wird.

2.2.3 Sammlung der Resultate

Wie oben erwähnt interpretiert JUnit einen Test als erfolgreich, wenn die Ausführung des Testfalles keine Fehler geworfen hat. Die Testergebnisse werden entsprechend dem *Collecting Parameter Pattern* in einer Instanz der Klasse `TestResult` zusammengeführt.

Hierbei wird zwischen den von den Assert Methoden geworfenen `AssertionFailedErrors` und allgemeinen Fehlern unterschieden. Erstgenannte werden als *Failure*, letztgenannte als *Error* am `TestResult` registriert. Das `TestResult` wird in der Regel der Methode zur Ausführung des Testfalles übergeben. An einem `TestResult` können `TestListener` registriert werden, die über den Start und das Ende einer Testmethode, aber auch über das Auftreten eines *Errors* oder *Failures* informiert werden.

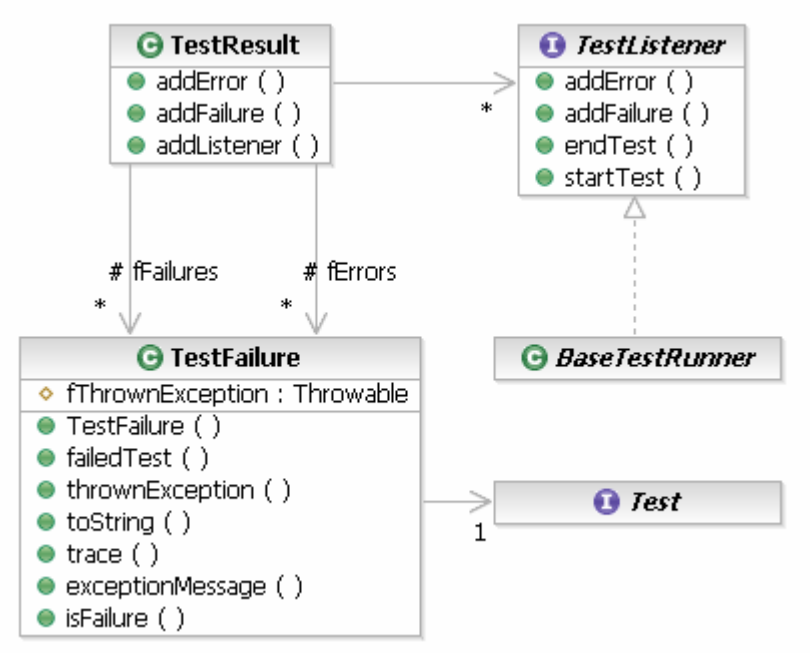

Abbildung 4 - Klassendiagramm TestResult

2.2.4 Aufbau von Suiten

Eine JUnit `TestSuite` implementiert, wie ein Testfall, entsprechend dem *Composite Pattern* das `Test` Interface. Einer `TestSuite` können sowohl einzelne Testfälle, als auch andere TestSuiten hinzugefügt werden. Die Konfiguration der TestSuiten erfolgt, wie auch bei den Testfällen, rein programmatisch. Ebenso wie die Testfälle werden auch die TestSuiten erst zur Laufzeit aufgelöst.

2.2.5 Aufbau der TestRunner

Zur Testausführung instanziert der *TestRunner* zuerst ein neues `TestResult` und registriert sich bei diesem als `TestListener`. Im Anschluss wird für jede in einem Testfall enthaltene Testmethode eine neue Instanz der jeweiligen `TestCase` Klasse erstellt. Im Konstruktor wird der Name der Methode übergeben. Die eigentliche Testausführung wird durch den Aufruf der *run()* Methode auf dem `TestCase`, mit dem TestResult als Argument gestartet. Die *TestRunner* implementieren das `TestListener` Interface und registrieren sich als Listener am `TestResult` um den Testfortschritt zeitnah zu visualisieren.

2.2.6 TestDecoratoren

Unter Nutzung der TestDecorator Klasse können beliebige JUnit Tests gemäß dem Decorator Pattern erweitert werden. Somit ist es möglich, eine einmal entwickelte zusätzliche Funktionalität auf mehr als einen Test anzuwenden, oder auch einen Test mit mehreren Erweiterungen zu dekorieren. Wie in Abbildung 5 dargestellt, erweitert ein konkreter `Decorator` die JUnit Klasse `TestDecorator`. Der konkrete `Decorator` überschreibt die `run()` Methode und implementiert innerhalb dieser die zusätzliche Funktionalität und delegiert durch die `basicRun()` Methode an die `run()` Methode des gekapselten Tests.

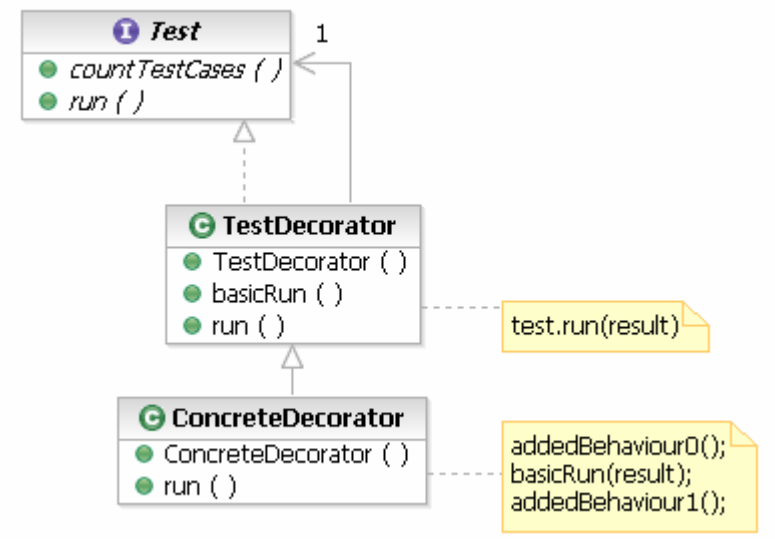

Abbildung 5 - JUnit Decorator Pattern

JUnit bringt schon einige `Decoratoren` mit, z.B. den `RepeatedTest`, der es ermöglicht, einen Testfall mehrfach ausführen zu lassen. Die JUnit Erweiterung JUnitPerf[10] nutzt den `TestDecorator TimedTest`, um beliebigen JUnit Tests einen Timeout hinzuzufügen. Die `run()` Methode des eigentlichen Tests wird dabei in einem eigenen Thread ausgeführt, der gegebenenfalls nach Ablauf des Timeouts abgebrochen wird. In diesem Fall wird dem `TestResult` ein neuer AssertionFailedError hinzugefügt und die `endTest()` Methode aufgerufen. JUnitPerf ermöglicht es auch, mit ähnlicher Vorgehensweise mittels des `LoadTest` Decorators einen Testfall in mehreren Instanzen parallel Ablaufen zu lassen.

[10] Vgl. [CLARK05].

2.3 Analyse der Lücken

2.3.1 Granularität der Testdefinitionen

In allen Test-First Entwicklungsprozessen ist die einfache, schnelle und inkrementelle Erstellung, sowie häufige Ausführung von Testfällen eine elementare Anforderung. Das Design der Testsuiten wird genau diesem Anspruch gerecht. Im Kontext der universellen Ablaufumgebung scheint dieser Ansatz allerdings nicht optimal zu sein. Durch die Testrunner kann gleichzeitig nur ein Testfall oder eine Suite ausgeführt werden. Dies ist im Test-First Konzept durchaus sinnvoll, da alle Tests in einer Suite zusammengefasst werden können. Im Rahmen dieser Arbeit haben die Suiten in der Regel aber keine komplexe Struktur. Der Fokus liegt eher auf der wiederholten, parallelen Ausführung von mehreren kleinen und flachen Test-Suiten, denen jeweils Konfigurationsparameter wie z.B. Timeouts zugeordnet werden müssen. Neben der Ausführung von Suiten als gesamte Einheit, ist es somit auch notwendig, gezielt einzelne parametrisierte Testdefinitionen einer Suite auszuführen.

Die programmatische Konfiguration ermöglicht zwar eine schnelle Erstellung der Testfälle, ist aber nicht sehr flexibel und erfordert auch schon bei leichten Veränderungen eine Neukompilierung. Somit scheint es notwendig, ein zentrales Konzept zur Parameterübergabe an die Testfälle zu entwickeln, das bei Änderung der Parameter keine Neukompilierung erfordert. Da die Evaluation der Testfallkonfiguration jetzt nicht mehr durch die Entwicklungsumgebung oder den Compiler erfolgen kann, ist ein zentrales Konzept zur Evaluation der Testkonfiguration zu entwickeln.

2.3.2 Testausführung

Die TestRunner führen die Testfälle rein seriell aus. Falls ein Testfall sehr lange läuft oder gar nicht terminiert, wird der Testablauf stark verzögert oder aber komplett unterbrochen. Um eine adaptive und nebenläufige Strategie etablieren zu können, muss eine Steuerung vorgesehen werden, durch die die Ausführung eines Testfalles nach einer bestimmten Zeit abgebrochen werden kann. Die JUnit Testrunner erlauben es nicht, den Ausführungszeitpunkt bestimmter Testfälle gezielt zu bestimmen. Es ist also ein Konzept zum individuellen und austauschbaren Scheduling der Testfälle zu entwickeln.

2.3.3 Auswertung

Die vorhandenen Ausgabemöglichkeiten der TestRunner beschränken sich auf die textuelle und grafische Ausgabe der Testergebnisse. Ein flexibles Konzept zur Verarbeitung der Resultate ist nicht gegeben. Es ist also auch hier notwendig ein neues Konzept zu entwickeln.

3 Threads in Java

Dieses Kapitel gibt einen Einblick in das Java Threading Framework. Dazu werden sowohl die Thread Klasse als auch die Java Threadpools eingehend untersucht.

3.1 Threads

Wie in [OAWO04], [HYDE99] und [WELL04] erläutert, werden Threads in Java durch Instanzen der Klasse `java.lang.Thread` repräsentiert. Der Heap der Java Virtual Machine (JVM) steht allen in dieser VM laufenden Threads zur Verfügung. Das Thread Objekt stellt mehrere wichtige statische Methoden bereit, so gibt z.B. die `currentThread()` Methode den aktuellen Thread, in dem der aktuelle Code ausgeführt wird, zurück. Durch die sleep() Methode wird der Thread angewiesen, für eine bestimmte Zeit zu pausieren. Ein Thread kann durch erzeugen einer Subklasse von Thread oder durch Übergabe eines Runnable Objekts an den Thread Konstruktor erzeugt werden.

3.1.1 Lebenszyklus eines Threads

Ein Thread Objekt kann nach seiner Instanzierung mit anderen Objekten interagieren, allerdings führt der an das Objekt gekoppelte Thread selbst noch keinen Code aus und befindet sich im *Initial* Zustand. Erst durch den Aufruf der start() Methode beginnt der Thread die Ausführung. Der Thread terminiert durch beenden der `run()` Methode.

3.1.2 Stoppen und unterbrechen von Threads

Wie eben erwähnt terminiert ein Thread nachdem das letzte Statement der `run()` Methode beendet wurde. Soll ein Thread abgebrochen werden, so scheint die `stop()` Methode der Thread Klasse auf den ersten Blick ein gute Wahl. Bei genauerer Betrachtung ergeben sich allerdings erhebliche Einschränkungen. Durch den Aufruf der stop() Methode wird auf dem Zielthread asynchron die unchecked[11] ThreadDeath Exception geworfen. Die Probleme dieser Vorgehensweise werden in [DEPREC] wie folgt beschrieben „*Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the* ThreadDeath *exception propagates up the stack.) If any of the objects previously*

[11] Java unterscheidet zwischen checked und unchecked Exceptions.

protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be damaged. When threads operate on damaged objects, arbitrary behavior can result. This behavior may be subtle and difficult to detect, or it may be pronounced. Unlike other unchecked exceptions, ThreadDeath *kills threads silently; thus, the user has no warning that his program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future".*

Hält der Thread keine Locks auf anderen Objekten, wie dies z.B. bei Gebrauch von synchronisierten Methoden oder Blöcken der Fall ist, könnte durchaus mit der `stop()` Methode gearbeitet werden, diese Forderung ist allerdings zu hart. Natürlich könnten entsprechend kritische Bereiche auch durch das Fangen der `ThreadDeath` Exception abgesichert werden, allerdings würde dies sehr viel Aufwand erfordern. Es müsste z.B. auch während des Aufräumvorgangs mit einer zweiten `ThreadDeath` Exception gerechnet werden.

Somit ergibt sich als einzige Möglichkeit zur Terminierung eines Thread, die `run()` Methode sauber zu beenden. Hierzu bietet sich die Nutzung der `Thread.interrupt()` Methode an, die folgende Effekte hat: Zum Einen werden alle Methoden, die eine `InterruptedException` werfen, wie z.B. join() oder sleep(), abgebrochen. Zum Anderen wird das interrupted Flag gesetzt, dass bei Bedarf durch `Thread.interrupted()` abgefragt werden kann.

3.1.3 Scheduling der Threads

Die Java Virtual Machine (JVM) verfolgt eine prioritätenbasierte Schedulingstrategie, jeder Thread hat eine vom Entwickler festgelegte Priorität[12], die von der Laufzeitumgebung nicht verändert wird. Der Scheduler ist präemtiv, d.h. ein laufender Thread wird stets von einem Thread mit höherer Priorität unterbrochen. Ein Thread hat einen der folgenden Zustände:

- **Initial:** Das Thread Objekt wurde erzeugt, allerdings wurde die `start()` Methode noch nicht aufgerufen.
- **Runnable:** Die `start()` Methode wurde aufgerufen und der Thread wird oder könnte ausgeführt werden.

[12] Java definiert elf unterschiedliche Prioritäten.

- **Blocked:** Der `Thread` ist blockiert und wartet auf ein Ereignis. Dies tritt z.B. bei einer der folgenden Situationen ein:
  - Der `Thread` versucht noch nicht verfügbare Daten von einem Socket zu lesen.
  - Der Thread befindet sich in einer der blockierenden `sleep()`, `wait()` oder `join()` Methoden.
  - Der `Thread` wartet auf ein Lock, das von einem anderen Thread gehalten wird.
- **Exiting:** Nachdem der Thread seine `run()` Methode beendet hat, befindet er sich in diesem Zustand.

In Abhängigkeit von der Plattform implementiert die JVM die Threads als User- oder System-Level Threads. Falls das Betriebssystem über einen geeigneten Scheduler verfügt, kommen meist System-Level Threads zum Einsatz[13]. In diesem Fall werden die Java Threads durch native Threads repräsentiert und das Scheduling wird an das Betriebssystem delegiert. Die meisten Betriebssysteme nutzen innerhalb einer Priorität ein Round-Robin Scheduling, d.h. Threads gleicher Priorität bekommen abwechselnd jeweils kurze Blöcke CPU-Zeit zugeteilt. Falls die Zielplattform über kein geeignetes Scheduling verfügt, muss die JVM eine eigene Schedulingstrategie implementieren.

3.2 Thread Pools

Ein Thread Pool übernimmt die Ausführung bestimmter Aufgaben, so genannter Tasks. Diese Tasks werden zuerst in eine Warteschlange eingereiht und dann durch einen, der im Pool befindlichen Threads, ausgeführt. Das Verhalten des Pools hängt vor allem von dessen Größe und dem Typ der verwendeten Warteschlange ab. Der Pool hält in der Regel stets eine gewisse Anzahl von Threads vor, hat ein Thread einen Task abgearbeitet, so wird er in den Pool zurückgegeben. Meist wird der Einsatz eines Thread Pools durch einen der folgenden Gründe motiviert:

- **Vermeidung des Overheads der Thread Erzeugung:** Die Erzeugung eines neuen Threads ist teuer, durch ein Pooling kann die wiederholte Erzeugung von Threads vermieden werden.

[13] Dies ist z.B. unter Solaris, Windows und seit Java 1.4.2 auch unter Linux der Fall.

- **Vereinfachung des Programm Designs:** Falls mehrere verschiedene Tasks ausgeführt werden sollen, ist in der Regel ein an vielen Stellen redundanter und oft nicht trivialer Code zur Threadverwaltung notwendig. Ein Pool kann hier viel Vereinfachung und Sicherheit bringen.
- **Performance Verbesserung:** Durch Begrenzung der Anzahl gleichzeitig laufender Threads kann der Durchsatz erheblich gesteigert werden.

3.2.1 Thread Pool Implementierung in java.util.concurrent

Das in der J2SE 5.0 enthaltenen `java.util.concurrent` Paket bringt mit der `ThreadPoolExecutor` Klasse eine leistungsfähige und ausgereifte Thread Pool Implementierung[14] mit. Dem `ThreadPoolExecutor` können beliebige Runnable Objekte zur Ausführung übergeben werden. Die Poolgröße kann dynamisch angepasst werden, die Warteschlange wird bei Erzeugung des Pools festgelegt. Folgende Warteschlagen sind vordefiniert:

- **SynchronusQueue:** Falls der Pool keine freien Threads hat, werden neue Tasks umgehend abgewiesen.
- **LinkedBlockingQueue:** Die Kapazität dieser Warteschlange kann entweder begrenzt oder unbegrenzt sein. Falls die Kapazität unbegrenzt ist, werden beliebig viele Tasks in der Warteschlange platziert, anderenfalls werden bei Erreichen des Kapzitätslimits neue Tasks abgewiesen.

Das Verhalten bei der Abweisung von neuen Tasks kann durch einen entsprechenden Handler festgelegt werden, folgende Vorgehensweisen sind vordefiniert:

- **AbortPolicy:** Falls die Warteschlange voll ist oder der Pool beendet wurde, wird der Task nicht ausgeführt und es wird ein Fehler geworfen.
- **CallerRunsPolicy:** Falls die Warteschlange voll ist, wird der Task direkt im aktuellen Thread ausgeführt. Falls der Pool geschlossen wurde, wird der Task nicht ausgeführt.
- **DiscardPolicy:** Falls keine direkte Ausführung möglich ist, wird die Task nicht ausgeführt.

---

[14] Vgl. [JSR166], Backport für Java 1.4 ist vorhanden.

- **DiscardOldestPolicy:** Falls keine sofortige Ausführung möglich ist, wird der älteste Task aus der Warteschlage entfernt. Falls der Pool bereits geschlossen ist, wird die Aufgabe unterdrückt.

Der Thread Pool hat zwei nicht blockierende Methoden zum geordneten Herunterfahren des Pools. Die `shutdown()` Methode erlaubt die Ausführung aller in der Warteschlange befindlichen Tasks, blockt allerdings neue Tasks. Die `shutdownNow()` Methode versucht einen sofortigen Abbruch der laufenden Aufgaben, d.h. alle, in der Warteschlange befindlichen Aufgaben, werden nicht zur Ausführung gebracht, alle laufenden Threads werden unterbrochen. Die awaitTermination() Methode blockiert bis alle Tasks abgearbeitet sind.

Für die einzelnen Tasks kann kein Zeitintervall vorgegeben werden, bis zu dem der Task gestartet oder seine Ausführung beendet sein muss. Ebenso ist keine direkte Abbruchfunktionalität vorgesehen. Um diesem Dilemma zu entkommen, kann die `FutureTask` Klasse verwendet werden. Sie kann ein beliebiges Runnable um eine Abbruchfunktionalität erweitern. Dazu wird der Thread, in dem die `run()` Methode des `Runnable` ausgeführt wird gespeichert. Die `cancel()` Methode ruft dann zum Abbruch des Tasks die `interrupt()` Methode auf dem gespeicherten Thread auf.

3.3 Task Scheduling

Das java.util.concurret Paket bringt, neben dem schon erwähnten ThreadPoolExecutor, auch einen ScheduledThreadPoolExecutor mit. Dieser basiert auf einem Thread Pool und ermöglicht es Tasks auf verschieden Arten zu schedulen:

- **schedule():** Der Task wird genau zum einem bestimmten Zeitpunkt ausgeführt.
- **scheduleAtFixedRate():** Der Task wird in einem bestimmten Zeitintervall wiederholt ausgeführt.
- **scheduleWithFixedDelay():** Der Task wird wiederholt mit einem bestimmten Delay ausgeführt.

4 Entwurf des Frameworks

Dieses Kapitel beschreibt die Kernaspekte des Frameworkentwurfs. Von besonderer Bedeutung ist die Umsetzung der vorgesehenen Erweiterungspunkte: Einbindung JUnit Testfälle, Scheduling und Testergebnisverarbeitung, sowie der Entwurf eines flexiblen Konfigurationskonzeptes.

4.1 Grundstruktur

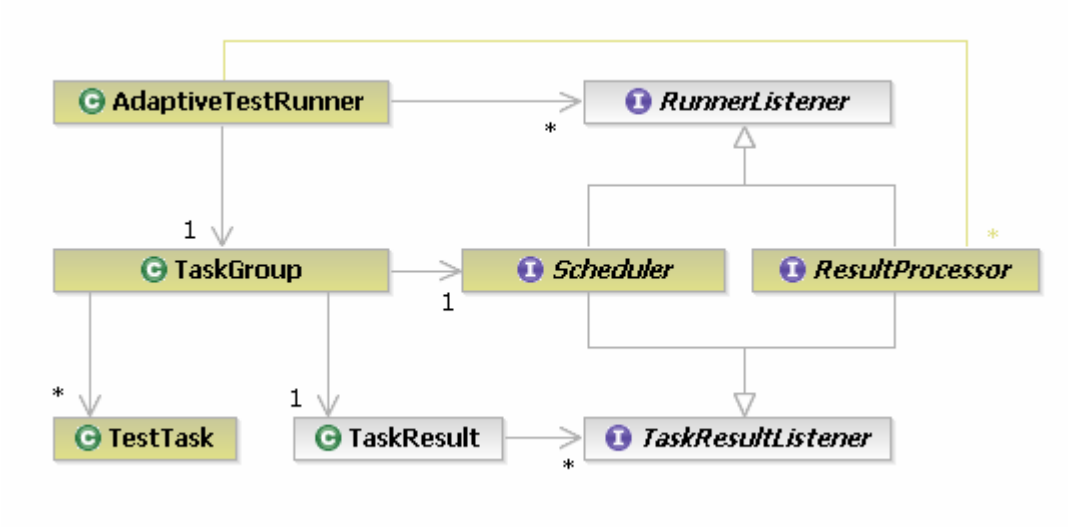

Abbildung 6 - Design Übersicht

Die Grundstruktur des Frameworks ist in Abbildung 6 dargestellt und wird im Folgenden erläutert. Entsprechend der Anforderungen können mit dem `AdaptiveTestRunner` beliebige JUnit Tests ausgeführt werden. Diese werden in einem TestTask gekapselt und in einer TaskGroup gesammelt. Ein Task kann entweder einen einzigen Testfall oder eine gesamte TestSuite repräsentieren.

Die Frequenz und Reihenfolge der Ausführung der Tasks wird durch den an die TaskGroup gebundenen Scheduler bestimmt. Damit der Scheduler adaptiv auf den Testablauf reagieren kann, wird er am TaskResult und am TestRunner als Listener registriert. Für die Verarbeitung der Resultate sind die ResultProcessoren zuständig, die ebenfalls am TaskResult und am TestRunner als Listener registriert werden. Die weiteren Abschnitte erläutern die einzelnen Designentscheidungen im Detail. Die Quelltexte sind auf der CD im Anhang zu finden.

4.2 Bereitstellung der Nebenläufigkeit

Eine zentrale Anforderung des Frameworks ist die nebenläufige Ausführung der Testfälle. Zur Umsetzung dieser Anforderung zeichnen sich aus Kapitel 3 zwei Varianten ab. Zum Einen ist dies eine Thread-basierte Umsetzung, zum Anderen eine auf einem Thread Pool aufsetzende Umsetzung. Die Vor- und Nachteile werden in Tabelle 2 aufgelistet.

	Thread	**ThreadPool**
+	• Lauffähig unter allen[15] JRE's.	• Übersichtliches Programmiermodell, Fokussierung auf der Anwendungslogik. • Thread Pooling und Scheduling wird unterstützt.
-	• Sehr viel Low Level Code zur Thread Verwaltung. • Für jeden Testfall werden zwei Threads erzeugt, ein Thread zur Ausführung und ein Thread zur Timeout Kontrolle. • Unübersichtliche Programmierung.	• Ohne zusätzliche Bibliotheken nur lauffähig unter JRE 5.0. • Es werden nicht alle Anforderungen bezüglich der Timeouts erfüllt.

Tabelle 2 - Gegenüberstellung Thread / Threadpool

Für die Verwendung eines Thread Pools spricht das mächtige und abstrahierende Programmiermodell, des weiteren ergeben sich durch die bereitgestellte Funktionalität interessante Ansätze für Schedulingalgorithmen. Gegen den Einsatz eines Thread Pools sprechen nur die Abhängigkeiten zur Java Laufzeit Umgebung, diese sind im Kontext dieser Arbeit akzeptabel. Der Einsatz eines Pools scheint also höchst sinnvoll. Die vorhanden funktionalen Lücken bei der Timeout Steuerung und der Integration JUnit Tests werden in den folgenden Abschnitten geschlossen. Ziel ist die Bereitstellung eines Pools, der die Nebenläufige Ausführung von JUnit Tests erlaubt und leicht von einem Scheduler genutzt werden kann.

[15] Einschränkungen siehe Kapitel 3.1.

4.2.1 Design eines Pools mit zeitbasiert abbruchfähigen Tasks

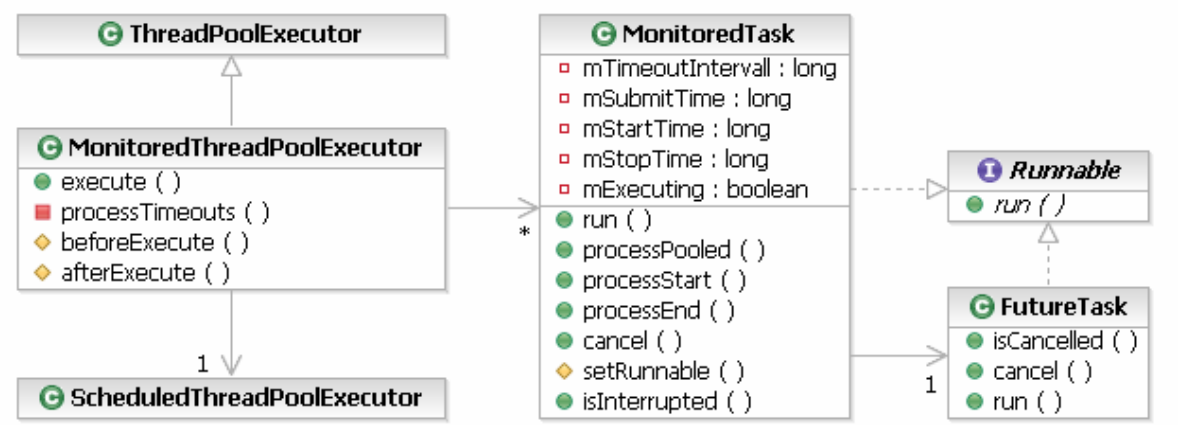

Abbildung 7 - Klassendiagramm Monitored Thread Pool

Entsprechend der Anforderungen sollte es möglich sein, die Ausführung der Testfälle zur Laufzeit zu beeinflussen. Z.B. kann es somit notwendig sein, einzelne lang laufende Testfälle nach einer gewissen Zeit abzubrechen. Wie in Kapitel 3.2 erläutert verfügt sowohl der `ThreadPoolExecutor` als auch die von ihm ausgeführten Tasks[16] über keine zeitbasierte Abbruchfunktionalität. Es wäre denkbar analog des JUnitPerf Ansatzes (siehe Kapitel 2.2.6) die Timeoutsteuerung ausschließlich an die Tasks zu delegieren. Das JUnitPerf Konzept könnte durch die Entwicklung einer neuen Task Klasse, die beliebige Runnables kapselt, auch generalisiert werden. Allerdings wird so für jeden Task eine neuer Thread erzeugt, der nicht unter der Kontrolle des Pools steht. Durch den Overhead für die Verwaltung der zusätzlichen Threads und durch den Kontrollverlust des Pools würden sich große Nachteile ergeben.

Erheblich übersichtlicher und ressourcenschonender ist es, die Laufzeit aller Tasks in regelmäßigen Abständen durch eine Methode aus einem einzigen separaten Thread überprüfen zu lassen. Der in Kapitel 3.3 eingeführte `ScheduledThreadPoolExecutor` stellt, mit der Option einen Task mit einem bestimmten zeitlichen Abstand wiederholt auszuführen, eine für diesen Zweck geeignete Basis bereit. Es wäre lediglich notwendig dem `ScheduledThreadPoolExecutor` eine geeignete, als Runnable gekapselte Kontrollmethode zu übergeben. Damit eine Kontrollmethode Kenntnis über die Laufzeit der einzelnen Tasks erlangen kann, muss zumindest für jeden Task der Zeitpunkt des Eintritts in die

[16] Ein ThreadPoolExecutor kann Runnables, Callables und FutureTasks ausführen.

Warteschlange gespeichert werden, ebenso könnte aber auch der Start- und Endzeitpunkt von Interesse sein. Hier drängt sich die Erstellung eines neuen Taskobjekts auf, dass die entsprechenden Attribute enthält und ein Runnable kapselt.

Die in Abbildung 7 skizzierte und im Folgenden erläuterte Lösung basiert folglich auf einer eigenen Executor Implementierung, die den `ThreadPoolExecutor` erweitert und beliebige `Runnables`, gekapselt durch eine neue Task Klasse, kontrolliert ausführt.

Den Tasks, repräsentiert durch Instanzen der `MonitoredTask` Klasse, kann durch die `setRunnable()` Methode ein Runnable übergeben werden, welches Klassenintern durch einen `FutureTask` gekapselt wird. Dies wird gemacht, um die in Abschnitt 3.2.1 erläuterte Abbruchfunktionalität der `FutureTask` Klasse zu nutzen, die durch die `cancel()` Methode angesprochen wird. Der `MonitoredTask` implementiert selbst das `Runnable` Interface, kann somit direkt durch den Executor ausgeführt werden und steht auch direkt in den beforeExecute() and afterExecute() Methoden zur Verfügung.

Die `MonitoredThreadPoolExecutor` Klasse erbt vom `ThreadPoolExecutor` und ist für die Ausführung der `MonitoredTasks` zuständig, die durch die `execute()` Methode übergeben werden. Die enthaltene processTimeouts() Methode implementiert die Abbruchcontrollerlogik, indem sie über alle wartenden oder in Ausführung befindlichen Tasks iteriert und diese ggf. bei Zeitüberschreitung abbricht. Damit diese Methode regelmäßig ausgeführt wird, wird sie als Runnable verpackt einem `ScheduledThreadsPoolExecutor` zur wiederholten Ausführung übergeben.

Damit die Controllerlogik eine Informationsbasis erhält, werden die Zeitpunkte des Starts, Stopps und Eintritts in die Warteschlange bei Aufruf der process-Methoden gespeichert.

4.2.2 Einbettung von JUnit Tests

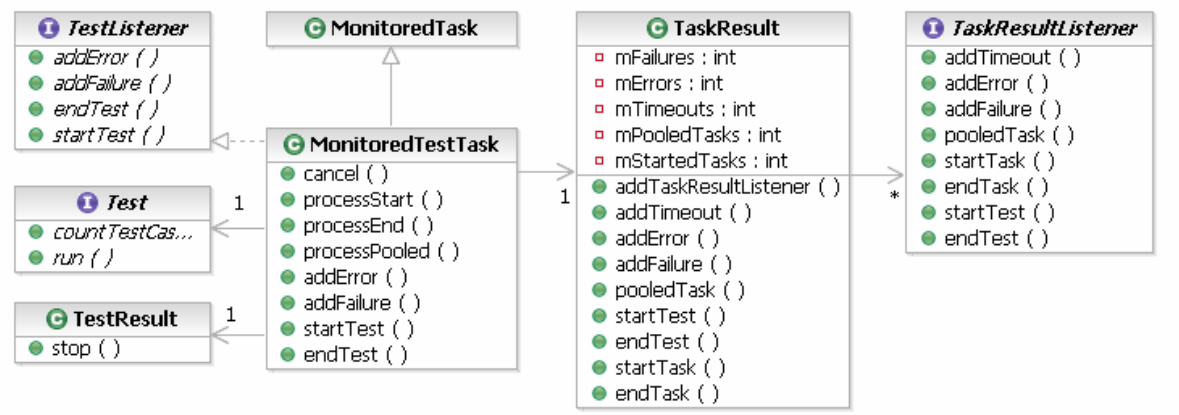

Abbildung 8 - Klassendiagramm Monitored Test Task

JUnit Tests implementieren nicht das `Runnable` Interface und können somit auch nicht direkt durch einen `MonitoredTask` gekapselt werden. Wie in Kapitel 2.2.1 geschildert erwartet die run() Methode des Tests ein `TestResult` als Parameter. Damit JUnit Tests durch den `MonitoredThreadPoolExecutor` ausgeführt werden können, erweitert der `MonitoredTestTask` den `MonitoredTask`. Der JUnit `Test` wird als `Runnable` übergeben, für die Weiterleitung des Ausführungsstandes und der Testresultate wird Sorge getragen.

Der `MonitoredTestTask` bekommt im Konstruktor ein `TaskResult` übergeben, in dieses werden die `TestListener` Methoden `addError()`, `addFailure()`, `startTest()`, `endTest()` übertragen, dabei wird jeweils zusätzlich der `MonitoredTestTask` als Parameter ergänzt. Für jeden MonitoredTestTask wird ein neues TestResult erzeugt.

Die vom `MonitoredThreadPoolExecutor` aufgerufenen `processStart()`, `processEnd()` und `processPooled()` Methoden rufen die entsprechenden `TaskResult` Methoden `startTask()`, `endTask()` und `pooledTask()` im `TaskResult` auf. Die `cancel()` Methode wird überschrieben, um das `TaskResult` über den Abbruch zu informieren und dem `TestResult` durch Aufruf der shouldStop() Methode den Wunsch zum Beenden mitzuteilen. Ansonsten würde bei der Abarbeitung von `TestSuiten` die weitere Ausführung nicht unterbrochen werden (vgl. Kapitel 2.2.3). Um den Abbruch korrekt zu bearbeiten, muss wie in Kapitel 3.1.2 beschrieben, in den Testfällen das `Thread.interrupted()` Flag beachtet werden.

Das `TaskResult` sammelt alle Testergebnisse einer `TaskGroup`, an diesem können `TaskResultListener` registriert werden. Ähnlich wie im `TestResult` werden die Failures, Errors, Timeouts, PooledTasks und StartedTask gezählt. Auf die Sammlung der Failures und Errors wird verzichtet.

4.3 Ausführung der Testfälle durch den Scheduler

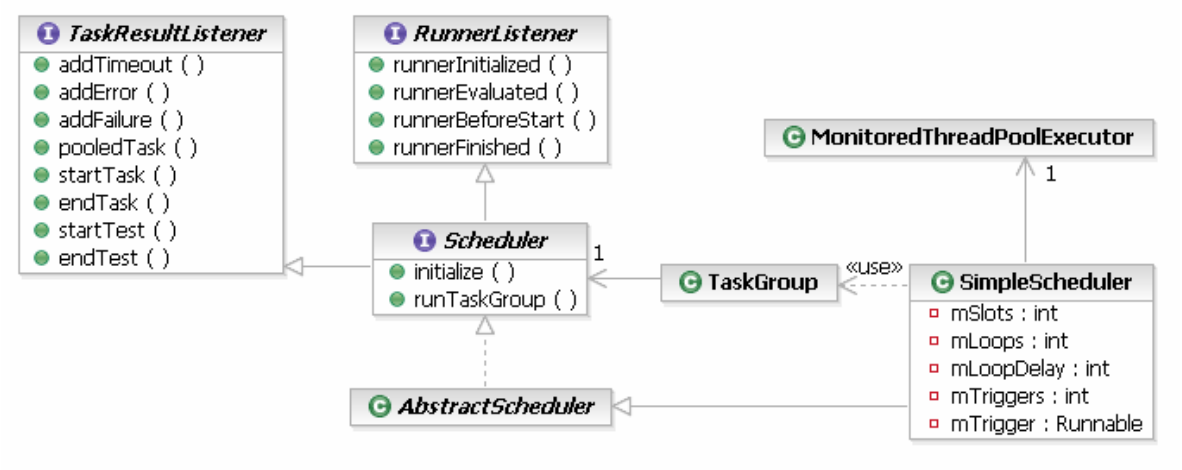

Abbildung 9 - Klassendiagramm Scheduler

Der Scheduler ist die zentrale Komponente des Frameworks, er steuert die nebenläufige und adaptive Ausführung der Testfälle. Entsprechend den Anforderungen sollte sich der Scheduler leicht austauschen lassen, bzw. neue Scheduler sollten einfach zu entwickeln sein. Hier ist zwischen Flexibilität und leichter Erweiterbarkeit abzuwägen.

Ein Scheduler ist am TaskResult und am Runner als Listener registriert und implementiert die entsprechenden Interfaces. Somit erhält der Scheduler alle wichtigen Informationen über den Testablauf. Der Scheduler erhält in der initialize() Methode ein TaskGroup übergeben, durch dem Aufruf der runTaskGroup() Methode wird der Scheduler aufgefordert, die TaskGroup eigenständig auszuführen. Die runTaskGroup() Methode sollte blockieren bis die Ausführung aller vorgesehenen Tasks abgeschlossen ist. Damit nicht jeder Scheduler alle Listenermethoden implementieren muss, wird mit dem AbstractScheduler, der alle entsprechenden Methoden implementiert, eine abstrakte Basisimplementierung.

Die Scheduler können sehr flexibel implementiert werden, da sehr wenige Vorgaben für sie gemacht werden. Eine Verwendung des MonitoredThreadPoolExecutors scheint

allerdings in allen Fällen sehr ratsam. In Kapitel 5 werden zwei Beispielimplementierungen von Schedulern vorgestellt.

4.4 Testergebnisverarbeitung und Konfiguration

Der `ResultProcessor` ist die zentrale Schnittstelle zur Erstellung von Klassen zur Testergebnisverarbeitung. Er erweitert, wie der Scheduler, die `TaskResultListener` und die `RunnerListener` Schnittstelle, die alle Methoden zur Testablaufsbenachrichtigung bereitstellen. An eine `TaskGroup` können beliebig viele `ResultProcessoren` gebunden werden. Eine Beispielimplementierung wird in Kapitel 5 vorgestellt.

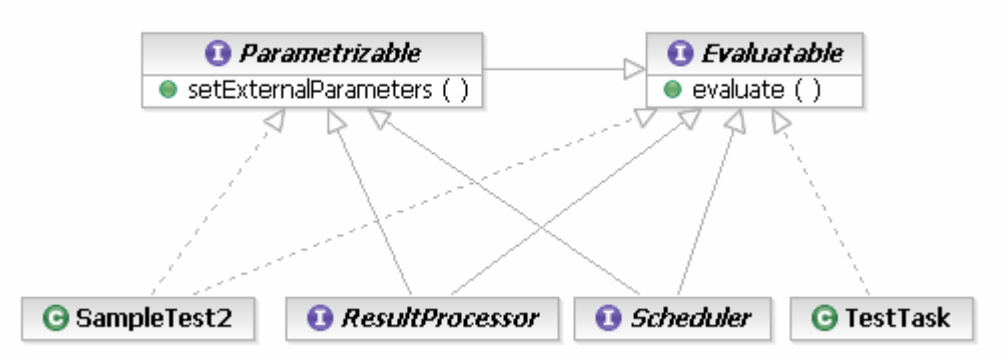

Abbildung 10 - Klassendiagramm Evaluation und Konfiguration

Der Runner wird durch eine zentrale XML Datei konfiguriert, die nach dem Start eingelesen und evaluiert wird. Die Ergebnisse de Evaluation werden entsprechend des *Collecting Parameter Patterns* in einem `EvaluationResult` zusammengeführt. Die zentralen Konfigurationsobjekte sind die `Scheduler`, `Resultprozessoren` und die `TestTasks`. Die `Scheduler` und die `ResultProzessoren` erweitern das `Parametrizable` und das `Evaluatable` Interface. Somit können aus der Konfigurationsdatei Parameter übergeben werden. Des weiteren kann nach dem Einlesen, die Konfiguration überprüft werden. Ein durch einen TestTasks gekapselter JUnit `TestCase` kann ebenfalls das `Parametrizable` und `Evaluatable` Interface implementieren und entsprechend angesprochen werden, für gekapselte `TestSuiten` funktioniert dieser Ansatz allerdings nicht.

5 Einsatzszenarien und Fazit

Dieses Kapitel stellt eine Beispielimplementierung der drei Erweiterungspunkte vor und erläutert ein in sich geschlossenes Einsatzszenario. Abschließend wird eine Bewertung vorgenommen und ein Ausblick auf mögliche Erweiterungen gegeben.

5.1 Beispielhafte Umsetzung der Erweiterungspunkte

In den folgenden Abschnitten wird für jeden, der in der **Abbildung 2** dargestellten Erweiterungspunkte, eine mögliche Implementierung vorgestellt. Die Quelltexte sind auf der CD im Anhang zu finden.

5.1.1 Einfacher Scheduler

In diesem Abschnitt soll ein Scheduler für den `AdaptiveTestRunner` erstellt werden, der die folgenden Testausführungsstrategien unterstützt:

- Einmalige serielle Ausführung der Testfälle.
- Wiederholte serielle Ausführung der Testfälle mit definiertem zeitlichem Abstand.
- Einmalige parallele Ausführung der Testfälle.
- Wiederholte parallele Ausführung der Testfälle mit definiertem zeitlichem Abstand.

Wie in Kapitel 4.3 vorgeschlagen erweitert der `SimpleScheduler` die abstrakte Scheduler Basisimplementierung. Unter Ausnutzung der `Parametrizable` Schnittstelle können folgende Parameter aus der Konfigurationsdatei übernommen werden:

- **loops:** Anzahl der Wiederholungen.
- **loopDelay:** Abstand (Delay) zwischen den einzelnen Wiederholungen.
- **slots:** Anzahl der maximal gleichzeitig ausgeführten Testfälle (Poolgröße).

Eine Evaluation der Parameter ist nicht notwendig, bei Fehlkonfiguration kommen Standardwerte zum Einsatz.

Die Testfälle werden durch einen `MonitoredThreadPoolExecutor` ausgeführt, der mit dem aus der Konfiguration übernommenen Wert für die Poolgröße und einer `LinkedBlockingQueue` initialisiert wird. Sollen die Testfälle ohne Delay ausgeführt werden, so werden die vorgesehenen Testfälle nacheinander in den Pool eingereiht. Die Testfälle werden hierzu aus der `TaskGroup` ausgelesen und einzeln in einem `MonitoredTasks`

gekapselt. Sollen die Testfälle mehrfach ausgeführt werden, so wird dieser Vorgang mehrfach wiederholt.

Sollen die Testfälle hingegen wiederholt mit einem Delay ausgeführt werden, so übernimmt der im `MonitoredThreadPoolExecutor` enthaltene ScheduledThreadPoolExecutor das Triggern der Ausführung. Ihm wird ein Task übergeben, der entsprechend des Delays die Testfälle wiederholt in den Pool einreiht.

5.1.2 Adaptiver Scheduler

In diesem Abschnitt wird ein einfacher adaptiver Scheduler entwickelt. Oft ist die Laufzeit einzelner Testfälle im Voraus unbekannt, es ist lediglich klar, dass ein gewisser maximaler Timeout nicht überschritten werden soll. Ist der Testfall jedoch in der Regel deutlich schneller als der Timeout, so scheint es nicht sinnvoll, bei Ausfällen auf den vollen Timeout zu warten. Somit ist es gut denkbar, dass der Scheduler während der Testausführung die Timeouts adapativ an die realen Laufzeiten der Testfälle anpasst.

Hier ist die Idee den Scheduler die ersten n Iterationen, das Timeout entsprechend des Testausgangs anpassen zu lassen. Läuft der Test in den Timeout, so wird dieser erhöht. Ist der Test erfolgreich, so wird der Timeout verkleinert. Entsprechend erhält der Scheduler folgende Parameter[17]:

- **adaptIterations:** Anzahl der Anpassungsiterationen.
- **minTimeout:** Minimales angenommenes Timeout.
- **maxTimeout:** Maximales angenommenes Timeout.
- **adaptStep:** Veränderung des Timeouts bei Anpassung in Sekunden.

Die Implementierung erweitert den im vorherigen Abschnitt entwickelten SimpleScheduler. Zusätzlich wird die endTest() Methode des TaskResultListeners überschrieben um während der Anpassungsiterationen bei einem Testabbruch das Timeoutintervall des Testfalles zu erhöhen oder bei Erfolg zu verkleinern.

---

[17] Es wäre gut denkbar, diese Parameter auch individuell in den einzelnen Testfällen zu konfigurieren. Dazu müssten die Testfälle das Parametrizable Interface implementieren und die Parameter bereitstellen. Durch eine entsprechende Implementierung der evaluate() Methode könnte im Scheduler sogar die Verfügbarkeit der Parameter sichergestellt werden.

5.1.3 Ein einfaches Frontend

Durch Implementierung des ResultProcessor Interfaces lassen sich die Testresultate auf viele unterschiedliche Arten visualisieren. Es wäre z.B. gut denkbar, in einen `ResultProcessor` einen HTTP-Server[18] zu integrieren, und die Ausgabe oder sogar die Steuerung über ein Browserinterface vorzunehmen. Zu Demonstrationszwecken soll allerdings ein einfaches textbasiertes Userinterface genügen, welches alle Methoden der TaskResultListener und RunnerListener Schnittstellen implementiert und jeweils eine kurze Meldung auf der Konsole ausgibt. Nach Durchlauf aller Testfälle wird eine kurze Zusammenfassung ausgegeben.

5.1.4 Ein einfacher HTTP Testfall

In diesem Abschnitt wird ein einfacher Testfall erstellt, der eine URL von einem HTTP-Server abfragt und die Antwort überprüft. Der Testfall soll als erfolgreich gewertet werden, wenn der Returncode 200 zurückgegeben wird und die Antwort einen bestimmten konfigurierbaren String enthält. Hierzu bietet es sich an, die Testdefinition unter Zuhilfenahme des HTTPUnit Frameworks [19] zu erstellen. Es wird eine Subklasse von `junit.framework.TestCase` erstellt. Diese ruft mittels HTTPUnit eine konfigurierbare URL von einem Server ab. Durch Implementierung der Parametrizable und Evaluatable Schnittstellen werden folgende Parameter übergeben und überprüft:

- **url:** Die URL des Servers.
- **inculdedString:** String, der in der abgerufenen Seite enthalten sein muss.

5.1.5 Beispielkonfigurationen

Auf Basis, der in diesem Abschnitt erstellten Umsetzungen wurden die in Tabelle 3 gelisteten Testkonfigurationen erstellt[20]. In allen Konfigurationen kommt der `SimpleScheduler` mit dem einfachen Frontend zum Einsatz.

---

[18] Wie z.B. den schlanken Servletcontainer Jetty (siehe [JETTY]).

[19] Mit HTTPUnit lässt sich in Java sehr leicht ein Browser simulieren (siehe [HTTPUNIT])

[20] Quelltexte sind auf der CD zu finden.

Beschreibung	Konfigurationsdatei	Testfälle	Konfiguration
Einfache serielle Ausführung der Testfälle	config_simple_serial	testSuccess, testFail, testWhile	Der Scheduler wird mit einem Slot, einer Loop und keinem Delay konfiguriert.
Mehrfach wiederholte parallele Ausführung der Testfälle	config_simple_par	testSuccess, testFail, testWhile	Der Scheduler wird mit vier Slots, zehn Loops und keinem Delay konfiguriert.
Einfache Ausführung einer Suite	config_suite	TestSuite mit testSuccess, testFail, testWhile	Der Scheduler wird mit einem Slot, einer Loop und keinem Delay konfiguriert.
Monitoring einer Webseite	config_http_mon	GenericHTTPTest auf zwei Webseiten	Der Scheduler wird mit einem Slot, einer unendlichen Wiederholung und einem Delay von 20 Sekunden konfiguriert.
Lasttest einer Webseite	config_http_last	GenericHTTPTest auf vier Webseiten	Der Scheduler wird mit zehn Slots, zehn Loops und einem Delay von einer Sekunde konfiguriert.

Tabelle 3 - Beispielkonfigurationen

5.2 Ein einfaches Einsatzszenario

Im Folgenden wird geschildert, wie eine einzige Testfalldefinition aufgrund der adaptiven und nebenläufigen Fähigkeiten des AdaptiveTestRunners in mehreren Testdomänen verwendet werden kann. Zielobjekt der Tests ist ein Servlet, dass in einen Servletcontainer deployt werden soll.

Nachdem der Entwickler das Servlet erstellt und in seiner lokalen Testumgebung installiert hat, erstellt er im Rahmen der **Modultests** einen einfachen HTTPUnit Testfall, der die korrekte Funktionalität des Servlets überprüft. Dieser wird dann aus der Entwicklungsumgebung des Entwicklers ausgeführt.

Im Anschluss wird das Servlet in die Zielumgebung deployt. Im Rahmen eines **Deploymentests** wird dieser Vorgang dann überprüft. Hier könnte im Prinzip der beste-

hende Testfall zum Einsatz kommen, es wird aber notwendig sein, die URL des Servers auszutauschen. Hier bietet es sich - analog zu Abschnitt 5.1.4 - an, eine Parametrisierung vorzunehmen und den Testfall durch den `AdaptiveTestrunner`, bestückt mit einem seriellen Scheduler, ausführen zu lassen.

Als nächstes muss überprüft werden, ob das Servlet seinen nicht funktionalen Anforderungen genügt, die z.B. einen **Lasttest** erfordern könnten. Dieser lässt sich mit dem `AdaptiveTestRunner` sehr leicht umsetzen, es ist lediglich der Scheduler auszutauschen. Der in Abschnitt 5.1.1 entwickelte `SimpleScheduler` scheint hier geeignet zu sein.

Hat das Servlet die bisherigen Tests überstanden, so wird es in den produktiven Betrieb übernommen und muss fortan durch einen **Monitoringprozess** überwacht werden. Auch diese Aufgabe kann leicht durch den `AdaptiveTestRunner` übernommen werden, es muss lediglich der Scheduler entsprechend der Anforderungen konfiguriert werden. Gegebenenfalls ist es sinnvoll auch noch einen passenden `ResultProcessor` zu erstellen, der gefiltert nur die benötigten Informationen ausgibt.

5.3 Fazit

Wie im vorherigen Abschnitt erläutert, erlaubt das entwickelte Framework die adaptive und nebenläufige Ausführung von Testfällen. Da das Konzept der Erweiterungspunkte erfolgreich umgesetzt wurde, lässt sich der Testrunner sehr flexibel konfigurieren und auch erweitern. Sicher kann der `AdaptiveTestRunner` nicht bestehende Testwerkzeuge ersetzen, dazu fehlt momentan einfach zu viel Funktionalität, die in den etablierten Tools bereits enthalten ist. Oft liegt allerdings zwischen Entwicklungsstart und der Bereitstellung entsprechender Testtools eine größere Zeitspanne, die durch das entwickelte Framework gut überbrückt werden kann. In vielen Fällen sind auch keine spezialisierten Testtools notwendig oder mangels Ressourcen nicht verfügbar. In diesen Bereichen kann speziell durch die Tatsache, dass viele vorhandene Testfalldefinitionen weiter genutzt oder modifiziert werden können, der `AdaptiveTestRunner` sicher eine gute Alternative darstellen. Die spätere sukzessive Migration zu spezialisierten Werkzeugen ist jederzeit möglich. Häufig kann auch durch den Einsatz mehrerer Tools ein besserer Blick auf schwierige Probleme erlangt werden.

Als Einschränkung gilt sicherlich, dass der erzeugte Code momentan eher den Status eines Prototyps besitzt. Die vorliegende Implementierung ist als Bestätigung des grundsätzlichen Konzepts zu verstehen. Für einen produktiven Einsatz sollte der Code noch ausführlich getestet werden. In den folgenden Abschnitten wird die Erfüllung der funktionalen und nicht funktionalen Anforderungen bewertet.

5.3.1 Bewertung der Funktionalen Anforderungen

Das JUnit Framework wurde ohne Modifikationen erfolgreich eingebunden. Auf den ersten Blick scheint die Einbindung der JUnit Testfälle gut gelungen, sowohl einzelne Testfälle als auch gesamte Suiten können ausgeführt werden. Allerdings können nur die Testfälle sinnvoll konfiguriert werden. Eine Testsuite wird stets als gesamte Einheit betrachtet. Wie im vorherigen Abschnitt geschildert, ist es sehr leicht, existente Testfälle in andere Testdomänen zu überführen. Leider obliegt, wie in Abschnitt 3.1.2 geschildert, die Timeoutsteuerung bis zu einem gewissen Maße den Testfällen selbst. Zwar wird der Timeout des Testfalles unmittelbar an den `ResultProcessor` propagiert, es liegt aber in der Verantwortung des Testfalles sich selbst zu beenden.

Das Scheduler Konzept erlaubt eine sehr große Flexibilität in der Testausführung, es lassen sich alle in den Anforderungen erwähnten Strategien umsetzten. Die beiden ThreadPools motivieren durchaus zur Umsetzung aufwendiger Schedulingstrategien. Leider bedingt das flexible Konzept einen recht hohen Aufwand, um eine neuen Scheduler zu erstellen. Mitunter wird diese Aufgabe auch sehr komplex, weil zu viele Interna der Pools beachtet werden müssen. Hier könnte es sicher sinnvoll sein, basierend auf der Scheduler Schnittstelle verschiede Basisimplementierungen bereitzustellen.

Der Bereich der Benutzerschnittstelle wurde in dieser Arbeit eher vernachlässigt, zwar werden durch die `ResultProcessor` Schnittstelle alle geforderten Informationen bereitgestellt, das in Abschnitt 5.1.3 entwickelte Frontend genügt aber nur geringen Ansprüchen. Um den produktiven Einsatz zu ermöglichen, müssen noch einige Arbeiten in diesem Bereich erledigt werden. Ähnlich wie bei den Schedulern scheint die Entwicklung von mehreren Basisimplementierungen sinnvoll.

Die zentrale Konfiguration des `AdaptiveTestRunners` erlaubt eine komfortable und übersichtliche Einstellung der Konfigurationsparameter. Die Implementierung der Eva-

luatable und Parametrizable Schnittstelle erfordert zwar etwas Aufwand, wird allerdings durch eine robuste Parametrisierbarkeit der Konfigurationseinheiten belohnt.

5.3.2 Bewertung der nicht funktionalen Anforderungen

Die erfolgreiche Durchführung des in Abschnitt 5.1.5 erstellten HTTP-Lasttests lässt zuerst auf eine Erfüllung der nicht funktionalen Anforderungen hoffen. Letztlich lässt sich dieses allerdings nur schwer als Beweis heranziehen, es kamen keine Werkzeuge zur Erzeugung einer aussagekräftigen Datenbasis zum Einsatz. Es gibt speziell in Hinsicht auf die Performanz der verwendeten Threadpools Unsicherheiten. Gerade durch eine geeignete Einstellung der Poolparameter scheinen sich aber auch gute Tuningmöglichkeiten zu ergeben.

5.4 Ausblick

Es gibt viele Angriffspunkte um den Funktionsumfang, die Zuverlässigkeit und speziell auch die Bedienbarkeit des `AdaptiveTestRunners` zu erweitern. Ein wichtiger Punkt ist die Erstellung weiterer parametrisierter Standardtests (vgl. Abschnitt 5.1.4). Durch die Entwicklung neuer Scheduler kann der Funktionsumfang und die Zuverlässig erheblich erweitert werden. Auch lassen sich durch eine Erweiterung in diesem Bereich leicht neue Testdomänen erschließen.

Das Frontend wurde in dieser Arbeit stark vernachlässigt, durch eine Erweiterung der ResultProcessoren lässt sich die Aussagekraft der Testresultate erheblich steigern. Die Zuverlässigkeit kann durch weiteres Testen erhöht werden. Damit die Bedienbarkeit zunimmt, sollte das Framework in eine einfache Deploymenteinheit verpackt werden.

Literaturverzeichnis

[ANT05]

Apache Ant Manual

http://ant.apache.org/manual/index.html

[BECGAM_A]

Kent Beck, Erich Gamma

JUnit a Cook’s Tour

http://junit.sourceforge.net/doc/cookstour/cookstour.htm

[BECGAM_B]

Kent Beck, Erich Gamma

JUnit CookBook

http://junit.sourceforge.net/doc/cookbook/cookbook.htm

[CLARK05]

Mike Clark

JunitPerf Documentation

http://www.clarkware.com/software/JUnitPerf.html

[DEPREC]

J2SE 1.5 Javadoc

http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

[GHJV94]

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Design Patterns: Elements of reusable object oriented software

Addison Wesley Publishing Company 1994

[HIGON04]

Richard Hightower, Warner Onstine

Professional Java Tools for Extreme Programming

Wrox Press 2004

[HTTPUNIT]

HTTPUnit Tutorial

http://httpunit.sourceforge.net/doc/tutorial/index.html

[HYDE99]

Paul Hyde

Java Thread Programming

Sams Publishing 1999

[JETTY]

Jetty Tutorial

http://jetty.mortbay.org/jetty/tut/index.html

[JMETER]

JMeter Manual

http://jakarta.apache.org/jmeter/

[JSR166]

JSR 166

http://www.jcp.org/en/jsr/detail?id=166

Backport Java 1.4 - http://www.mathcs.emory.edu/dcl/util/backport-util-concurrent/

[JUnit]

JUnit 3.8

Manual and Binaries

http://junit.sourceforge.net

[LINFRO03]

Johannes Link and Peter Fröhlich

Unit Testing in Java: How Tests Drive the Code

Morgan Kaufmann Publishers 2003

[NAGIOS05]

Nagios Manual

http://nagios.sourceforge.net/docs/2_0/

[OAWO04]

Scott Oaks, Henry Wong

Java Threads, 3rd Edition

O'Reilly 2004

[RAFUNC]

Rational Functional Tester

Product Overview

http://www-306.ibm.com/software/awdtools/tester/functional/

[RAPERF]

Rational Performance Tester

Product Overview

http://www-306.ibm.com/software/awdtools/tester/performance/

[WELL04]

Andy Wellings

Concurrent and Real-Time Programming in Java

John Wiley & Sons 2004

Quelltexte

Bitte die README auf der CD beachten.