Test Driven Development

als Maßnahme zur Qualitätssicherung bei der Softwareentwicklung

am Beispiel eines Bestandsführungssystems

Auszug[1] der **Masterarbeit**

im Fachgebiet **Software-Engineering**

GEORG-SIMON-OHM
HOCHSCHULE NÜRNBERG

vorgelegt von: Stefan Macke
Matrikelnummer: 2071371
eingereicht im: Sommersemester 2009
Erstgutachter: Prof. Dr. Hans-Georg Hopf
Zweitgutachter: Prof. Dr.-Ing. Helmut Herold

© 2009

Dieses Werk einschließlich seiner Teile ist **urheberrechtlich geschützt**. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen sowie die Einspeicherung und Verarbeitung in elektronischen Systemen.

[1] Dieses Dokument enthält lediglich die Teile der Arbeit, die sich mit der Entwicklung von NUNIT befassen.

Zusammenfassung

Test Driven Development (TDD) hat das Ziel, anpassbare Software zu entwickeln. Mit einer Vielzahl von Unit-Tests wird sichergestellt, dass die Software nach Änderungen immer noch wie erwartet funktioniert. Dadurch kann ihre interne Struktur gefahrlos selbst weitreichenden Refactorings unterzogen werden, um ihre Qualität zu erhöhen und den Anforderungen bestmöglich zu entsprechen.

Die vorliegende Masterarbeit gibt eine Einführung in das Thema Test Driven Development und zeigt insbesondere seine Auswirkungen auf die Softwarequalität. Im Rahmen der Arbeit wird ein Unit-Test-Framework für die Programmiersprache Natural entwickelt und eine C#-Applikation einem umfangreichen Refactoring unterzogen.

Die testgetriebene Entwicklung erweist sich als wirkungsvoller Ansatz, der die Anwendung vieler zentraler Prinzipien eines soliden Softwaredesigns forciert und somit tatsächlich zu qualitativ hochwertigem Code führt. Außerdem wird die Änderbarkeit der Software durch die Unit-Tests deutlich verbessert.

Abstract

The goal of Test Driven Development (TDD) is the implementation of software that can easily be modified and maintained. Large numbers of unit tests make sure that the software still works as expected even after bigger changes. That gives developers the courage to refactor their code mercilessly to improve its quality and make sure it meets all requirements.

This master's thesis provides the reader with an introduction to Test Driven Development and its effects on software quality. Furthermore, a unit test framework for the programming language Natural is developed and a C# application is refactored to make it testable.

Test Driven Development proves itself as a useful development method that enforces the use of core principles of software design and eventually leads to high quality code. In addition, the tests definitely improve the maintainability of the software.

© Stefan Macke

Inhaltsverzeichnis

Abkürzungsverzeichnis II

Abbildungsverzeichnis IV

Verzeichnis der Listings V

1. Einleitung 1

2. Entwicklung von NUNIT 3
- 2.1. Frameworks für Unittests . . . 4
  - 2.1.1. Definitionen . . . 4
    - 2.1.1.1. Assertions . . . 5
    - 2.1.1.2. Test, TestCase und TestSuite . . . 5
    - 2.1.1.3. Stubs und Mocks . . . 6
  - 2.1.2. Architektur von JUnit . . . 7
- 2.2. Vorgehen bei der Entwicklung von NUNIT . . . 9
- 2.3. Testgetriebene Entwicklung von NUNIT . . . 10
  - 2.3.1. Aufruf eines TestCases . . . 11
  - 2.3.2. Automatische Überprüfung des Testergebnisses . . . 12
  - 2.3.3. Aufruf eines einzelnen Tests . . . 15
  - 2.3.4. Dynamische Ermittlung von definierten Tests in TestCases . 18
  - 2.3.5. SETUP und TEARDOWN ausführen . . . 21
  - 2.3.6. Weiterer Verlauf der Entwicklung . . . 25
- 2.4. Das fertige Framework . . . 25
- 2.5. Nachbetrachtung und Ausblick . . . 29
  - 2.5.1. Offene Probleme . . . 31
  - 2.5.2. Erweiterungsmöglichkeiten . . . 32

Literaturverzeichnis i

A. Anhang A1
- A.1. Einführung in Natural . . . A2
- A.2. Listings und Abbildungen . . . A5

Abkürzungsverzeichnis

4GL .... Fourth Generation Language
AAA .... Arrange, Act, Assert
AO .... Alte Oldenburger
API .... Application Programming Interface
BaFin .... Bundesanstalt für Finanzdienstleistungsaufsicht
BDD .... Behaviour Driven Development
BDUF .... Big Design Up Front
Ca .... Afferent Coupling
CC .... Cyclomatic Complexity
Ce .... Efferent Coupling
CI .... Continuous Integration
CVS .... Concurrent Versions System
DIP .... Dependency Inversion Principle
GUI .... Graphical User Interface
H .... Relational Cohesion
IDE .... Integrated Development Environment
IEEE .... Institute of Electrical and Electronics Engineers
ISO .... International Organization for Standardization
ISP .... Interface Segregation Principle
ISTQB .... International Software Testing Qualifications Board
KISS .... Keep it simple, stupid
KLOC .... Kilo-Lines of Code
LCOM .... Lack of Cohesion of Methods
LDA .... Local Data Area
LOC .... Lines of Code
LSP .... Liskov Substitution Principle
MVC .... Model-View-Controller

NASA .... National Aeronautics and Space Administration
NOM .... Number of Methods
OCP .... Open Closed Principle
PDA .... Parameter Data Area
PKV .... Private Krankenversicherung
SOLID .... Akronym der fünf Designprinzipien von Martin (2009)
SPoD .... Single Point of Development (Natural-IDE)
SRP .... Single Responsibility Principle
TDD .... Test Driven Development
VERSIS .... Versicherungsinformationssystem
XP .... eXtreme Programming
YAGNI .... You aren't gonna need it

Abbildungsverzeichnis

2.1. Architektur von JUnit inkl. verwendeter Design Patterns 8
2.2. Aufruf von NUNIT mit Stub-Implementierung von TESTCASE 12
2.3. Aufruf von NUNIT mit Implementierung von TESTCASE 13
2.4. Komponenten von NUNIT nach dem ersten größeren Refactoring . . 23
2.5. Screenshot eines Testlaufs von NUNIT (roter Balken) 27
2.6. Screenshot der Detailausgabe eines NUNIT-Testlaufs 28

A.1. Architektur von NUNIT A6

Verzeichnis der Listings

2.1. Zentrale Abläufe in JUnit (Pseudocode) . . . 9
2.2. NUNIT testet den Aufruf eines TestCases . . . 11
2.3. Stub des Subprograms TESTCASE . . . 12
2.4. Implementiertes Subprogram TESTCASE, das den Test besteht . . . 13
2.5. NUNIT testet automatisch die Ausführung eines TestCases . . . 14
2.6. Die Konstante ASSERTION-ERROR in der LDA NUCONST . . . 15
2.7. Subroutine ASSERT-TRUE . . . 15
2.8. NUNIT nach dem ersten Refactoring . . . 16
2.9. TestCase TESTCASE, der einen bestimmten Test gezielt ausführen kann . . . 18
2.10. PDA NUTESTP . . . 18
2.11. TestCase TESTCASE mit eingebundener PDA . . . 18
2.12. NUNIT mit eingebundener PDA . . . 19
2.13. NUNIT ermittelt Tests dynamisch . . . 20
2.14. Stub-Implementierung von NUPARSE . . . 21
2.15. Beispiel einer Wrapper-Routine zur Umwandlung des Ergebnisses einer Assertion in einen booleschen Wert . . . 22
2.16. Beispiel für den automatischen Test des Ergebnisses einer Assertion durch den Aufruf ihrer Wrapper-Routine . . . 22
2.17. Einfache Bildschirmausgabe einer Assertion im Fehlerfall . . . 22
2.18. Automatischer Test des Ablaufs eines Tests (NUTCSEQ) . . . 24
2.19. *Template Method* zum Ablauf eines Tests (CopyCode NUTCTEMP) . 24
2.20. TestCase NUTCTEST inkl. Setup, Teardown und Protokollierung seines Ablaufs . . . 33
2.21. Beispiel für einen NUNIT-TestCase . . . 34

A.1. Workfile mit NUNIT-TestCases eines Benutzers . . . A5

1. Einleitung

In einer Studie aus dem Jahr 2004 bezifferten 74% der teilnehmenden Unternehmen ihre jährlichen Kosten aufgrund mangelhafter Softwarequalität auf bis zu 500.000 Euro und insgesamt 51% der Befragten gaben an, dass ihre Software mit Qualitätsproblemen an die Kunden augeliefert wurde (Zimmermann 2009). Dabei ist es eigentlich ganz einfach, qualitative Software zu entwickeln, die den Kundenwünschen entspricht: Man muss sie lediglich ausgiebig testen. Microsoft beschäftigt laut Bill Gates ebenso viele Software*tester* wie -entwickler, wobei letztere auch noch die Hälfte ihrer Arbeitszeit mit dem Testen der von ihnen entwickelten Software verbringen (Foley und Murphy 2002). Und bei Siemens werden bis zu 50% des Projektbudgets der Softwareentwicklung für Testaktivitäten verwendet (Paulisch u. a. 2009).

Doch obwohl die meisten Programmierer wissen, dass sie Tests für ihren Code schreiben sollten, tun dies in der Praxis die wenigsten (Beck und Gamma 2000). Testen gehört für sie nicht zu ihren Aufgaben. Es wird als langweilig und destruktiv im Vergleich zum Entwurf und der Implementierung von Software angesehen und lieber an spezielle Tester abgegeben (Stal 2005, S. 114). Dabei ist das Testen so alt wie die Softwareentwicklung selbst. Bereits in den 1960er Jahren, als sich die Softwareentwicklung noch auf das Ausstanzen von Lochkarten beschränkte, wurden im Rahmen des Mercury-Programms[2] der NASA Tests verwendet, um die Funktionalität der Software zu prüfen (Feathers und Freeman 2009, 5:06). Laut Weinberg (2008), einem Pionier der ersten Stunde, galt es damals sogar explizit als unprofessionell, keine Tests zu haben, weil das Ausführen der Software auf den Großrechnern viel zu teuer war, um Fehlschläge in Kauf nehmen zu können.

Diese Tests wurden daher sogar *vor* der eigentlichen Entwicklung der Software entworfen: ein Vorgehen, das heute als *Test-First* bezeichnet wird. Diese Reihenfolge ist charakteristisch für das heutige **Test Driven Development** (TDD), das in dieser Arbeit näher betrachtet werden soll. Denn obwohl die Ursprünge dieses Vorgehens der Softwareentwicklung bereits fast 50 Jahre zurückliegen und auch der Begriff TDD selbst schon vor 10 Jahren eingeführt wurde, hat es sich längst noch nicht

[2]Das Mercury-Programm war das erste bemannte Raumfahrtprogramm der USA.

weiträumig in der Praxis etabliert. Dies mag vielleicht auch am weit verbreiteten Missverständnis liegen, dass TDD ausschließlich auf das Testen abzielt (immerhin steht der Begriff *Test* sogar im Namen), das bei den meisten Entwicklern wie gesagt nicht sehr hoch angesehen ist. Test Driven Development verfolgt jedoch vielmehr das Ziel, sauberen Code und eine solide Softwarearchitektur zu erzeugen, wobei die *Unit-Tests* lediglich Mittel zum Zweck sind. Sie erlauben es den Entwicklern nämlich, ihren bereits vorhandenen Code rücksichtslos zu überarbeiten, ohne Gefahr zu laufen, bereits vorhandene Funktionalität zu beeinträchtigen.

Die Architektur und das Design von Software gehören nach Meinung des Autors zu den Kernaufgaben eines jeden Softwareingenieurs. Und ähnlich sieht das auch ein Großteil der Vertreter der Industrie, die in einer Umfrage aus dem Jahr 2009 zusammen mit Hochschullehrern gebeten wurden, die Wichtigkeit verschiedener Kenntnisse, Fähigkeiten und Fertigkeiten von Softwareingenieuren zu beurteilen (Coldewey 2009, S. 12). Die Industrievertreter hatten jedoch zusätzlich die Aufgabe, die tatsächlichen Qualifikationen der Absolventen in diesen Bereichen zu bewerten, wobei diese Einschätzung teils deutlich hinter den Erwartungen zurückblieb. So wurden hauptsächlich die praxisnahen Themen wie das *Refactoring* oder eben die testgetriebene Softwareentwicklung lediglich mit *ausreichend* bewertet. Dabei führen genau diese Ansätze in der Praxis zu stabilen und erweiterbaren Softwarearchitekturen, wie sie von jedem Softwareingenieur erwartet werden. Daher ist eine eingehende Betrachtung des Test Driven Development im Rahmen dieser Masterarbeit im Bereich Software-Engineering durchaus sinnvoll und praxisrelevant.

2. Entwicklung von NUNIT

Zum testgetriebenen Entwickeln von Software ist ein Unit-Test-Framework unerlässlich. Zum Zeitpunkt dieser Arbeit liegt für die Programmiersprache Natural jedoch noch kein solches Framework vor. Die Liste von Jeffries (2009) führt Natural nicht auf und als Antwort auf eine direkte Anfrage des Autors bei der Software AG, dem Hersteller von Natural, kam lediglich die Aussage, dass es bislang keine Möglichkeit gibt, Natural-Anwendungen automatisiert zu testen. Auch bei Gesprächen des Autors mit anderen Entwicklern auf Veranstaltungen der Natural-Community konnten keine Lösungen für automatisierte Natural-Tests bei anderen Unternehmen in Erfahrung gebracht werden.

Das auf Eclipse basierende Produkt *NaturalONE* der Software AG, das sich zum Zeitpunkt dieser Arbeit noch in der Entwicklung befindet, soll eine rudimentäre Möglichkeit für den Test von Natural-Programmen bieten (Kronauer 2009, S. 13f.). Dieser beschränkt sich jedoch auf die Ausführung von Programmen mit definierten Eingangsparametern und dem anschließenden Vergleich mit festgelegten Ausgangswerten. Eine Möglichkeit, umfangreichere Fixtures zu definieren, sowie komplexere Assertions fehlen. Außerdem ist es nicht möglich, mehrere Tests automatisch durchzuführen.

Eine recht umständliche Möglichkeit, Natural-Programme automatisch zu testen, wäre beim Einsatz der *Natural Business Services* gegeben. Dabei werden Natural-Programme als (Web-)Services angeboten, die dann über die üblichen Test-Frameworks getestet werden könnten. Die Nachteile dabei wären zum Einen, dass jedes zu testende Programm zunächst als Service angeboten werden müsste, was einen hohen Aufwand für den Entwickler zur Folge hätte, und zum Anderen, dass die Tests nicht in Natural, sondern in der Programmiersprache des eingesetzten Test-Frameworks implementiert werden müssten. Außerdem sind die Natural Business Services ein kommerzielles Produkt, dessen Kosten den potentiellen Nutzen der sehr umständlichen Tests bei Weitem übersteigen. Für die Praxis ist dieser Ansatz somit nicht relevant.

Im Folgenden wird daher die Konzeption und Implementierung eines Unit-Test-Frameworks für die Programmiersprache Natural beschrieben. Das Framework soll

den Namen NUNIT tragen. Zwar ist eine Unterscheidung zu NUnit im .NET-Umfeld damit auf die Schreibweise in Großbuchstaben beschränkt, aber der Hintergrund der Nomenklatur ist ganz pragmatisch: Da die Programmnamen in Natural auf acht Zeichen beschränkt sind, soll ein kurzer und trotzdem sprechender Name gewählt werden, der noch ein wenig Spielraum für Ergänzungen lässt. Und bei NATUNIT wären bereits sieben der acht Zeichen verwendet.

Da der Kreis der Natural-Anwender recht eingeschränkt ist und Kenntnisse der Programmiersprache nicht vorausgesetzt werden können, wird dieser Arbeit mit Kapitel A.1 im Anhang eine kurze Einführung in die Programmiersprache beigefügt. Dabei werden hauptsächlich der grundsätzliche Aufbau von Natural und die verschiedenen Programm- und Variablentypen erläutert. Andere Natural-spezifische Konzepte werden während der Beschreibung der Implementierung in den entsprechenden Kapiteln erklärt, sofern sie benötigt werden. Die Ausführungen basieren auf der offiziellen Natural-Dokumentation der Software AG (2009a).

2.1. Frameworks für Unittests

Unit-Tests sind im Rahmen des Test Driven Developments unerlässlich. Doch ohne die Unterstützung durch Werkzeuge sind Unit-Tests nur umständlich zu implementieren. Das führt dazu, dass die Entwickler, die meist ohnehin schon wenig Lust verspüren, Tests zu schreiben, gleich wieder damit aufhören. Daher gibt es inzwischen eine große Anzahl an Unit-Test-Frameworks, die dem Entwickler das Erstellen und Verwalten von Unit-Tests einfacher machen. Laut Gamma (2006) ist das oberste Ziel eines solchen Frameworks, Entwickler dazu zu bringen, Tests zu schreiben, indem genau das einfach und schnell geht. Ambler (2009) sagt gar, dass TDD ohne ein Unit-Test-Framework unmöglich ist. In den folgenden Kapiteln werden zunächst einige Begriffe im Umfeld von Unit-Test-Frameworks erläutert, bevor der grundsätzliche Aufbau des wohl bekanntesten Unit-Test-Frameworks JUnit vorgestellt wird, an dem sich viele weitere Frameworks orientiert haben, und das auch als Vorlage zur Implementierung von NUNIT dient.

2.1.1. Definitionen

Im Folgenden werden nun einige grundlegende Begriffe aus dem Bereich der Unit-Tests erläutert, die später bei der Entwicklung von NUNIT verwendet werden.

2.1.1.1. Assertions

Die eigentlichen Prüfungen in den Unit-Tests werden mit **Assertions** durchgeführt. Assertions sind Zusicherungsanweisungen, die bei Verletzung einer Zusicherung einen (vom Unit-Test-Framework behandelten) Fehler auslösen (Stal 2005, S. 116). In objektorientierten Frameworks lösen die Assertions meist Exceptions aus (bei JUnit z. B. einen AssertionFailedError), die dann in einem `try/catch`-Block aufgefangen werden und den Test als Ergebnis fehlschlagen lassen. Der Sinn von Assertions ist, absolut jede menschliche Interpretation aus den Unit-Tests verschwinden und alle Annahmen durch den Computer testen zu lassen (Beck 2003, S. 157).

Die einfachsten Assertions dürften assertTrue() und assertFalse() sein, es gibt jedoch noch eine ganze Reihe an weiteren sinnvollen Assert-Methoden wie z. B. assertEquals() oder assertNotNull(). Einige Unit-Test-Frameworks bieten auch noch komplexere Assertions an, wie z. B. CollectionAssert.AllItemsAreUnique() oder StringAssert.AreEqualIgnoringCase in NUnit.

Die meisten Assertions haben drei Parameter: den erwarteten Wert (`expected`), den tatsächlichen Wert (`actual`) und eine optionale Meldung, die beim Fehlschlagen der Assertion ausgegeben wird. Üblicherweise wird der erwartete Wert zuerst angegeben, aber nicht alle Entwickler finden das sinnvoll oder denken immer daran (Beck 2003, S. 157). In NUnit haben die Assertions daher einen etwas anderen Aufbau: Es gibt lediglich die Assertion Assert.That(), die als Parameter den tatsächlichen Wert und ein *Constraint* annimmt, gegen das der Wert geprüft wird.[3] Diese Constraints implementieren *Fluent Interfaces* und sind kaskadierbar. Dadurch sind die Assertions deutlich leichter zu lesen und zu verstehen:

```
Assert.That(3, Is.Not.GreaterThan(7));
```

`Is`, `Not` und `GreaterThan` sind drei Constraints, die einfach hintereinander gehängt werden können.

2.1.1.2. Test, TestCase und TestSuite

Die einzelnen Assertions werden innerhalb von **Tests** ausgeführt, die üblicherweise in Form von Methoden vorliegen. Man kann grundsätzlich beliebig viele Assertions pro Test aufrufen, aber es sollten so wenig Assertions wie möglich pro Test verwendet werden. In älteren Versionen von JUnit und anderen Frameworks wie

[3] Zusätzlich zu diesem Vorgehen bietet NUnit auch die bekannten Assertions an, aber im Rahmen dieser Arbeit soll das neue Verfahren verwendet werden.

z. B. PHPUnit[4] müssen die Namen der Test-Methoden mit `test` beginnen, damit sie automatisch als Tests erkannt und ausgeführt werden können. Ab Version 4 von JUnit wird allerdings die Annotation `@Test` für diesen Zweck verwendet. In NUnit werden die Test-Methoden mit dem Attribut `[Test]` ausgezeichnet. Egal in welcher Form die Tests gekennzeichnet werden, zu ihrer Ermittlung wird immer eine Form von *Reflection* benötigt, da dynamisch zur Laufzeit ermittelt werden muss, welche Test-Methoden es aktuell gibt.

Die Tests organisiert man am besten um ein *Fixture* herum. Dieses Fixture stellt die Voraussetzungen dar, auf die alle ihm zugeordneten Tests aufsetzen. In den meisten Frameworks werden die Test-Methoden in einer Test-Klasse zusammengefasst, die diesem Fixture entspricht, und **TestCase** genannt wird. Bei JUnit wurden bis zur Version 3 die TestCases von der Klasse TestCase abgeleitet,[5] während bei NUnit hierfür die Klassen mit dem Attribut [TestFixture] ausgezeichnet werden, das von der Nomenklatur her etwas besser zur Aufgabe eines TestCases passt. Beck (2003, S. 162) sagt nämlich, dass es pro Fixture genau einen TestCase geben sollte. TestCases enthalten eine setUp-Methode, die das Fixture vor dem Aufruf jedes einzelnen Tests neu aufbaut, und eine tearDown-Methode, die es danach wieder abbaut. Die Kennzeichnung der beiden Methoden ist wieder unterschiedlich zwischen den einzelnen Frameworks: In NUnit werden sie z. B. mit den Attributen `[SetUp]` und `[TearDown]` versehen, in JUnit mit den Annotations `@Before` und `@After`.

Zur weiteren Gruppierung der TestCases dienen in JUnit und anderen vergleichbaren Frameworks die **TestSuites**, während in NUnit hierfür das Attribut `[Category]` verwendet wird. Dadurch kann man die TestCases dann z. B. nach getesteten Komponenten oder Dauer der Ausführung aufteilen und nur die Tests laufen lassen, die gerade benötigt werden.

2.1.1.3. Stubs und Mocks

Beim Einstieg in die Entwicklung von Unit-Tests kommt man insbesondere bei objektorientierter Software schnell an die Grenzen dessen, was man testen kann. Die Assertions können nur harte Werte überprüfen, aber nicht alle zu testenden Ergebnisse liegen in Form von Zahlen, Strings oder anderen Objekten vor. Oft muss nicht das Endergebnis, sondern die Kommunikation der Objekte untereinander auf dem Weg zum Ziel getestet werden. Genau dafür werden **Stubs** und **Mocks** benötigt.

[4] http://www.phpunit.de/
[5] In Version 4 haben die Klassen keine besondere Auszeichnung mehr, nur die Methoden bekommen die Annotation `@Test`.

Mocks testen die Interaktionen eines Objekts mit anderen Objekten, ohne diese selbst zu testen (Banks 2008, 8:05). Mock-Objekte spielen dem zu testenden Objekt vor, ein Objekt eines von diesem benötigten Typs zu sein. Soll z. B. ein Objekt getestet werden, dass Daten auf ein Speichermedium schreibt und dafür ein Objekt vom Typ StreamWriter benötigt, kann ein Mock-Objekt erzeugt und dem Test-Objekt untergeschoben werden, das das Verhalten eines StreamWriters simuliert und am Ende der Interaktion prüft, ob die richtigen Methoden zum Speichern der Daten in der richtigen Reihenfolge aufgerufen wurden. Wenn das Mock-Objekt dann ungültige Methodenaufrufe feststellt, schlägt der Test fehl.

Ähnlich funktionieren Stubs, mit dem zentralen Unterschied, dass sie Tests nicht zum Fehlschlagen bringen können (Osherove 2008, S. 85). Auch Stubs simulieren zwar das Verhalten anderer Objekte, mit denen das Test-Objekt kommuniziert, aber anders als Mocks testen Stub-Objekte selbst nichts. Es kommen vielmehr die üblichen Assertions zum Einsatz, die die Ergebnisse des Test-Objekts prüfen, das nur eben nicht mit echten anderen Objekten interagiert hat, sondern mit simulierten. Ein Stub muss sich daher z. B. auch nicht die Reihenfolge seiner Methodenaufrufe merken, sondern gibt einfach nur definierte Test-Werte zurück.

Sowohl Stubs als auch Mocks können vom Entwickler selbst implementiert werden, indem er im obigen Beispiel etwa eine Klasse StubStreamWriter erzeugt, die von StreamWriter abgeleitet ist, und diese dem Test-Objekt unterschiebt. Die Methoden der Klasse liefern dann z. B. nur hart codierte Werte und schreiben auch keine Daten auf die Festplatte, verhalten sich aber nach außen hin so wie die normalen Methoden es auch täten. Allerdings wird die manuelle Implementierung von Stubs und vor allem von Mocks, die sich wie gesagt auch die Reihenfolge ihrer Methodenaufrufe merken müssen, schnell sehr mühsam. Daher gibt es **Mocking-Frameworks**, die das Erstellen von Stubs und Mocks deutlich vereinfachen. Sie ermitteln z. B. mittels Reflection, welche Methoden die Mock-Objekte anbieten müssen und liefern definierte Werte zurück (Stal 2005, S. 117).

2.1.2. Architektur von JUnit

Abbildung 2.1 (eigene Anfertigung in Anlehnung an Gamma (2006)) zeigt die grundlegende Architektur von JUnit.

Im Kern besteht JUnit aus 4 Komponenten:

- Das Interface Test definiert lediglich die Methode run(), die einen Test ausführt. Es entspricht dem Entwurfsmuster *Command*, welches das Ausführen einer

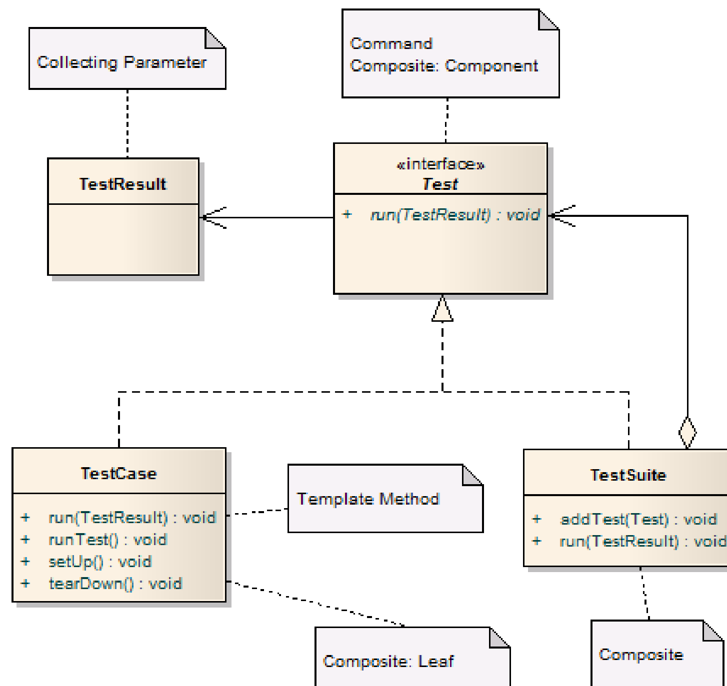

Abbildung 2.1.: Architektur von JUnit inkl. verwendeter Design Patterns

beliebigen Methode in einem Objekt hinter einer definierten Schnittstelle (in diesem Fall run()) kapselt. Über diese Schnittstelle können dann die Test-Methoden mit unterschiedlichen Namen aufgerufen werden.

- Die Klasse TestCase implementiert Test und stellt die eigentliche Logik zur Ausführung von Tests und auch die Assertion-Methoden bereit. Die Methode run() sorgt als *Template Method* (siehe Zeilen 1–16 in Listing 2.1) dafür, dass vor jedem Test setUp() und danach tearDown ausgeführt wird. Der eigentliche Test wird dann in runTest() ausgeführt, wobei die konkret aufzurufende Test-Methode mittels Reflection ermittelt wird (siehe Zeilen 18–21 in Listing 2.1). Fehlgeschlagene (*Failure*) und fehlerhafte (*Error*) Tests werden im `try/catch`-Block protokolliert. Erstere sind das Ergebnis einer fehlgeschlagenen Assertion, während letztere irgendeinen Laufzeitfehler enthalten (z. B. einen Null-Pointer) und daher abbrechen.
- Die Klasse TestResult sammelt als *Collecting Parameter* die Ergebnisse der Tests ein, indem sie bei allen Aufrufen von run() übergeben wird.
- Die Klasse TestSuite gruppiert mehrere TestCases und implementiert auch das Interface Test. Sie stellt damit ein *Composite* für Tests dar (siehe Zeilen 23–27 in Listing 2.1).

```
TestCase.run(TestResult result) {
  result.startTest(this);
  this.setUp();
  try {
    this.runTest();
  }
  catch (AssertionFailedError e) {
    result.addFailure(this, e);
  }
  catch (Throwable e) {
    result.addError(this, e);
  }
  finally {
    this.tearDown();
  }
}

TestCase.runTest() throws Throwable {
  Method runMethod = getClass().getMethod(mName);
  runMethod.invoke();
}

TestSuite.run(TestResult result) {
  for (Test t: this.tests) {
    t.run(result);
  }
}
```

Listing 2.1: Zentrale Abläufe in JUnit (Pseudocode)

Der Ablauf eines Testdurchlaufs kann wie folgt zusammengefasst werden:

1. Der TestRunner von JUnit initialisiert einen TestResult und ermittelt alle auszuführenden Tests (TestCases oder TestSuites, da er dank des *Composite*-Patterns beide einheitlich behandeln kann).

2. Er ruft auf allen Tests die Methode run() mit dem TestResult als Parameter auf.

3. Jeder TestCase ermittelt seine auszuführenden Methoden und führt die *Template Method* aus. Aufgetretene Fehler werden dabei im TestResult protokolliert.

4. Wenn alle Tests gelaufen sind, zeigt der TestRunner das Ergebnis an.

2.2. Vorgehen bei der Entwicklung von NUNIT

NUNIT soll ganz im Sinne des Themas dieser Arbeit testgetrieben entwickelt werden, um zu zeigen, dass prinzipiell auch ohne ein bereits vorhandenes Test-Framework

testgetrieben vorgegangen werden kann. Dabei werden zunächst manuelle Tests verwendet bis das Framework im Laufe der Entwicklung irgendwann die benötigten Funktionen bereitstellt, um automatische Tests durchzuführen. Dann können die manuellen in automatische Tests überführt werden, sodass sich das Framework selbst testet.

Die ersten fünf Schritte der testgetriebenen Entwicklung von NUNIT werden in Kapitel 2.3 detailliert mit Quelltext-Beispielen beschrieben. Jeder Schritt folgt dabei dem gleichen Ablauf:

1. **Problem**: Was ist die nächste Anforderung, die NUNIT umsetzen soll?
2. **Test**: Wie kann die erfolgreiche Umsetzung dieser Anforderung getestet werden?
3. **Test lauffähig machen**: Üblicherweise läuft der Test aufgrund fehlender Abhängigkeiten zunächst nicht. Diese sind nun zu implementieren.
4. **Test schlägt fehl**: Sobald der Test läuft und fehlschlägt, wird er durch die einfachste mögliche Implementierung im Produktivcode zum erfolgreichen Durchlaufen gebracht.
5. **Refactoring**: Der Produktivcode muss dann abschließend noch refaktorisiert werden.

2.3. Testgetriebene Entwicklung von NUNIT

Zunächst sollen einige grundsätzliche Überlegungen zur Architektur des Frameworks angestellt werden. Der TestRunner von NUNIT muss zwangsläufig ein Program[6] sein, da nur ein solches ausgeführt werden kann. Da die TestCases ihre Ergebnisse als Parameter an den TestRunner zurückliefern müssen, kommen nur die Programmtypen Subprogram oder Subroutine in Frage, da globale Variablen vermieden werden sollen, die beim Einsatz von Programs verwendet werden müssten. Da die TestCases aber jeweils mehrere Tests enthalten sollen, können Subroutines ausgeschlossen werden, da sich diese nicht ineinander schachteln lassen. Daher werden die TestCases als Subprograms implementiert. Die Tests können dann als Subroutines inline in den TestCases definiert und dabei sogar 32 Zeichen für einen sprechenden Namen verwendet werden. Die Assertions sollten dann als externe Subroutines implementiert

[6] Zur Unterscheidung der Programmtypen siehe Kapitel A.1.

werden, da sie auch einen sprechenden Namen benötigen, um die Tests lesbar zu machen, und aus mehreren TestCases heraus aufgerufen werden müssen.

Die sonstige Nomenklatur (z. B. TestCase und Assert) wird von JUnit übernommen und als Entwicklungssprache wird daher – und um das Framework nach der Fertigstellung in der Natural-Community vorstellen und verbreiten zu können – Englisch verwendet. Alle Sourcen der im Folgenden beschriebenen Entwicklungsschritte und die des fertigen Frameworks sind auf der dieser Arbeit beiligenden DVD im Verzeichnis `Listings\NUNIT` zu finden.

2.3.1. Aufruf eines TestCases

Problem Wie ruft man einen einzelnen TestCase auf und prüft, ob der Aufruf auch tatsächlich stattgefunden hat?

Test Als erstes wird das Program NUNIT (Listing 2.2) erstellt, das im Folgenden das zentrale Steuerprogramm von NUNIT werden soll. Um den Aufruf einen TestCases zu prüfen, wird ein `CALLNAT` auf das Subprogram TESTCASE mit dem boolschen Parameter `#TESTCASE-WAS-RUN` durchgeführt und manuell geprüft, ob der Wert sich nach dem Aufruf des TestCases verändert hat.

```
1  DEFINE DATA
2  *
3  LOCAL
4  01 #TESTCASE-WAS-RUN (L)
5  *
6  END-DEFINE
7  *
8  #TESTCASE-WAS-RUN := FALSE
9  WRITE 'before TestCase..:' #TESTCASE-WAS-RUN
10 CALLNAT 'TESTCASE' #TESTCASE-WAS-RUN
11 WRITE 'after TestCase...:' #TESTCASE-WAS-RUN
12 *
13 END
```

Listing 2.2: NUNIT testet den Aufruf eines TestCases

Test lauffähig machen NUNIT bricht ab, da das Subprogram TESTCASE nicht vorhanden ist. Es wird daher als Subprogram mit einem booleschen Parameter, aber ohne Implementierung angelegt (Listing 2.3). Nun läuft NUNIT, liefert aber noch nicht das gewünschte Ergebnis: Der Parameter ändert sich nicht (siehe Abbildung 2.2).

```
1 DEFINE DATA
2 *
3 PARAMETER
4 01 #TESTCASE-WAS-RUN (L)
5 *
6 END-DEFINE
7 *
8 END
```

Listing 2.3: Stub des Subprograms TESTCASE

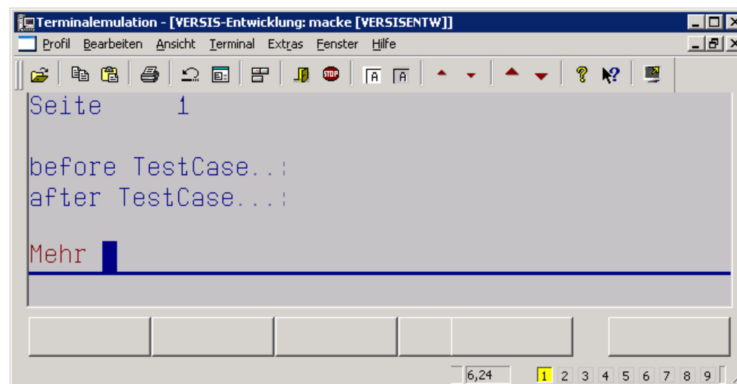

Abbildung 2.2.: Aufruf von NUNIT mit Stub-Implementierung von TESTCASE

Test schlägt fehl Die einfachste Lösung, um den Test zu bestehen, ist das Setzen des Parameters in TESTCASE, wie in Listing 2.4 zu sehen ist. Dies wird gleich in eine Subroutine ausgelagert, da dies der geplanten Struktur entspricht. Danach läuft NUNIT durch und erzeugt die Ausgabe in Abbildung 2.3.

Refactoring Im Moment ist noch kein Refactoring nötig und möglich.

2.3.2. Automatische Überprüfung des Testergebnisses

Problem Wie kann das Ergebnis des Aufrufs des TestCases automatisch geprüft werden, ohne dass manuell das `X` kontrolliert werden muss?

Test Der automatische Test des Rückgabeparameters von TESTCASE wäre theoretisch sehr einfach, wenn es in Natural eine Möglichkeit gäbe, nach der Prüfung einer bestimmten Bedingung einen Fehler zu erzeugen, wie es z. B. mit Debug.Assert() in C# oder `assert()` in Java auch ohne ein Unit-Test-Framework möglich ist. Natural bietet laut der offiziellen Dokumentation der Software AG (2009a) jedoch leider keinen Befehl dafür an, sodass die Prüfung mit einer `if`-Abfrage (Listing 2.5)

```
1  DEFINE DATA
2  *
3  PARAMETER
4  01 #TESTCASE−WAS−RUN (L)
5  *
6  END−DEFINE
7  *
8  PERFORM TEST
9  *
10 DEFINE SUBROUTINE TEST
11 #TESTCASE−WAS−RUN := TRUE
12 END−SUBROUTINE
13 *
14 END
```

Listing 2.4: Implementiertes Subprogram TESTCASE, das den Test besteht

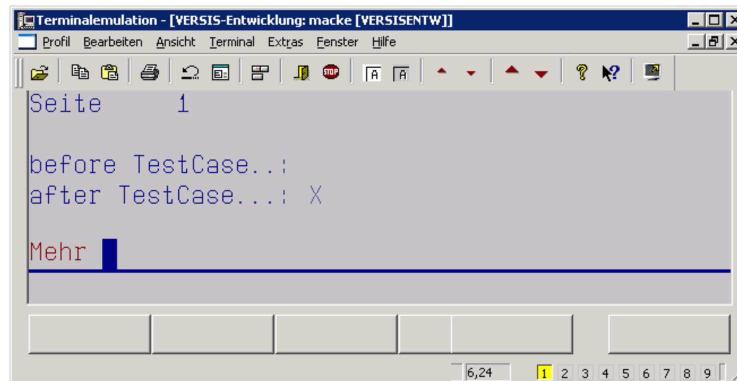

Abbildung 2.3.: Aufruf von NUNIT mit Implementierung von TESTCASE

durchgeführt wird. Natural bietet keine Unterstützung von Exceptions oder anderen Mechanismen zur Fehlerbehandlung. Daher müssen Entwickler mit Returncodes arbeiten, um fachliche Fehlersituationen zu kennzeichnen. *Echte* Programmfehler (wie z. B. ein fehlendes Subprogram oder die falsche Parameteranzahl beim Aufruf eines solchen) werden über Natural-Fehler angezeigt, die das Programm sofort beenden. Diese Natural-Fehler bestehen letztlich nur aus einer Fehlernummer, der ein Fehler- und ein Hilfetext zugewiesen ist. Der Entwickler kann diese Fehler explizit provozieren, indem er der Systemvariablen `*ERROR-NR` einfach einen Wert zuweist (einen Befehl wie `raise` oder `throw` gibt es nicht). Die Natural-Fehler können in einem `ON ERROR`-Block behandelt werden. Bei den Assertions in NUNIT soll nicht mit Returncodes, sondern mit Natural-Fehlern gearbeitet werden, damit sie zu einem sofortigen Abbruch des Programms führen, wie es in anderen Programmiersprachen auch üblich ist.

```
DEFINE DATA
*
LOCAL
01 #TESTCASE-WAS-RUN (L)
*
END-DEFINE
*
#TESTCASE-WAS-RUN := FALSE
*
IF #TESTCASE-WAS-RUN
  *ERROR-NR := 1234
END-IF
*
CALLNAT 'TESTCASE' #TESTCASE-WAS-RUN
*
IF NOT #TESTCASE-WAS-RUN
  *ERROR-NR := 1234
END-IF
*
ON ERROR
  WRITE 'TestCase failed.'
  ESCAPE ROUTINE
END-ERROR
*
END
```

Listing 2.5: NUNIT testet automatisch die Ausführung eines TestCases

Test lauffähig machen Der Test ist bereits lauffähig und sogar erfolgreich. Aber um zu prüfen, ob die Fehlerbehandlung funktioniert, wenn er fehlschlägt, wird der Parameter in TESTCASE wieder auf `FALSE` gesetzt und der Test erneut durchgeführt. Und tatsächlich erscheint `TestCase failed.` auf dem Bildschirm. Nun kann der Parameter ruhigen Gewissens wieder auf `TRUE` gesetzt werden.

Refactoring Da der Test nun durchläuft, kann mit dem Refactoring begonnen werden. Als erstes fällt die *Magic Number* `1234` auf, die an zwei Stellen im Code als Fehlernummer für eine fehlgeschlagene Assertion verwendet wird. Um diese Redundanz zu beseitigen, wird sie als Konstante `ASSERTION-ERROR` in der *Local Data Area* (LDA) NUCONST definiert (Listing 2.6). Die beiden Assertions werden in eigene Subroutines (siehe Beispiel in Listing 2.7) ausgelagert, damit sie wiederverwendbar sind und der Code lesbarer wird.[7] Zuletzt wird dann noch die generische Fehlerbehandlung auf den `ASSERTION-ERROR` eingeschränkt. Das Ergebnis zeigt Listing 2.8.

[7] Die beiden Assertions ASSERT-TRUE und ASSERT-FALSE können nicht mit entsprechenden Tests abgedeckt werden, da sie selbst erst die grundlegende Funktionalität zur automatischen Prüfung von Ergebnissen bereitstellen. Ihre Logik ist allerdings auch so trivial, dass auf Tests verzichtet werden kann.

```
DEFINE DATA LOCAL
 1 ASSERTION-ERROR (N4,0) CONST
   <1234>
END-DEFINE
```

Listing 2.6: Die Konstante `ASSERTION-ERROR` in der LDA NUCONST

```
DEFINE DATA
*
PARAMETER
01 #ACTUAL (L)
*
LOCAL USING NUCONST
*
END-DEFINE
*
DEFINE SUBROUTINE ASSERT-TRUE
IF NOT #ACTUAL
  *ERROR-NR := ASSERTION-ERROR
END-IF
END-SUBROUTINE
*
END
```

Listing 2.7: Subroutine ASSERT-TRUE

2.3.3. Aufruf eines einzelnen Tests

Problem Wie ruft man gezielt einen einzelnen Test in einem TestCase auf?

Bei diesem Problem wurde vom Test-First-Ansatz abgewichen, da die Möglichkeiten zur Implementierung in Natural stark eingeschränkt sind und zunächst einige Prototypen ausprobiert werden mussten. Um den Rahmen der Arbeit nicht zu sprengen, werden die drei verschiedenen Möglichkeiten hier nur kurz vorgestellt. Der direkte Aufruf einer Subroutine in einem Subprogram von außen ist nicht möglich, da die internen Subroutinen nur im Scope des Subprograms bekannt sind. Ein Aufruf kann somit nur erfolgen, indem dem Subprogram der Name der gewünschten Subroutine als Parameter übergeben wird und dieses den Aufruf absetzt.

Leider ist der direkte Aufruf einer Subroutine mittels `PERFORM` anhand ihres Namens in einer Variablen nicht möglich, da `PERFORM` nur Konstanten als Parameter akzeptiert (Software AG 2009b). Die einzige Möglichkeit, eine Subroutine dynamisch aufzurufen, bietet das Statement `RUN`, das Quelltext zur Laufzeit übersetzt und ausführt. Allerdings wird dabei Naturals Programmstack gelöscht und das aufrufende und alle weiteren aktiven Programme werden beendet (Software AG 2009c). Daher müsste sich das aktive Programm vor jedem Aufruf von `RUN` selbst auf den Stack

```
DEFINE DATA
*
LOCAL USING NUCONST
*
LOCAL
01 #TESTCASE−WAS−RUN (L)
*
END−DEFINE
*
#TESTCASE−WAS−RUN := FALSE
*
PERFORM ASSERT−FALSE #TESTCASE−WAS−RUN
*
CALLNAT 'TESTCASE' #TESTCASE−WAS−RUN
*
PERFORM ASSERT−TRUE #TESTCASE−WAS−RUN
*
ON ERROR
  IF *ERROR−NR EQ ASSERTION−ERROR
    WRITE 'TestCase failed.'
    ESCAPE ROUTINE
  END−IF
END−ERROR
*
END
```

Listing 2.8: NUNIT nach dem ersten Refactoring

legen, um nach der Ausführung des TestCases erneut aufgerufen zu werden, wobei es dann natürlich irgendwie erkennen müsste, dass es jetzt *nach* dem TestCase zum zweiten Mal aufgerufen wird. Außerdem wird eine direkte Parameterübergabe unmöglich und alle Daten müssten in globalen Variablen gespeichert werden. Dieses Problem zöge sich dann durch die gesamte Anwendung, sodass ständig umständliche und schwer verständliche Stack-Operationen und Manipulationen an global gültigen Variablen durchzuführen wären. Ein weiteres (und größeres) Problem beim Einsatz des `RUN`-Statements wäre, dass die TestCases, die den variablen Codeanteil für den Aufruf der Subroutine enthalten, sich nicht mehr separat kompilieren lassen, da sie erst zur Laufzeit gültigen Code enthalten. Das wäre ein großer Nachteil, da Syntaxfehler erst zur Laufzeit erkannt und die Entwickler daher deutlich mehr Zeit für die Fehlersuche benötigen würden.

Die Alternative zum dynamischen Übersetzen des Quelltextes mit `RUN` wäre die direkte Entscheidung, welche Subroutine aufzurufen ist, in einem `IF`- oder `DECIDE`-Statement zu Beginn des TestCases, wie z. B. `IF #SUBROUTINE EQ 'TEST1' PERFORM TEST1`. Dieses müsste dann allerdings für jede neue Subroutine um einen entsprechenden Eintrag erweitert werden, was zu Redundanzen im Code führen und sich negativ auf die Akzeptanz bei den Entwicklern auswirken würde.

Nachdem sich beide Varianten als nicht praktikabel herausgestellt hatten und auch eine Anfrage in der Natural-Community[8] keine anderen Möglichkeiten zum dynamischen Aufruf von Subroutines aufzeigen konnte, wurde der Ansatz, die Tests in Subroutines zu implementieren, verworfen. Stattdessen wurde ein Parser implementiert, der zur Laufzeit den Quelltext der TestCases nach bestimmten Anweisungen durchsucht und daraus die Tests ableitet. Dabei werden die TestCases auch daraufhin überprüft, ob sie die von NUNIT benötigten Parameter akzeptieren und eine `SETUP`- und `TEARDOWN`-Routine anbieten. Dies ist ein wichtiger Schritt, da in Natural nicht durch native Sprachfeatures sichergestellt werden kann, dass ein Programm bestimmte Parameter akzeptiert oder Subroutines definiert.[9] Nachdem diese grundlegende Designentscheidung getroffen wurde, kann nun weiter testgetrieben entwickelt werden.

Im Folgenden werden nicht mehr alle Programme komplett im Quelltext dargestellt, sondern nur noch wichtige Implementierungen. Die gesamten Quelltexte befinden sich auf der beigelegten DVD im Ordner `Listings\NUNIT`.

Test In NUNIT wird dem TestCase ein zusätzlicher Parameter mit dem Bezeichner des auszuführenden Tests übergeben: `CALLNAT 'TESTCASE' 'TestCase was run' #TESTCASE-WAS-RUN`.

Test lauffähig machen TESTCASE benötigt diesen zusätzlichen Parameter und die Logik zur Ausführung des übergebenen Tests wird wie in Listing 2.9 gezeigt implementiert. Der Test läuft erfolgreich durch, kann aber durch Manipulation des String-Parameters `Test was run` in NUNIT zum Scheitern gebracht werden.

Refactoring Der Parameter, der den Bezeichner des Tests enthält, wird in die *Parameter Data Area* (PDA) NUTESTP (Listing 2.10) übertragen, da er von allen späteren TestCases verwendet wird. Der boolesche Parameter zur Prüfung, ob der Test ausgeführt wurde, wird ebenfalls in diese PDA verschoben und in `RESULT` umbenannt. Die Listings 2.11 und 2.12 zeigen TESTCASE bzw. NUNIT mit der eingebundenen PDA.

Was noch auffällt, ist das doppelte Auftreten des Test-Bezeichners `Test was run` in NUNIT und TESTCASE. Um dies zu verhindern, muss eine Möglichkeit geschaffen werden, die definierten Tests in TESTCASE dynamisch zu ermitteln. Dafür muss

[8] Vgl. http://tech.forums.softwareag.com/viewtopic.php?C=23,18,24,20,21&p=62957.
[9] Es fehlen die Möglichkeiten der Vererbung oder Definition von Interfaces. Bei JUnit erben die TestCases z. B. Methoden von der Klasse TestCase.

```
DEFINE DATA
*
PARAMETER
01 #TEST (A) DYNAMIC BY VALUE
01 #TESTCASE-WAS-RUN (L)
*
END-DEFINE
*
IF #TEST EQ 'TestCase was run'
  #TESTCASE-WAS-RUN := TRUE
END-IF
*
END
```

Listing 2.9: TestCase TESTCASE, der einen bestimmten Test gezielt ausführen kann

```
DEFINE DATA PARAMETER
 1 NUTESTP
   2 TEST (A) DYNAMIC
   2 RESULT (L)
END-DEFINE
```

Listing 2.10: PDA NUTESTP

der zu Beginn dieses Kapitels erwähnte Parser implementiert werden. Der Aufwand dafür ist aber nicht im Rahmen des Refactorings sinnvoll, sondern vielmehr als neue Anforderung.

2.3.4. Dynamische Ermittlung von definierten Tests in TestCases

Problem Wie können die in einem TestCase definierten Tests vom NUNIT-Framework dynamisch ermittelt und ausgeführt werden?

```
DEFINE DATA
*
PARAMETER USING NUTESTP
*
END-DEFINE
*
IF NUTESTP.TEST EQ 'TestCase was run'
  NUTESTP.RESULT := TRUE
END-IF
*
END
```

Listing 2.11: TestCase TESTCASE mit eingebundener PDA

```
DEFINE DATA
*
LOCAL USING NUCONST
LOCAL USING NUTESTP
*
END-DEFINE
*
PERFORM ASSERT-FALSE NUTESTP.RESULT
*
NUTESTP.TEST := 'TestCase was run'
CALLNAT 'TESTCASE' NUTESTP
*
PERFORM ASSERT-TRUE NUTESTP.RESULT
*
ON ERROR
  IF *ERROR-NR EQ ASSERTION-ERROR
    WRITE 'TestCase failed.'
    ESCAPE ROUTINE
  END-IF
END-ERROR
*
END
```

Listing 2.12: NUNIT mit eingebundener PDA

Test NUNIT (Listing 2.13) ermittelt die in TESTCASE definierten Tests durch einen Aufruf von NUPARSE und prüft die Anzahl der ermittelten Tests und den Namen des einzigen erwarteten Tests mittels ASSERT-NUM-EQUALS bzw. ASSERT-STRING-EQUALS. Dann werden alle Tests nacheinander durchgeführt.

Test lauffähig machen Weder das Subprogram NUPARSE noch die beiden neuen Assertions sind vorhanden. Diese werden direkt implementiert, während NUPARSE lediglich als leerer Stub erstellt wird, damit der Test fehlschlägt.

Test schlägt fehl Die einfachste Lösung, den Test erfolgreich durchlaufen zu lassen, ist die Implementierung von NUPARSE wie in Listing 2.14.

Refactoring Es stehen nun mehrere teils umfangreiche Refactorings an:

- NUPARSE muss korrekt implementiert werden. Es liest den Quelltext des Test-Cases aus und ermittelt zeilenweise anhand eines String-Vergleichs auf `IF NUTESTP.TEST EQ '...'` die definierten Tests.
- Die beiden neuen Assertions müssen mit Tests abgedeckt werden, da ihre Logik im Gegensatz zu ASSERT-TRUE und ASSERT-FALSE komplexer und damit

```
DEFINE DATA
*
LOCAL USING NUCONST
LOCAL USING NUTESTP
*
LOCAL
01 #TESTCASE (A8)
01 #TESTS (A/1:*) DYNAMIC
01 #NUM-TESTS (N2)
01 #T (N2)
*
END-DEFINE
*
#TESTCASE := 'NUTCTEST'
CALLNAT 'NUPARSE' #TESTCASE #TESTS(*)
#NUM-TESTS := *OCCURRENCE(#TESTS)
*
FOR #T 1 #NUM-TESTS
*
  RESET NUTESTP
  NUTESTP.TEST := #TESTS(#T)
  CALLNAT #TESTCASE NUTESTP
  WRITE 'Result for <' NUTESTP.TEST (AL=20) '>:' NUTESTP.RESULT
*
END-FOR
*
ON ERROR
  IF *ERROR-NR EQ ASSERTION-ERROR
    WRITE 'TestCase' #TESTCASE 'failed.'
    ESCAPE ROUTINE
  END-IF
END-ERROR
*
END
```

Listing 2.13: NUNIT ermittelt Tests dynamisch

möglicherweise fehlerhaft ist. Die Assertions wurden aber (bewusst) so entwickelt, dass sie Natural-Fehler erzeugen. Wenn ein solcher Fehler auftritt, ist es in Natural jedoch nicht möglich, diesen abzufangen und das Programm fortzusetzen. Das Programm bleibt im `ON ERROR`-Block und kann nur durch ein `ESCAPE` verlassen werden. Um nun die Ergebnisse mehrerer Assertions prüfen zu können, die ja durchaus auch einmal negativ ausfallen können, werden Wrapper-Routinen benötigt, die die Fehler abfangen und das Ergebnis in einen booleschen Wert umwandeln (Vgl. Listing 2.15), der dann mittels ASSERT-TRUE und ASSERT-FALSE automatisch geprüft werden kann. Ein Beispiel dafür ist in Listing 2.16 zu sehen.

- NUNIT muss aufgeteilt werden, da es zu viele Aufgaben hat: *echte* Tests durchführen und die eigene Funktionalität und die der anderen Komponenten von

```
DEFINE DATA
*
PARAMETER
01 #TESTCASE (A8)
01 #TESTS (A/1:*) DYNAMIC
*
END-DEFINE
*
RESIZE ARRAY #TESTS TO (1:1)
#TESTS(1) := 'TestCase was run'
*
END
```

Listing 2.14: Stub-Implementierung von NUPARSE

NUNIT testen. Als Ergebnis führt ab sofort NUNIT nur noch die echten TestCases durch, während TESTNU die NUNIT-internen Tests der Assertions und des Parsers durchführt (Selbsttest von NUNIT).

- Die Assertions müssen irgendwie voneinander unterschieden werden können, da im Moment nicht erkennbar ist, aufgrund welcher der verschiedenen Assertions die Tests fehlschlagen. Daher wird als einfachste Lösung zunächst in alle Assertions eine Bildschirmausgabe eingebaut, die den erwarteten und den tatsächlichen Wert anzeigt (Vgl. Listing 2.17).
- Redundanter Code (z. B. doppeltes Auftreten des TestCase-Bezeichners `TESTCASE` in NUNIT) muss eliminiert werden.
- TESTCASE wird in NUTCTEST umbenannt, um sich an die aktuelle Nomenklatur der internen Tests anzupassen: `NU` als Kürzel für NUNIT, `TC` als Kürzel für TestCase und die letzten vier Buchstaben für eine mehr oder weniger sprechende Beschreibung des Inhalts. Die vorhandenen Tests, die auf TESTCASE zeigten, müssen entsprechend angepasst werden.

Nach den Refactorings besteht NUNIT nun also aus den in Abbildung 2.4 gezeigten Komponenten.

2.3.5. SETUP und TEARDOWN ausführen

Problem Wie kann sichergestellt werden, dass die Subroutinen SETUP und TEARDOWN vor bzw. nach jedem Test ausgeführt werden?

```
1  *********************************************************************
2  DEFINE SUBROUTINE WRAP-ASSERT-NUM-EQUALS
3  *********************************************************************
4  *
5  #RESULT := TRUE
6  PERFORM ASSERT-NUM-EQUALS #EXPECTED #ACTUAL
7  *
8  ON ERROR
9    IF *ERROR-NR EQ ASSERTION-ERROR
10     #RESULT := FALSE
11     ESCAPE ROUTINE
12   END-IF
13 END-ERROR
14 *
15 END-SUBROUTINE
```

Listing 2.15: Beispiel einer Wrapper-Routine zur Umwandlung des Ergebnisses einer Assertion in einen booleschen Wert

```
1 DEFINE SUBROUTINE TEST-ASSERT-STRING-EQUALS
2 *
3 #STRING1 := 'test'
4 #STRING2 := 'test2'
5 *
6 PERFORM WRAP-ASSERT-STRING-EQUALS #STRING1 #STRING2 #RESULT
7 PERFORM ASSERT-FALSE #RESULT
8 *
9 END-SUBROUTINE
```

Listing 2.16: Beispiel für den automatischen Test des Ergebnisses einer Assertion durch den Aufruf ihrer Wrapper-Routine

Test Der Test wird in einem neuen Subprogram NUTCSEQ (Listing 2.18) erstellt, das in TESTNU zusätzlich zu den vorhandenen Selbsttests (für die Assertions und den Parser) aufgerufen wird. Er prüft den korrekten internen Ablauf anhand eines weiteren Parameters (das String-Array `#SEQUENCE` in dem der Ablauf im TestCase protokolliert werden soll.

```
1 DEFINE SUBROUTINE ASSERT-NUM-EQUALS
2 IF #ACTUAL NE #EXPECTED
3   WRITE '<' #ACTUAL '> should be <' #EXPECTED '>'
4   *ERROR-NR := ASSERTION-ERROR
5 END-IF
6 END-SUBROUTINE
```

Listing 2.17: Einfache Bildschirmausgabe einer Assertion im Fehlerfall

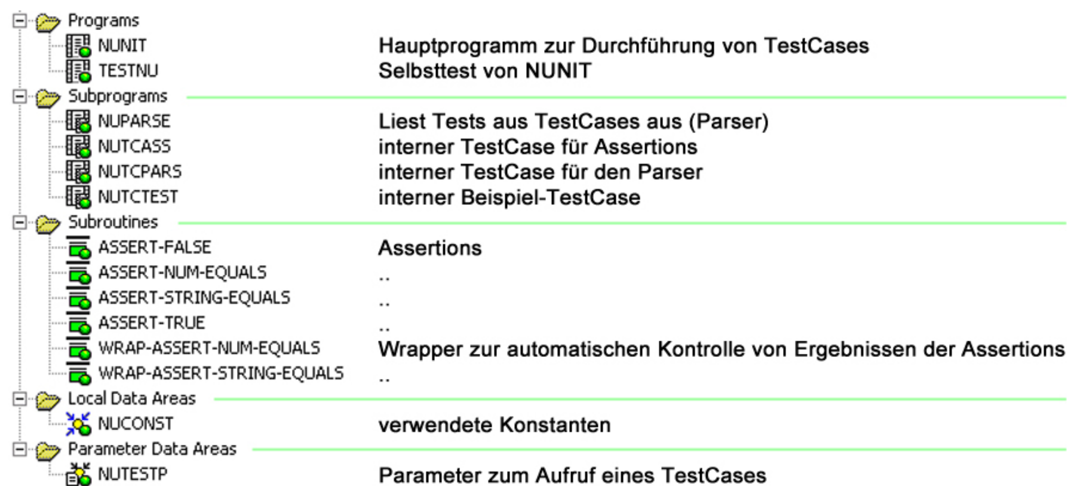

Abbildung 2.4.: Komponenten von NUNIT nach dem ersten größeren Refactoring

Test lauffähig machen NUTCTEST muss um den benötigten Parameter ergänzt werden. Dieser wird jedoch nur NUNIT-intern benötigt und nicht bei späteren normalen TestCases. Daher bricht nun der Aufruf von NUNIT ab, da dieser zusätzliche Parameter fehlt. Zur Lösung des Problems wird der TestCase kopiert und ohne zusätzlichen Parameter vorerst als TESTCASE abgelegt. Es wird somit ab sofort zwischen NUNIT-internen TestCases und Beispiel-TestCases für fachliche Tests unterschieden.

Test schlägt fehl Die einfachste Lösung, um den Test erfolgreich durchlaufen zu lassen, ist das Setzen der erwarteten Strings in #SEQUENCE in NUTCTEST (z. B. durch `#SEQUENCE(3) := 'TEARDOWN The first test'`).

Refactoring Nachdem der Test nun erfolgreich ist, wird die Logik in NUTCTEST korrekt implementiert. Die Subroutines SETUP und TEARDOWN werden angelegt und die eigentlichen Tests in die Subroutine TEST ausgelagert, um mittels eines *CopyCodes* die Einhaltung der Reihenfolge beim Ablauf des Tests gewährleisten zu können. Der CopyCode NUTCTEMP (Listing 2.19) dient somit als *Template Method* für den Ablauf eines Tests (erst das Setup, dann der Test und zuletzt das Teardown). Der fertige TestCase ist in Listing 2.20 zu sehen.

Einige Zeilen des TestCases sind besonders interessant, da sie den grundsätzlichen Aufbau der NUNIT-TestCases festlegen:

- Zeile 3: jeder TestCase muss die Parameter NUTESTP entgegennehmen

```
1  DEFINE DATA
2  *
3  LOCAL USING NUTESTP
4  *
5  LOCAL
6  01 #SEQUENCE (A/1:*) DYNAMIC
7  01 #NUM-STEPS (N2)
8  *
9  END-DEFINE
10 *
11 NUTESTP.TEST := 'The first test'
12 CALLNAT 'NUTCTEST' NUTESTP #SEQUENCE(*)
13 NUTESTP.TEST := 'The second test'
14 CALLNAT 'NUTCTEST' NUTESTP #SEQUENCE(*)
15 *
16 #NUM-STEPS := *OCCURRENCE(#SEQUENCE)
17 PERFORM ASSERT-NUM-EQUALS 6 #NUM-STEPS
18 *
19 PERFORM ASSERT-STRING-EQUALS 'SETUP The first test' #SEQUENCE(1)
20 PERFORM ASSERT-STRING-EQUALS 'TEST The first test' #SEQUENCE(2)
21 PERFORM ASSERT-STRING-EQUALS 'TEARDOWN The first test' #SEQUENCE(3)
22 PERFORM ASSERT-STRING-EQUALS 'SETUP The second test' #SEQUENCE(4)
23 PERFORM ASSERT-STRING-EQUALS 'TEST The second test' #SEQUENCE(5)
24 PERFORM ASSERT-STRING-EQUALS 'TEARDOWN The second test' #SEQUENCE(6)
25 *
26 END
```

Listing 2.18: Automatischer Test des Ablaufs eines Tests (NUTCSEQ)

```
1 PERFORM SETUP
2 PERFORM TEST
3 PERFORM TEARDOWN
```

Listing 2.19: *Template Method* zum Ablauf eines Tests (CopyCode NUTCTEMP)

- Zeile 14: jeder TestCase muss die *Template Method* inkludieren
- Zeilen 22, 27 und 32: jeder TestCase muss die Subroutinen SETUP, TEST und TEARDOWN implementieren
- Zeilen 37 und 41: jeder TestCase muss mindestens einen Test definieren

Wie bereits in Kapitel 2.3.3 erläutert, gibt es keine Möglichkeit, diese Voraussetzungen mit Mitteln der Programmiersprache zu gewährleisten. Daher muss der Parser um diese Funktionalität erweitert werden, damit nur gültige TestCases von NUNIT verarbeitet werden und der Entwickler nicht durch harte Natural-Fehler z. B. auf fehlende Subroutinen hingewiesen wird.

2.3.6. Weiterer Verlauf der Entwicklung

Durch die obige Beschreibung der ersten fünf Schritte bei der testgetriebenen Entwicklung von NUNIT dürfte das grundsätzliche Vorgehen beim TDD nun klar geworden sein. In Anbetracht des eingeschränkten Umfangs dieser Arbeit soll auf weitere detaillierte Ausführungen zum Fortgang der Entwicklung verzichtet werden. Schritt für Schritt wurden die folgenden Anforderungen an NUNIT in Form von Tests definiert, diese dann so schnell wie möglich zum Laufen gebracht und zuletzt korrekt implementiert:

- Wie kann sichergestellt werden, dass NUNIT nur gültige TestCases verarbeitet?
- Wie können mehrere TestCases hintereinander aufgerufen werden?
- Wie kann zwischen fehlgeschlagenen (Failure) und fehlerhaften (Error) TestCases unterschieden werden?
- Wie können die Ergebnisse mehrerer Tests und TestCases zentral gesammelt werden?
- Wie kann eine Statistik mit gelaufenen, fehlgeschlagenen und fehlerhaften Tests erstellt werden?
- Wie können die Entwickler ihre TestCases definieren und z. B. zu Suiten zusammenfassen?

Diese und noch weitere Anforderungen an NUNIT wurden im Rahmen dieser Arbeit testgetrieben umgesetzt. Das Endergebnis wird in Kapitel 2.4 in Form einer Architekturübersicht und Funktionsbeschreibung vorgestellt. In Kapitel 2.5 wird dann noch ein Fazit zum Einsatz von TDD bei der Natural-Entwicklung gezogen. Dabei werden mögliche Probleme und Erweiterungen von NUNIT und auch die ersten Beispiel-TestCases aus dem Produktivbetrieb vorgestellt.

2.4. Das fertige Framework

Abbildung A.1 auf Seite A6 im Anhang zeigt die Architektur des fertigen Unit-Test-Frameworks NUNIT. Im Vergleich zur Struktur von JUnit (Abbildung 2.1 auf Seite 8) wirkt sie auf den ersten Blick etwas umfangreicher, was aber hauptsächlich an den in das Diagramm integrierten *Parameter Data Areas*[10] (PDAs) liegt, die zum

[10] Siehe Kapitel A.1 im Anhang.

Verständnis sehr wichtig sind. Die Darstellungsform des Kommunikationsdiagramms der UML wurde etwas uminterpretiert, um die Natural-Module sinnvoll darstellen zu können:

- die *Objekte* sind Natural-Module, deren Typ jeweils durch die Stereotypen angegeben wird
- die durchgezogenen Pfeile beschreiben Programmaufrufe, der gestrichelte Pfeil steht für ein `INCLUDE`[11]
- die bei einem Programmaufruf übergebenen PDAs sind direkt als *Objekt* an den jeweiligen Aufruf-Pfeil angehängt
- zu den PDAs werden die enthaltenen Variablen angegeben, wobei Ein- und Ausgangsparameter über die Sichtbarkeitsmodifizierer unterschieden werden
- die drei `RESULT`-Konstanten sind in einer *Local Data Area* (LDA) definiert, die in jedem Programm eingebunden ist

Demnach beschränkt sich der Ablauf eines kompletten Testlaufs auf die folgenden Schritte:

1. NUNIT ermittelt die Namen der auszuführenden TestCases aus einer Textdatei und übergibt sie als Array an NURUNTS. Die Textdatei enthält lediglich eine Liste mit Bibliothek und Namen der TestCases in der Form `LIBNAME.TCNAME`.[12]
2. NURUNTS (**NU**NIT **run** **T**est**S**uite) führt dann alle TestCases der Reihe nach aus, indem es NURUNTC aufruft.
3. NURUNTC (**NU**NIT **run** **T**est**C**ase) überprüft zunächst mit Hilfe von NUPARSE, ob es sich um einen gültigen TestCase handelt und ruft ihn dann für jeden in ihm enthaltenen Test separat auf.
4. TESTCASE steht stellvertretend für den konkreten TestCase und inkludiert mit NUTCTEMP (**NU**NIT **T**est**C**ase **Temp**late) die *Template Method* und die Fehlerbehandlung für `ASSERTION-ERROR`s, die in den einzelnen Tests durch die ASSERT-*-Subroutinen beim Fehlschlagen einer Assertion provoziert werden.

Eine zentrale Rolle bei diesem Ansatz spielt der Parser. Natural bietet weder eine Form von Reflection zum Ermitteln der in einem Subprogram definierten Subroutinen an, noch eine Möglichkeit, um sicherzustellen, dass ein Subprogram eine bestimmte Subroutine definiert.[13] Dies ist aber wichtig, um die Tests je TestCase zu

[11] Siehe Kapitel A.1 im Anhang.
[12] Eine Beispieldatei ist in Listing A.1 auf Seite A5 im Anhang dargestellt.
[13] Vergleiche Kapitel 2.3.3.

ermitteln und den Ablauf SETUP → TEST → TEARDOWN zu gewährleisten. Diese Aufgaben übernimmt nun der Parser, der einfach die Quelltextzeilen der TestCases durchläuft und sie auf das Einhalten definierter Regeln prüft. Ein weiterer positiver Nebeneffekt des Parsers ist, dass TestCases, die als Subprograms der Beschränkung auf achtstellige Programmnamen unterliegen, und Tests, deren Namen bei Implementierung in Form von Subroutinen auf 32 Zeichen begrenzt wären, nun beliebig lange Freitext-Bezeichner verwenden können, da das Fixture und die Testbezeichner in Form von String-Variablen definiert werden (siehe Listing 2.21). Somit wäre sogar die Erstellung von sprechenden TestCases und Tests wie beim BDD möglich.

Abbildung 2.5 zeigt die Ausgabe des TestRunners während eines Testdurchlaufs. Analog zur Kommandozeilenversion von JUnit werden erfolgreiche Tests mit einem Punkt (`.`), fehlgeschlagene mit einem `F` und fehlerhafte mit einem `E` gekennzeichnet. Der so entstehende *Balken* färbt sich nach dem ersten Auftreten eines Fehlers auch tatsächlich rot.[14] Die Detailergebnisse der fehlgeschlagenen/fehlerhaften Tests werden im Anschluss in Abbildung 2.6 gezeigt. Dort sind auch die sprechenden Fixture- und Testbezeichner zu erkennen.

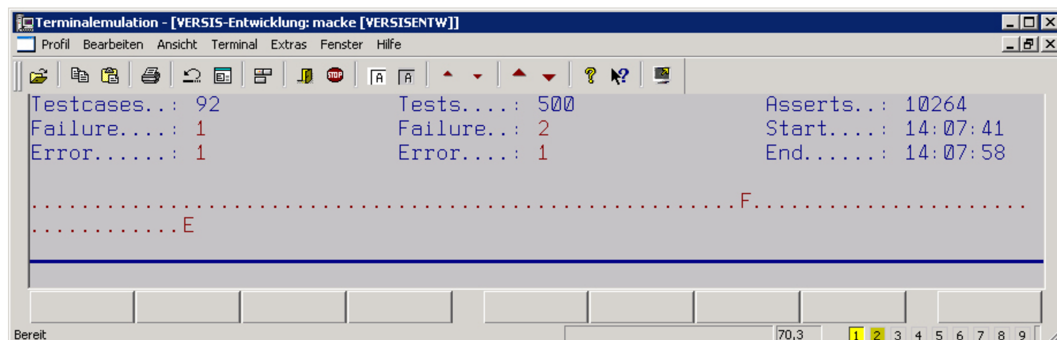

Abbildung 2.5.: Screenshot eines Testlaufs von NUNIT (roter Balken)

JUnit basiert auf vier (objektorientierten) Entwurfsmustern, die so nicht in Natural umgesetzt werden können. Es wurde jedoch versucht, ihre Ziele mit Natural-Mitteln zu erreichen:

- *Collecting Parameter*: Zwar wird zwischen den Programmen kein einzelnes Parameter-*Objekt* herumgereicht, aber durch den Einsatz der PDAs können die Ergebnisse der Tests zwischen den Programmen ausgetauscht und akkumuliert werden. Es wurde dabei jedoch bewusst auf den Einsatz einer einzigen großen PDA verzichtet. Nach dem Prinzip der kleinstmöglichen Schnittstellen (ISP) bekommt jedes Programm nur die absolut notwendigen Parameter übergeben.

[14] Auf der DVD ist dies im Video `Präsentationen\Abschlusspräsentation\NUNITTestRunner.avi` zu sehen.

```
Terminalemulation - [VERSIS-Entwicklung: macke [VERSISENTW]]
Profil  Bearbeiten  Ansicht  Terminal  Extras  Fenster  Hilfe
Seite     1                                                    09-08-30  14:08:23
                                 Fixture
TestCase TCEXMPL1: Example TestCase 1
  -> compare two different numbers should fail (Failure): TCEXMPL1 ( 470 ) fail
     ed: <2> should be <1>
TestCase TCEXMPL2: Example TestCase 2        Test
  -> compare to different strings should fail (Failure): TCEXMPL2 ( 570 ) faile
     d: <test2> should be <test1>.
  -> raises a workfile error (Error): TCEXMPL2 ( 630 ) error: 1599
Mehr
Bereit                                                   6,24
```

Abbildung 2.6.: Screenshot der Detailausgabe eines NUNIT-Testlaufs

Dadurch gibt es zwar mehrere PDAs, die Teile anderer PDAs enthalten, aber die Testbarkeit wird dadurch erhöht und die Programme können z. B. nicht Werte über ihrem Abstraktionsniveau manipulieren.

- *Template Method*: Die *Template Method* konnte recht gut durch den Einsatz des *CopyCodes* abgebildet werden, den alle TestCases inkludieren müssen (der Parser prüft dies). In der *Template Method* von JUnit wird auch die Fehlerbehandlung mittels `try/catch/finally` vorgenommen. Dieses Konzept ist in Natural jedoch nicht vorhanden. Die Fehler, die in den Assertions hervorgerufen werden, müssen aber im TestCase behandelt werden, da sie NUNIT sonst komplett abbrechen lassen. Daher wird die Fehlerbehandlung mittels `ON ERROR` im CopyCode vorgenommen.
- *Composite*: Zur Abbildung des Composite-Pattern benötigt man Interfaces, für die es in Natural keine Entsprechung gibt. Daher wurde das zusätzliche Subprogram NURUNTS nötig, dass TestSuites ausführt. Diese TestSuites werden in der Textdatei mit den TestCases definiert, indem sie einfach als String der Form `[Name der TestSuite]` vor die in ihr enthaltenen TestCases geschrieben werden.[15]
- *Command*: Auch das Command-Pattern verwendet Interfaces und konnte nicht abgebildet werden. Seine Aufgabe übernimmt der Parser, der die TestCases daraufhin überprüft, ob sie die benötigten Subroutines definieren.

Zum einfacheren Einsatz in der Praxis wurden zusätzlich zu NUNIT, das alle TestCases des aktuellen Benutzers ausführt, noch die beiden Programs NUSINGLE und NUSUITE erstellt, die einen einzelnen TestCase bzw. eine gesamte TestSuite ausführen.

[15] Listing A.1 auf Seite A5 im Anhang zeigt dies.

Außerdem wurde noch das Program NUREG erstellt, das zeitgesteuert alle definierten TestCases ausführt und die detaillierten Ergebnisse per Mail (siehe NUMAIL) an die Entwickler schickt. Damit wird die Möglichkeit für nächtliche Regressionstests geschaffen.

Um NUNIT auch auf andere Umgebungen portieren zu könne, wurden die Funktionen, die auf unterschiedlichen Plattform anders funktionieren, bewusst in kleine Subroutines ausgelagert, damit sie einfach ausgetauscht werden können. Es handelt sich hierbei um GET-TESTCASES und GET-TESTCASE-SOURCE. Je nach Umgebung[16] werden nämlich die Quelltexte der Programme und die Liste der TestCases entweder in einer Datenbank oder im Dateisystem gespeichert.

2.5. Nachbetrachtung und Ausblick

Nach der Fertigstellung von NUNIT wurden bereits einige TestCases erzeugt, die echte Module in VERSIS testen. Zwei Beispiele wurden bereits vorgestellt und sind auf der dieser Arbeit beiliegenden DVD enthalten[17]:

- TCGETPER testet das bereits vorhandene Subprogram GETPERSO, das anhand einer Vertrags- oder Partnernummer eine Person, ihr Geschlecht und Alter, sowie ihr Geburtsdatum ermittelt. Dieser Test prüft die Ergebnisse einer Datenbankabfrage in GETPERSO. Dies sollte noch entkoppelt werden, was jedoch nicht so einfach ist, da die vorhandene Anwendung direkt mit der Datenbank kommuniziert.
- Der TestCase TCVUMBU wurde (im Gegensatz zu TCGETPER, der ein bereits vorhandenes Subprogram überprüft) zur testgetriebenen Entwicklung eines neuen Moduls verwendet. VUMBUCHN führt abhängig von den Salden der vier übergebenen Konten Umbuchungen zwischen diesen durch, um negative Salden auszugleichen, wobei bestimmte Konten Vorrang vor anderen haben. In TCVUMBU wurde diese Logik zunächst in Form von Tests definiert, woraufhin nur noch ein Programm implementiert werden musste, das diese Tests erfolgreich besteht. Dieses Vorgehen hat dank des NUNIT-Frameworks auch sehr gut funktioniert und im Laufe der Entwicklung wurden noch weitere Tests (z. B. für bestimmte Sonderfälle) hinzugefügt.

[16] Großrechner, Unix oder Windows.
[17] Sie liegen im Verzeichnis `Listings\NUNIT\Finalversion`. Aufgrund ihres großen Umfangs wurden sie nicht mit in den Text der Arbeit aufgenommen.

Beim Praxiseinsatz erwies sich NUNIT als sehr performant und leicht zu bedienen. Wo vorher simple Aufrufprogramme mit rudimentärer Logik zum Abbilden von Tests für neue Module (etwa mittels Bildschirmausgaben) erstellt wurden, können nun *echte* TestCases erzeugt werden, die sich mittels NUNIT selbst prüfen und wiederholt in nächtlichen Regressionstests durchgeführt werden.

Es muss jedoch ganz klar gesagt werden, dass NUNIT wohl nur für Neuentwicklungen oder bei eingegrenzten Refactorings eingesetzt werden wird. Erste Versuche, Tests für vorhandene Programme zu erstellen (siehe GETPERSO), scheiterten meist an den verwobenen Abhängigkeiten der vorhandenen Programme untereinander und insbesondere von der Datenbank. Hier muss noch eine Lösung gefunden werden, um einfach von der Datenbanklogik abstrahieren zu können.

In Kapitel 2.3 wird das grundsätzliche Vorgehen von TDD gut sichtbar: Schrittweise formt sich eine wachsende Applikation, wobei durch den Fokus auf jeweils eine einzige Anforderung die Iterationen klein und überschaubar bleiben. Die Tests geben dann die nötige Sicherheit, um den nächsten Schritt zu tun. Allerdings wird auch bestätigt, dass selbst beim Einsatz von TDD ein grundsätzliches Design der Anwendung unumgänglich ist. Es muss zumindest eine grobe Richtung vorgegeben werden, damit man bei der Entwicklung nicht in einer Sackgasse landet, aus der man auch mit noch so vielen Tests nicht mehr herausfindet. Ohne die schnell und ohne Testabdeckung erstellten Prototypen wären die in Kapitel 2.3.3 beschriebenen Defizite in Natural viel zu spät entdeckt worden. Die letztendliche Lösung konnte nur durch Ausprobieren und Verwerfen der möglichen Alternativen erkannt werden. Nichtsdestotrotz konnte nach dieser Grundsatzentscheidung tatsächlich testgetrieben (weiter-)entwickelt werden.

Trotz der anfänglichen Skepsis gegenüber den technischen Möglichkeiten von Natural zur Umsetzung eines Unit-Tests-Frameworks, konnte letztlich ein solides Framework erstellt werden, dass (fast) alle Funktionen seiner objektorientierten Vorgänger abdeckt. Das Hauptproblem ist die fehlende Unterstützung für Interfaces und Reflection, die für weiterführende Konzepte wie z. B. Stubs und Mocks benötigt werden. Die Kernanwendungen des Frameworks können durch den eigenen Parser jedoch sehr gut diesen Nachteil kompensieren und mittels einer *Registry*[18] lassen sich wahrscheinlich auch Stubs implementieren. Eine Funktion, die sich in der aktuellen Architektur definitiv nicht implementieren lässt, ist das Ausführen von separaten SETUP- und TEARDOWN-Routinen vor bzw. nach einem kompletten TestCase (`@BeforeClass` und `@AfterClass` in JUnit), da die Variablen innerhalb der TestCases (Subprograms) beim Verlassen (nach jedem einzelnen Test) gelöscht werden. Allerdings weist

[18]Vergleiche Kapitel 2.5.2.

der Einsatz dieser *globalen* Routinen laut Stal (2005, S. 116) ohnehin auf Lücken in der Implementierung hin.

In den folgenden zwei Kapiteln werden nun noch kurz die bekannten Probleme mit NUNIT erläutert und mögliche Erweiterungen des Frameworks aufgezeigt.

2.5.1. Offene Probleme

Das Hauptproblem beim Einsatz von NUNIT bei der Pflege von VERSIS ist die enorme Menge an (nicht-testbarem) Legacy Code. Selbst wenn der Einsatz von Tests auf die *Weiter*entwicklung von VERSIS eingeschränkt wird, weisen die vorhandenen Programme, die als Abhängigkeiten berücksichtigt werden müssen, eine starke Kopplung untereinander und zur Datenbank auf. Daher ist es sehr aufwändig, auch nur Teile der Anwendung mit Tests zu versehen bzw. *gute* Tests für sie zu entwickeln. Auch werden in VERSIS viele Programs verwendet, die sich mit NUNIT aufgrund der fehlenden Möglichkeit, ihnen Parameter zu übergeben, nicht testen lassen. Und zuletzt basiert der Großteil der Funktionen von VERSIS auf Logik, die direkt an Bildschirmmasken gekoppelt ist, die sich ebenfalls nicht mit NUNIT testen lassen. Es ist also eine große Herausforderung, bei der Weiterentwicklung von VERSIS testgetrieben vorzugehen und nicht durch den voraussichtlich sehr hohen Zusatzaufwand für das Refactoring der vorhandenen Programme abgeschreckt zu werden.

Genau dies führt zum zweiten Problem: Wie können die Natural-Entwickler, für die automatische Tests und damit ein testbarer Entwurf bisher keine Relevanz hatten, davon überzeugt werden, NUNIT zu nutzen? Wenn selbst für einfachste Test zunächst aufwändige (und aufgrund fehlender Tests fehleranfällige) Refactorings durchzuführen sind, wird die Akzeptanz von NUNIT gegen Null gehen. Beck und Gamma (2000) haben hierfür eine interessante Lösung vorgeschlagen, die sie in Analogie zum Test-infizierten Entwickler „spread the infection“ nennen: Wenn ein Kollege eine Frage hat, wie er eine bestimmte Anforderung am besten implementieren könnte, sollte man ihn einfach bitten zu erklären, woran er erkennen würde, dass diese Anforderung umgesetzt ist. Dann sollte man dies in Form eines fehlschlagenden Tests aufschreiben und ihm diesen überlassen, damit er die entsprechende Implementierung durchführen kann. Nach einigen Durchläufen dieses Vorgehens wird der Kollege dann selbst anfangen, Tests zu schreiben, weil er die Vorteile dieses Vorgehens erkennt. Eine große Hilfe für die Natural-Entwickler wäre dabei sicherlich eine Art *Coding Guide* für NUNIT, in dem bestimmte Regeln für gute Tests festgelegt werden, z. B. in Analogie zu den möglichst kurzen Klassen und Methoden bei der

Objektorientierung, möglichst kurze Programs und Subprograms in Natural. Grundsätzlich folgen die Natural-Entwickler bereits den allgemein gültigen Prinzipien der Programmierung, wie der Modularisierung oder dem Fokus auf die Wiederverwendbarkeit. Jetzt müssten sie nur noch von den Vorteilen automatischer Tests überzeugt werden, die das Vorhandensein dieser Prinzipien in ihren Programmen direkt nachweisen und Schwachstellen aufzeigen.

2.5.2. Erweiterungsmöglichkeiten

Obwohl die Kernfunktionalitäten eines Unit-Test-Frameworks von NUNIT abgedeckt werden, gibt es noch einige mögliche Erweiterungen, die kurz genannt werden sollen, aber im Rahmen dieser Arbeit nicht mehr implementiert werden konnten.

- Die Assertions sollten erweitert werden. Es wären z. B. eine Reihe weiterer einfacher Vergleiche wie ARRAY-CONTAINS, aber auch komplexerer VERSIS-spezifischer Prüfungen wie das Vorhandensein einer speziellen Tarifkombination denkbar.
- Die TestSuites sollten kaskadierbar sein. Dies könnte z. B. durch Einträge der Form `SUITE:NameDerSuite` in der TestCase-Datei erreicht werden.
- Darüber hinaus gibt es noch weitere Optimierungen der TestCase-Verwaltung, wie z. B. das Einbinden externer Textdateien der Form `FILE:NameDerDatei.txt`.
- Dann wäre noch die Möglichkeit zur Erstellung von Stubs und Mocks sinnvoll, um die vielen Abhängigkeiten aufzulösen. Dies wäre jedoch ein größerer Aufwand, da wie gesagt keine Interfaces verwendet werden können. Es könnte aber z. B. eine *Registry* verwendet werden, bei der Programme für eine gewünschte Funktion registriert werden können, und die von den zu testenden Programmen verwendet wird, um das aktuell für die benötigte Funktionalität registrierte Programm zu ermitteln. Dadurch könnte im TestCase für diese Funktionalität in der Registry ein Stub-Programm registriert werden, das dann anstelle des echten Programms im zu testenden Programm aufgerufen wird. Damit läge die Verantwortung zur Ermittlung des zu nutzenden Programms jedoch im Gegensatz zur *Dependency Injection* in den zu testenden Programmen und diese wären alle abhängig von der Registry, was sich schnell als Flaschenhals und *Single Point of Failure* herausstellen könnte. Eine sinnvolle Lösung muss also noch im Detail erarbeitet werden.

```
1  DEFINE DATA
2  *
3  PARAMETER USING NUTESTP
4  *
5  PARAMETER
6  01 #SEQUENCE (A/1:*) DYNAMIC
7  *
8  LOCAL
9  01 #S (N2)
10 01 #STEP (A) DYNAMIC
11 *
12 END-DEFINE
13 *
14 INCLUDE NUTCTEMP
15 *
16 DEFINE SUBROUTINE LOG-STEP
17 #S := *OCCURRENCE(#SEQUENCE) + 1
18 RESIZE ARRAY #SEQUENCE TO (1:#S)
19 #SEQUENCE(#S) := #STEP
20 END-SUBROUTINE
21 *
22 DEFINE SUBROUTINE SETUP
23 COMPRESS *CURRENT-UNIT NUTESTP.TEST INTO #STEP
24 PERFORM LOG-STEP
25 END-SUBROUTINE
26 *
27 DEFINE SUBROUTINE TEARDOWN
28 COMPRESS *CURRENT-UNIT NUTESTP.TEST INTO #STEP
29 PERFORM LOG-STEP
30 END-SUBROUTINE
31 *
32 DEFINE SUBROUTINE TEST
33 *
34 COMPRESS *CURRENT-UNIT NUTESTP.TEST INTO #STEP
35 PERFORM LOG-STEP
36 *
37 IF NUTESTP.TEST EQ 'The first test'
38   NUTESTP.RESULT := TRUE
39 END-IF
40 *
41 IF NUTESTP.TEST EQ 'The second test'
42   NUTESTP.RESULT := FALSE
43 END-IF
44 *
45 END-SUBROUTINE
46 *
47 END
```

Listing 2.20: TestCase NUTCTEST inkl. Setup, Teardown und Protokollierung seines Ablaufs

```
DEFINE DATA
PARAMETER USING NUTESTP /* parameters for a single test
LOCAL USING NUCONST /* NUNIT constants (error code etc.)
LOCAL USING NUASSP /* parameters for the assertions
END-DEFINE
*
NUTESTP.FIXTURE := 'Example TestCase 1' /* this TestCase's fixture
*
INCLUDE NUTCTEMP /* the template method (SETUP -> TEST -> TEARDOWN)
INCLUDE NUTCSTUB /* stub implementations of SETUP and TEARDOWN
*
DEFINE SUBROUTINE TEST /* the main test routine
*
***********************************************************
IF NUTESTP.TEST EQ 'comparing two equal numbers should pass' /* name of first test
***********************************************************
  ASSERT-LINE := *LINE; PERFORM ASSERT-NUM-EQUALS NUASSP 2 2
END-IF
*
***********************************************************
IF NUTESTP.TEST EQ 'comparing two different numbers should fail' /* name of second test
***********************************************************
  ASSERT-LINE := *LINE; PERFORM ASSERT-NUM-EQUALS NUASSP 1 2
END-IF
*
END-SUBROUTINE
*
END
```

Listing 2.21: Beispiel für einen NUNIT-TestCase

Literaturverzeichnis

Ambler 2009

AMBLER, Scott W.: *Introduction to Test Driven Design (TDD)*. Version: Juli 2009. http://www.agiledata.org/essays/tdd.html, Abruf: 03.07.2009 2.1

Banks 2008

BANKS, Richard: *A Quick Introduction to Test Driven Development in C#*. Screencast. http://www.youtube.com/watch?v=f60aIlNhMoE. Version: Januar 2008 2.1.1.3

Beck 2003

BECK, Kent: *Test-Driven Development: by Example*. Boston : Addison-Wesley, 2003 http://www.amazon.de/o/ASIN/0321146530. – ISBN 0–321–14653–0 2.1.1.1, 2.1.1.2

Beck und Gamma 2000

Kapitel 9. In: BECK, Kent ; GAMMA, Erich: *Test-infected: programmers love writing tests*. New York, NY, USA : Cambridge University Press, 2000. – ISBN 0–521–77477–2, 357–376 1, 2.5.1

Coldewey 2009

COLDEWEY, Jens: Schlechte Noten für Informatik-Ausbildung. In: *objektSpektrum* (2009), September, Nr. 5, S. 12–15 1

Feathers und Freeman 2009

FEATHERS, Michael ; FREEMAN, Steve: *Test Driven Development: Ten Years Later*. Videoaufzeichnung eines Vortrags auf der QCon. http://www.infoq.com/presentations/tdd-ten-years-later. Version: August 2009 1

Foley und Murphy 2002

FOLEY, John ; MURPHY, Chris: Q&A: Bill Gates On Trustworthy Computing. In: *InformationWeek* (2002), Mai. http://www.informationweek.com/showArticle.jhtml?articleID=6502378 1

Gamma 2006

GAMMA, Erich: *JUnit: A Cook's Tour*. Version: März 2006. http://

junit.sourceforge.net/doc/cookstour/cookstour.htm, Abruf: 05.07.2009 2.1, 2.1.2

Jeffries 2009
Jeffries, Ron: *Software for Unit Testing.* Version: 2009. http://xprogramming.com/software, Abruf: 11.08.2009 2

Kronauer 2009
Kronauer, Karlheinz: *Natural Integrated Development Environment.* Vortrag auf der International User Group Conference der Software AG, Juni 2009 2

Martin 2009
Martin, Robert C.: *The Principles of OOD.* Version: Mai 2009. http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, Abruf: 26.05.2009 (document)

Osherove 2008
Osherove, Roy: *The Art of Unit Testing.* Manning Publications, 2008 http://www.amazon.de/o/ASIN/1933988274. – ISBN 1–933–98827–4 2.1.1.3

Paulisch u. a. 2009
Paulisch, Frances ; Stal, Michael ; Zimmerer, Peter: Software-Curriculum: Architektenausbildung bei Siemens. In: *objektSpektrum* 4 (2009), Juli, S. 20–25 1

Software AG 2009a
Software AG: *Natural for UNIX Documentation.* Version: 2009. http://documentation.softwareag.com/natural/nat636unx/overview.htm, Abruf: 22.08.2009 2, 2.3.2

Software AG 2009b
Software AG: *Natural for UNIX Documentation: Statement PERFORM.* Version: 2009. http://documentation.softwareag.com/natural/nat636unx/sm/perform.htm, Abruf: 22.08.2009 2.3.3

Software AG 2009c
Software AG: *Natural for UNIX Documentation: Statement RUN.* Version: 2009. http://documentation.softwareag.com/natural/nat636unx/sm/run.htm, Abruf: 22.08.2009 2.3.3

Stal 2005
Stal, Michael: Design your Test – Unit-Tests mit dem .Net-Framework. In:

iX 7 (2005), 114–118. http://www.heise.de/kiosk/archiv/ix/2005/7/114_kiosk 1, 2.1.1.1, 2.1.1.3, 2.5

Weinberg 2008

WEINBERG, Gerald M.: *How We Used to Do Unit Testing*. Version: Dezember 2008. http://secretsofconsulting.blogspot.com/2008/12/how-we-used-to-do-unit-testing.html, Abruf: 03.09.2009 1

Zimmermann 2009

ZIMMERMANN, Torsten: *Testen kostet – nicht zu testen auch!* Version: Juni 2009. http://www.business-wissen.de/qualitaet/fehleranalyse/fachartikel/software-qualitaet-testen-kostet-nicht-zu-testen-auch.html, Abruf: 15.06.2009 1

A. Anhang

A.1. Einführung in Natural

Natural ist als 4GL-Sprache darauf ausgelegt, möglichst lesbare Quelltexte erzeugen zu können. Die Sprache bietet viele Statements an, die auf einem sehr hohen Abstraktionsniveau liegen. So kann z. B. der Zugriff auf eine Datenbanktabelle mit einem einzigen Befehl (`READ Tabellenname`) realisiert werden. Die fachliche Geschäftslogik eines Programms lässt sich mit Natural somit sehr gut abbilden, aber sobald die Anforderungen ein wenig technischer werden, lassen sie sich nur sehr umständlich umsetzen. So fehlen z. B. Statements für rudimentäre Aufgaben wie String-Manipulationen oder Bit-Operationen, die für Entwickler anderer Programmiersprachen zum Standardrepertoire zählen.

Natural wird inzwischen für verschiedene Plattformen angeboten, sein Ursprung liegt jedoch im Bereich der Großrechner. Natural arbeitet üblicherweise mit der Datenbank Adabas der Software AG zusammen, in der z. B. auch die Quelltexte der Natural-Programme gespeichert werden können. Bei der Alte Oldenburger läuft Natural auf einem UNIX-Server (Sun Solaris) und die Programme werden als Textdateien im Dateisystem abgelegt. Für die Programmnamen gilt eine Beschränkung der Länge auf acht Zeichen, was nicht erst bei einer großen Anzahl an Modulen zu kryptischen Bezeichnern führt. Die Programme können in Bibliotheken (Libraries) organisiert werden, deren Namen ebenfalls auf acht Zeichen beschränkt sind.

Grundsätzlich gibt es drei verschiedene Typen von Programmen[19]: Programs[20], Subprograms und Subroutines. Die Unterschiede zeigt Tabelle A.1.

Programmtyp	Beschreibung
Program	Programs sind als einzige Programmtypen selbstständig ausführbar und dürfen daher keine Parameter verwenden. Sie werden von anderen Programmen mittels `FETCH` aufgerufen.
Subprogram	Subprograms machen den Großteil der meisten Anwendungen aus. Sie sind nicht selbstständig ausführbar und müssen daher von anderen Programmen mittels `CALLNAT` aufgerufen werden. Sie können Parameter übergeben bekommen und zurückliefern.

[19] Es gibt noch weitere Typen wie *Function* oder *Class*, die jedoch für die weiteren Ausführungen nicht relevant sind und in der Praxis nur sehr selten verwendet werden.

[20] Im Folgenden bezeichnet der Begriff *Programme* allgemein die drei Programmtypen, während mit *Programs* nur der Programmtyp Program gemeint ist.

Subroutine	Subroutines sind die kleinsten Programmtypen. Sie können *inline* in Programs und Subprograms definiert werden und ihre Namen sind im Gegensatz zu den üblichen 8 auf 32 Zeichen beschränkt. Zusätzlich können sie auch in eigenständigen *externen* Modulen definiert werden. Sie sind nicht selbstständig ausführbar und müssen mittels `PERFORM` aufgerufen werden. Sie können Parameter übergeben bekommen und zurückliefern.
CopyCode	CopyCodes sind Quelltextausschnitte, die mittels des Statements `INCLUDE` in ein Natural-Programm eingefügt werden können. Sie sind nicht kompilierbar und werden einfach als Text 1-zu-1 in das Programm eingefügt, das dann jedoch inkl. der Inhalte des CopyCodes kompilierbar sein muss.

Tabelle A.1.: Mögliche Programmtypen in Natural

Die Variablen in Natural haben grundsätzlich einen Programm-globalen Gültigkeitsbereich, und können daher auch nur im Kopf eines Programms (im `DEFINE DATA`-Bereich) deklariert werden. Abhängig vom Programmtyp lässt sich aber auch noch ein programmübergreifender Gültigkeitsbereich definieren, wie in Tabelle A.2 beschrieben.

Bereich	**Beschreibung**
`GLOBAL`	Globale Variablen sind automatisch in allen Programmen verfügbar, in denen sie definiert sind.
`LOCAL`	Lokale Variablen haben einen auf das aktuelle Programm beschränkten Gültigkeitsbereich. Sie können jedoch als Parameter einem anderen Programm übergeben werden.
`PARAMETER`	Parameter werden in Subprograms und Subroutinen verwendet, um die benötigten Variablen für ihren Aufruf zu definieren. Sie können im Gegensatz zu den anderen beiden Typen keine Konstanten enthalten.

Tabelle A.2.: Mögliche Gültigkeitsbereiche von Variablen in Natural

Die Variablen werden entweder direkt im Programm definiert, oder in separaten Modulen (*Data Areas*), die dann in mehreren Programmen verwendet werden können. Dabei werden immer ihr Datentyp und ihre Länge angegeben. Die wichtigsten Da-

tentypen sind: `A` alphanumerisch, `N` numerisch und `L` boolean. Auch Arrays können erzeugt werden, wie das folgende Beispiel zeigt:

```
01 #ZAHLEN (N6,2/1:10)
```

Die `01` steht für die Ebene der Variablen, da man diese auch schachteln und redefinieren kann, um z. B. auf die ersten 3 Stellen einer Zahl über eine separate Variable zugreifen zu können. Die Variablenbezeichner beginnen üblicherweise mit einem `#`. Darauf folgt dann der Datentyp und die Länge (`N6,2`) und die Kennzeichnung als Array mit zehn Elementen `/1:10`.

A.2. Listings und Abbildungen

```
[Suite 1]
LIB1.TCTEST1
LIB2.TCTEST2

[Suite 2]
LIB1.TCTEST3
LIB2.TCTEST4
```

Listing A.1: Workfile mit NUNIT-TestCases eines Benutzers

«Program»
NUNIT

1.

2.

«Parameter»
NUGETTCP
- USERNAME: A8
- LIBRARY: A8
+ MESSAGE: A
+ TESTCASES: A8/1:*

Parameters:
- IN
+ OUT

«Parameter»
NUTSP
- LIBRARY: A8
- TESTCASES: A8/1:*
+ SUITE-RESULT: RESULT
+ RESULTS: 1:*
+ TC-RESULT: RESULT
+ TC-MESSAGE: A
+ FIXTURE: A
+ TESTS: 1:*
+ TEST: A
+ NUM-ASSERTS: N8
+ TEST-RESULT: RESULT
+ TEST-MESSAGE: A
+ STARTTIME: T
+ ENDTIME: T

«Constants»
RESULT
- SUCCESS
- FAILURE
- ERROR

Platform specific

«Subroutine»
GET-TESTCASES

reads TestCases from workfile or DB

from file system or DB

«Subroutine»
GET-TESTCASE-SOURCE

«Subprogram»
NURUNTS

for each TESTCASE

«Parameter»
NUTCP
- LIBRARY: A8
- TESTCASE: A8
+ TC-RESULT: RESULT
+ TC-MESSAGE: A
+ FIXTURE: A
+ TESTS: 1:*
+ TEST: A
+ NUM-ASSERTS: N8
+ TEST-RESULT: RESULT
+ TEST-MESSAGE: A

Collecting Parameter(s)

«Parameter»
NUTCSRCP
- LIBRARY: A8
- TESTCASE: A8
+ SOURCE-LINES: A/1:*

«Subprogram»
NURUNTC

1.

2.

«Subprogram»
NUPARSE

«Parameter»
NUPARSEP
- LIBRARY: A8
- TESTCASE: A8
- SOURCE-LINES: A/1:* OPTIONAL
- VALID-TESTCASE: L
+ MESSAGE: A
+ FIXTURE: A
+ TESTS: A/1:*

- validates TESTCASE
- reads TESTS

for each TEST

«Parameter»
NUTESTP
- FIXTURE: A
- TEST: A
+ NUM-ASSERTS: N8
+ RESULT: RESULT
+ MESSAGE: A

«Subprogram»
TESTCASE

«CopyCode»
NUTCTEMP

- Template Method
- Subroutine LOG
- ON ERROR for ASSERTION-ERROR

«Parameter»
NUASSP
- MESSAGE: A
+ NUM-ASSERTS: N8

- EXPECTED
- ACTUAL

- NUM-ASSERTS++
- *ERROR-NR := ASSERTION-ERROR

«Subroutine»
ASSERT...

Abbildung A.1.: Architektur von NUNIT