Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeitarbeit

André Jeworutzki

Unit-Test mit XML

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

André Jeworutzki

Unit-Test mit XML

Bachelorarbeitarbeit eingereicht im Rahmen der Bachelorarbeitprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Jörg Raasch
Zweitgutachter : Prof. Dr. Olaf Zukunft

Abgegeben am 6. Februar 2008

André Jeworutzki

Thema der Bachelorarbeit

Unit-Test mit XML

Stichworte

Unit-Test, Regressionstest, Automatisierung, Vergleichsverfahren, Ähnlichkeitsbestimmung, XML

Kurzzusammenfassung

Diese Arbeit handelt über die Entwicklung eines Prototyps, der den Aufwand für Unit-Tests durch Automatisierung minimiert. Die Idee ist es, Vergleichsfunktionen automatisch zu erzeugen, um sie anschließend im Unit-Test zu verwenden. Hierzu sind Verfahren notwendig, die selbstständig ähnliche Objekte erkennen und vergleichen. Darüber hinaus werden Testdaten für Regressionstest automatisch verwaltet. Der Prototyp verwendet XML, um die Testdaten zu repräsentieren. Weiterhin sind Architektur und Konfiguration darauf zugeschnitten, Unit-Tests effizient durchzuführen.

André Jeworutzki

Title of the paper

Unit-Test with XML

Keywords

Unit testing, Regression testing, Automation, Comparison, Similarity, XML

Abstract

This paper is about the development of a prototype, that minimizes the effort of unit testing by automation. The idea is to generate comparison functions automatically in order to use them in a unit test. This requires an algorithm, which independently recognizes and compares similar objects. Moreover test data are managed automatically for regression testing. The prototype uses XML, to represent the test data. Finally architecture and configuration is adjusted to accomplish more efficient unit tests.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einführung

“Softwarefehler (Englisch: “bugs”) findet man immer dann, wenn man sie am wenigsten braucht.” - das Zitat eines frustrierten Anwenders, der unfreiwillig für die Qualitätssicherung eingeteilt wurde. Dabei hat sich seit den Anfängen der Informatik herausgestellt, dass Softwaretests notwendig sind. Und obwohl Softwaretests zusätzlichen Aufwand und damit Kosten bedeuten, zahlen sie sich spätestens dann aus, wenn die ersten Softwarefehler frühzeitig gefunden werden. Jedoch wird oftmals aus Zeitmangel die Weiterentwicklung gegenüber dem Schreiben von Softwaretests vorgezogen. Schuld daran ist der hohe Aufwand, um Softwaretests zu schreiben. Einer ungeschriebenen Daumenregel zufolge ist das Verhältnis zwischen Anwendungscode und Testcode mittlerweile 50 zu 50. Aus diesem Grund existieren viele Werkzeuge auf dem Markt, die das Testen erleichtern und automatisieren. Hierzu gehört der Unit-Test: die Grundausrüstung eines jeden Softwaresoldaten im Kampf für die Fehlerfreiheit. Allerdings beschränkt sich der Unit-Test darauf, Softwaretests zu automatisieren, nicht sie zu schreiben. Genau diese Lücke schließt der hier in Java entwickelte Prototyp, der unter Verwendung von XML den Aufwand für Softwaretests minimiert, indem er Vergleichsfunktionen automatisch erzeugt und Regressionstests vereinfacht.

1.1. Umfeld

Diese Bachelorarbeit und der entwickelte Prototyp sind im Rahmen der Arbeiten bei der Versicherungs-AG Deutscher Ring im Auftrag der C1 WPS GmbH entstanden. Entwicklung, Gespräche und Auswertung fanden vor Ort beim Deutschen Ring statt.

1.2. Aufbau

Die Bachelorarbeit ist in sechs Kapitel unterteilt, wobei das erste Kapitel diese Einleitung ist. Das zweite Kapitel gibt einen Eindruck von der Idee Softwaretests zu automatisieren und welche Ziele diese Arbeit verfolgt. Das dritte Kapitel beinhaltet fachliche und technische Anforderungen sowie eine Marktanalyse. Die fachlichen Anforderungen werden anhand von Fallbeispielen erläutert. Dabei geht es um Aufwand, Automatisierung und Vereinfachung von Unit-Tests sowie Regressionstests. Bei den technischen Anforderungen geht es um die Frage, welche Datenrepräsentation am besten für die Automatisierung von Unit-Tests geeignet ist. In der Marktanalyse werden vorhandene Anwendungen vorgestellt und ihre Grenzen erläutert.

Die Kenntnisse aus den Anforderungen und der Marktanalyse bilden die Grundlage für die Entwicklung eines Prototyps im vierten Kapitel. Verschiedene Konfigurationsmodelle zeigen, wie der Prototyp mit minimalem Aufwand steuerbar bleibt. Die Analyse gesammelter Testdaten aus einer realen Anwendung hilft bei der Entwicklung geeigneter Vergleichsverfahren für XML-Dokumente. Suchverfahren zur Ähnlichkeitsbestimmung von Objekten haben dabei eine große Bedeutung. Die Maßnahmen werden anschließend in der Evaluation bewertet. Das fünfte Kapitel beschreibt schließlich die Architektur des Prototyps.

Den Schluss bildet das sechste Kapitel; es enthält eine Zusammenfassung über den erreichten Stand und die offenen Punkte. Abschließend wird ein Blick in die Zukunft gewagt.

2. Vision

2.1. Einleitung

Unit-Tests [Spillner und Linz (2005)] sind unerlässlich, um die Funktionalität eines Programms zu prüfen. Aber ist es unerlässlich, sie von Hand zu schreiben?

Beim Schreiben eines Unit-Tests tritt für den Entwickler häufig das Problem auf, dass er Objekte mit vielen Eigenschaften testet, die wiederum Objekte mit Eigenschaften sind. Hierfür einen Unit-Test zu schreiben, ist wie Zwiebelschälen: vielschichtig und nur unter Tränen. Denn jedes Objekt benötigt eine Vergleichsfunktion, die eventuell jede Eigenschaft des Objekts prüft. Besonders bei einfachen Datenhaltern (zum Beispiel Business Objects, Fachwerte oder POJOs) ist der Aufwand groß, da sie üblicherweise viele Eigenschaften besitzen. Der Entwickler sieht sich mit immer wiederkehrenden Schritten konfrontiert: Schreibe eine Vergleichsfunktion für die zu testene Klasse. Prüfe in der Vergleichsfunktion jede Eigenschaft. Ist eine Eigenschaft ein Objekt einer anderen Klasse, dann schreibe für diese Klasse eine weitere Vergleichsfunktion. Diese Tätigkeit setzt sich für alle verwendeten Klassen fort. Beispiel 2.1 veranschaulicht den Vorgang anhand einer Vergleichsfunktion für Schiffe und zwei weiteren Klassen.

```
Vergleiche Schiff1 mit Schiff2:
    Prüfe: Name von Schiff1 ist gleich Name von Schiff2
    Prüfe: Reederei von Schiff1 ist gleich Reederei von Schiff2

Vergleiche Reederei1 mit Reederei2:
    Prüfe: Adresse von Reederei1 ist gleich Adresse von Reederei2

Vergleiche Adresse1 mit Adreese2:
    Prüfe: Straße von Adresse1 ist gleich Straße von Adresse2
```

Beispiel 2.1: Pseudocode um zwei Schiffe zu vergleichen

Die Vergleichsfunktion beginnt mit dem Vergleich zwischen den Objekten Schiff1 und Schiff2. Sie vergleicht alle Eigenschaften zwischen beiden Schiffen, darunter die Eigenschaft Reederei. Reederei ist eine weitere Klasse in der Anwendung. Für die Klasse Reederei ist somit auch eine Vergleichsfunktion erforderlich. Sie vergleicht alle Eigenschaften der Reederei. Da

die Reederei das Objekt Adresse als Eigenschaft enthält, ist eine weitere Vergleichsfunktion für die Klasse Adresse notwendig.

2.2. Ziele

Datenhalter zu vergleichen, ist unabhängig von der Anwendung, weil sie keine Geschäftslogik enthalten: Egal ob die Anwendung für das Versandhaus Amazon, für die deutsche Verkehrsleitzentrale oder für die Justizvollzugsanstalt Jülich ist, der Vergleich bleibt immer derselbe, da letztlich Werte (Bits) verglichen werden. Darum lässt sich ein allgemeines Programm entwickeln, das dem Entwickler die Routinearbeit abnimmt - im gewissen Sinne ein Programm zu schreiben, das Tests für Programme schreibt.

- Ziel ist es, den Unit-Test zu automatisieren, indem ein Programm die Vergleichsfunktionen erzeugt.

Die Automatisierung führt zu Einschränkungen: Schreibt der Entwickler den Test selbst, kann er ihn bei Problemen einsehen. Schreibt ein Programm den Test und es tritt ein Problem auf, wird es schwierig. Der Ablauf und die Ergebnisse des erzeugten Tests sind daher menschengerecht aufzubereiten und darzustellen, sodass der Entwickler im Problemfall Rückschlüsse auf die Ursache erhält.

- Ziel ist es, den Unit-Test zu automatisieren, ohne Verlust der Transparenz.

Durch die Automatisierung kommt erschwerend hinzu, dass der Entwickler die Kontrolle abgibt. Schreibt er den Test selbst, bestimmt er: was, wann und wie geprüft wird, zum Beispiel um bestimmte Eigenschaften in einer bestimmten Reihenfolge zu prüfen. Sobald ein Programm diese Entscheidungen trifft, geht dem Entwickler ein Stück Freiheit verloren.

- Ziel ist es, den Unit-Test zu automatisierien, sodass der Entwickler die Kontrolle behält.

2.3. Fazit

Der Aufwand für Softwaretests soll minimiert werden, indem ein Programm die Vergleichsfunktionen erzeugt. Als Vorbedingung darf die Vergleichsfunktion keine Geschäftslogik enthalten, da die erzeugte Vergleichsfunktion nur die Werte prüft, aber nicht ihre Bedeutung. Durch die Automatisierung verliert der Entwickler den Einblick, wie der Vergleich abläuft, daher ist eine detailierte Ausgabe mit Fehlerursache erforderlich. Nicht jeder Test kann gleich behandelt werden, deshalb behält der Entwickler die Möglichkeit, das Verhalten der erzeugten Vergleichsfunktion zu steuern.

3. Anforderungen

3.1. Fachliche Anforderungen

3.1.1. Einleitung

Die fachlichen Anforderungen beschreiben, welche Aufgaben der Prototyp erfüllen muss, um den Aufwand für Unit-Tests zu verringern. Sie stammen aus Gesprächen mit Entwicklern. Für jede Anforderung zeigt ein Fallbeispiel den Ursprung des Problems und die Notwendigkeit für eine Lösung.

3.1.2. 1. Ablauf des Unit-Tests beibehalten

Unit-Test ist ein Konzept, um Software zu testen. Der Ablauf ist immer derselbe, unabhängig von der Anwendung, Bibliothek oder Programmiersprache. Das Beispiel 3.1 veranschaulicht den allgemeinen Ablauf für einen Unit-Test:

```
1. Schreibe eine Funktion innerhalb der Anwendung
2. Rufe die Funktion im Unit-Test mit Testwerten auf
3. Prüfe das Ergebnis der Funktion mit einem Erwartungswert
   (Ergebnis und Erwartungswert sind abhängig von den Testwerten)
```

Beispiel 3.1: Allgemeiner Ablauf für einen Unit-Test

Wenn das Ergebnis dem Erwartungswert entspricht, geht der Entwickler davon aus, dass sich die Funktion aus Schritt 1 korrekt verhält. Das Verhalten der Funktion wird demnach nicht explizit geprüft, sondern implizit durch das Ergebnis. Aus diesem Grund sind Funktionen ohne Ergebnisse schwer zu testen, wie zum Beispiel eine Ausgabefunktion, die lediglich etwas auf den Bildschirm anzeigt, aber keine Werte verändert (von den Pixeln auf den Monitor abgesehen). Deshalb sind Unit-Tests wertorientiert: Es läuft immer darauf hinaus, zwei Werte zu vergleichen. Der Weg dahin unterscheidet sich je nach Art des Tests. Ferner existieren verhaltensorientierte Ansätze, beispielsweise in EasyMock [Freese (2007)] oder in FIT [Cunningham (2007)]. Sie haben in der Praxis bisher keine große Bedeutung und werden daher nicht weiter betrachtet.

3.1.3. 2. Vergleichsfunktionen für Unit-Tests automatisch erzeugen

Einfache Werte vergleichen

Einfache Werte kann ein Computer automatisch vergleichen. Sie sind fester Bestandteil einer Programmiersprache. Die Programmiersprache führt den Vergleich durch. Typische einfache Werte sind Zahlen, Buchstaben oder Wahrheitswerte. Das folgende Beispiel 3.2 zeigt den Ablauf für einen Unit-Test mit einfachen Werten:

```
1. Funktion:          addiere A und B
2. Funktionsaufruf:   addiere 1 und 2 in Ergebnis
3. Prüfen:            Ergebnis ist gleich 3 (Erwartungswert)
```

Beispiel 3.2: Unit-Test mit einfachen Werten

Die Wertüberprüfung findet statt in Schritt 3. Hier garantiert (Englisch: “assert”) der Unit-Test, dass Ergebnis und Erwartungswert gleich sind. Um das Konzept von Unit-Test beizubehalten, ist es notwendig, diesen Ablauf beizubehalten.

Komplexe Werte vergleichen

Ein komplexer Wert setzt sich aus beliebig vielen einfachen Werten zusammen. Unit-Tests für komplexe Werte haben einen höheren Aufwand, da der Entwickler zunächst eine Vergleichsfunktion bereitstellen muss, in der er festlegt, wie ein komplexer Wert zu vergleichen ist. Der Ablauf aus Beispiel 3.2 ist folgendermaßen zu erweitern:

```
1. Funktion:              setze Bundeskanzler
2. Vergleichsfunktion:    ist gleich für Bundeskanzler
3. Funktionsaufruf:       Angela ist Bundeskanzler
4. Prüfen:                Bundeskanzler ist gleich Angela (Erwartungswert)
```

Beispiel 3.3: Unit-Test für komplexe Werte

Damit der Entwickler den komplexen Wert Bunderkanzler vergleichen kann, schreibt er in Schritt 2 eine Vergleichsfunktion, die in Schritt 4 vom Unit-Test verwendet wird. In Hinblick auf die Vision soll der hinzugekommene Schritt 2 automatisiert werden und damit wegfallen. Das Konzept von Unit-Test wird somit beibehalten und verschlankt.

Einfache Werte			
Zahlen:	1	-1	1,23
Buchstaben:	'A'	'b'	'c'
Wahrheitswerte:	true	false	

Komplexer Wert			
	name	alter	weiblich
Bunderkanzler	A' 'n' 'g' 'e' 'l' 'a'	53	true

→ *Setzt sich aus einfachen Werten zusammen.*

Abbildung 3.1.: Einfache und komplexe Werte

3.1.4. 3. Listen mit komplexen Werten automatisch vergleichen

Für einen Computer ist es schwer, Listen zu vergleichen, da er ähnliche Werte finden und zuordnen muss. Ein Fallbeispiel verdeutlicht das Problem:

Eine Waldorfschule unternimmt einen Schulausflug mit einem Reisebus. Zwei Lehrer begleiten den Ausflug als Betreuer: Frau Rose und Herr Nelke. Der Reisebus holt die Schüler von der Schule ab. Frau Rose steht an der Bustür mit Stift und Papier. Sie notiert sich jeden Schüler, der an ihr vorbei in den Bus steigt. Nachdem alle Schüler eingestiegen sind, geht die Reise los. Am Ziel angekommen, stellt sich jetzt Herr Nelke an die Bustür mit eigenem Stift und eigenem Papier (Herr Nelke ist ein wenig eigen, was das Eigentum betrifft). Er notiert sich jeden Schüler, der an ihm vorbei aus dem Reisebus steigt. Anschließend prüfen Frau Rose und Herr Nelke, ob kein Schüler unterwegs verloren gegangen ist, indem sie ihre Listen vergleichen. Die beiden Lehrer stellen dabei fest, dass das nicht so einfach ist: Die Schüler sind weder in derselben Reihenfolge eingestiegen wie ausgestiegen, noch hat es Herr Nelke für nötig gehalten, die Nachnamen auszuschreiben, während Frau Rose jeden Schüler mit vollständigen Vor- und Nachnamen notiert hat. Beide einigen sich darauf, dass Frau Rose ihre Liste von oben nach unten vorliest und Herr Nelke jeden vorgelesenen Schüler in seiner Liste sucht und abhakt. Erfreut stellen beide fest, dass kein Schüler fehlt.

Sobald sich die Reihenfolge der Listeneinträge unterscheidet, muss für jeden Listeneintrag der einen Liste der passende Listeneintrag der anderen Liste gesucht werden. Im Fallbeispiel liest Frau Rose die Namen der Schüler vor, zum Beispiel Martin Vonwegen. Herr Nelke sucht dann in seiner Liste nach einen Martin V. Die Suche ist für Herrn Nelke etwas erschwert, da er die Namen anders aufgeschrieben hat, als es Frau Rose getan hat. Aber Herr Nelke weiß schließlich, dass sein Martin V. derselbe ist, wie Martin Vonwegen von Frau Rose. Schwierig wird es, einem Computer diesen Unterschied beizubringen, der eigentlich keiner ist. Das Problem tritt immer dann auf, wenn ein Computer Information interpretieren muss. Gesucht sind also Verfahren, mit denen ein Computer Listen ohne menschliche Hilfe vergleichen kann.

3.1.5. 4. Regressionstest im Unit-Test vereinfachen

Ein Regressionstest vergleicht den Istzustand mit einem früheren Sollzustand und garantiert, dass beide Zustände gleich geblieben sind. Der Unit-Test ermöglicht Regressionstests, allerdings ist der Entwickler dafür verantwortlich, sich den Sollzustand zu merken. Dies soll vereinfacht werden, indem die Verwaltung der Sollzustände automatisiert wird.

Hierzu ein Fallbeispiel: In der Strafvollzugsanstalt Jülich pflegt der Gefängnisleiter eine Liste aller derzeit Inhaftierten. Aufgrund der häufigen Ausbruchversuche beschließt der Gefängnisleiter, eine Inventur durchzuführen. Die Vollzugsbeamten kontrollieren daraufhin jede einzelne Zelle und erstellen eine neue Liste aller Inhaftierten. Der Gefängnisleiter erhält die zweite Liste und vergleicht seine alte Liste mit der neuen. Er hofft natürlich, dass alle Inhaftierten noch anwesend sind, die beiden Listen folglich denselben Inhalt aufweisen. Ob die Inhaftierten in derselben Reihenfolge stehen, ist dem Gefängnisleiter dabei egal.

Für den Unit-Test mit Regressionstest ergibt sich folgender Ablauf:

```
1. Funktion:          Inventur aller Zellen
2. Merke Zustand:     Erstelle Sollliste der Inhaftierten durch Inventur
3. Funktionsaufruf:   Erstelle Istliste der Inhaftierten durch Inventur
4. Prüfen:            Soll- und Istliste sind gleich (Erwartungswert)
```

Beispiel 3.4: Regressionstest

Am allgemeinen Ablauf aus Beispiel 3.1 hat sich wenig geändert. Hinzugekommen ist Schritt 2, in dem der Sollzustand für später gemerkt wird. Um diesen Schritt muss sich der Entwickler beim Regressionstest zusätzlich kümmern. In Schritt 4 findet der Vergleich zwischen Soll- und Istzustand statt. Um den Ablauf für Regressionstests zu vereinfachen, soll Schritt 2 wegfallen. Der Unit-Test merkt sich dann die Sollzustände und vergleicht sie mit den passenden Istzuständen.

Wichtig dabei ist die Erwartungskonformität des Entwicklers: Die Erweiterung muss sich nahtlos in Unit-Test einfügen und sich ähnlich verhalten, wie der Entwickler es vom Unit-Test gewohnt ist.

3.1.6. 5. Konfiguration automatischer Vergleiche

In den bisherigen Fallbeispielen kommt es häufig vor, dass das Gleichheitskriterium aufgeweicht wird. So spielt die Reihenfolge von Listenelementen keine Rolle, solange die Listen inhaltlich übereinstimmen, wie bei den Inhaftierten im Gefängis oder den Schülern auf Klassenfahrt. Auf der einen Seite soll ein automatisches Verfahren ohne zusätzliche Einstellungen auskommen, auf der anderen Seite haben die Fallbeispiele gezeigt, dass Ausnahmen

sinnvoll sind. Wünschenswert ist ein Kompromiss, der beide Welten zusammenbringt. Folgendes Fallbeispiel verdeutlicht die Notwendigkeit für Ausnahmen:

Sabine Zwietracht braucht dringend neue Schuhe, steht doch bald die Hochzeit ihrer Schwester an. Stefan Zwietracht ist die Hochzeit ziemlich egal, er interessiert sich viel mehr für einen größeren Flachbildfernseher. Die beiden betreten ein Schuhgeschäft. Nach einer Weile und vielen ausprobierten Schuhen später hat Sabine zwei Paar in die engere Wahl gezogen: beides Stöckelschuhe in Schwarz. Sabine kann sich nicht entscheiden und sucht Rat bei Stefan. Achselzuckend steht er vor den Schuhen, die für ihn beide gleich aussehen. Schwarz könne man immer tragen, meint er schließlich. Fassungslos schüttelt Sabine den Kopf und hält Stefan daraufhin einen Vortrag, über die riesigen Unterschiede beider Schuhe. So erfährt Stefan den Unterschied zwischen Pumps und Stiefel, die Bedeutung der Riemenzahl und wie hoch ein Absatz maximal sein darf. Selbst nach dieser ausführlichen Aufklärung bleibt Stefan dabei: Es sind nur Schuhe. Sabine genervt, weil Stefan sie nicht versteht, beschließt heute keine neuen Schuhe zu kaufen. Und den neuen Flachbildfernseher kann Stefan sich auch abschminken, der alte (Röhrenfernseher) sei schließlich genauso gut.

Wenn man etwas vergleicht, sind die Eigenschaften entscheidend. Sabine verfolgt jedes kleinste Detail eines Schuhs mit großem Interesse, während für Stefan keine Eigenschaft eines Schuhs ein Kaufgrund ist, solange er schwarz ist. In einem automatischen Vergleich sollten folglich die relevanten Eigenschaften wählbar sein.

3.1.7. Fazit

Der Prototyp gliedert sich nahtlos in das vorhandene Konzept von Unit-Test ein, indem der Ablauf beibehalten wird. Der Aufwand für das Testen wird verringert, indem Vergleichsfunktionen automatisch erzeugt werden. Eine besondere Herausforderung ist der Listenvergleich, da hier ähnliche Elemente zugeordnet werden müssen. Für Regressionstests werden Sollzustände automatisch verwaltet. Mithilfe der Konfiguration bestimmt der Entwickler relevante Eigenschaften für den Testfall.

3.2. Technische Anforderungen

3.2.1. Einleitung

Die fachlichen Anforderungen sind aufgestellt. Jetzt geht es darum, sie mithilfe vorhandener Technologien umzusetzen. Gesucht ist eine geeignete Technologie zum automatischen Vergleich von Werten. Hierbei sind verschiedene Auswahlkriterien zu berücksichtigen, die nachfolgend erläutert werden.

3.2.2. Auswahlkriterien

Die Verbreitung einer Technologie gibt erste Rückschlüsse auf vorhandene Dokumentation und Programme. Programme werden hierbei als Algorithmen aufgefasst, da der Schwerpunkt beim automatischen Vergleich in Algorithmen und Daten liegt. Vergleichsverfahren haben große Ähnlichkeit mit Suchverfahren: Zu einem Wert soll ein vergleichbarer anderer Wert gefunden werden, um anschließend die genauen Unterschiede der beiden Werte zu suchen. Um den Entwicklungsaufwand zu minimieren, lohnt es sich, die Technologien auf fertige Algorithmen zu untersuchen.

Ferner spielt die Abhängigkeit zu einer Technologie eine Rolle, da das Konzept von Unit-Test allgemein erweitert wird und nicht für eine bestimmte Programmiersprache oder Plattform.

Ein weiteres Kriterium für Vergleichsverfahren ist die Art und Weise, wie die Werte im Computer repräsentiert werden. Man kann Werte beispielsweise in Tabellenform festhalten oder sie in einem Netz aus verbundenen Knoten darstellen. Die Wahl der Repräsentation führt häufig zu einer bestimmten Technologie, so eignen sich für Tabellen zum Beispiel relationale Datenbanken. Darüber hinaus kann eine geschickte Repräsentation das Suchverfahren vereinfachen oder sogar beschleunigen, indem zusätzliche Information zu einem Wert hinterlegt werden. In der Regel entscheidet die Anwendung über die geeignete Repräsentation.

Die Lesbarkeit der Testwerte ist wichtig beim Softwaretest, wenn Entwickler sie von Hand kontrollieren müssen. Auch Anwender sollen in der Lage sein, die Testwerte auswerten zu können, ohne ein Informatikstudium absolviert zu haben.

Die Auswahlkriterien im Überblick:

- Verbreitung der Technologie
- Vorhandene Algorithmen
- Abhängigkeit von der Technologie
- Einschränkungen durch die Technologie

- Repräsentation der Werte
- Lesbarkeit der Werte
- Aufwandersparnis durch die Technologie

Anhand dieser Auswahlkriterien werden nachfolgend drei mögliche Technologien gegenübergestellt.

3.2.3. Repräsentation durch Objekte einer Programmiersprache

Da ein Unit-Test in einer bestimmten Programmiersprache vorliegt, liegt es nahe, die Programmiersprache auch für den Prototyp einzusetzen, um Technologiewildwuchs zu vermeiden. Betrachtet man die Programmiersprache als Technologie, hat man ein universelles Werkzeug gefunden. Auch die Repräsentation von komplexen Werten wird mitgeliefert: Strukturen in prozeduralen Programmiersprachen oder Objekte in objektorientierten. Die Konzepte sind beide ähnlich und werden nachfolgend als Objekt gleich behandelt [Lahres und Rayman (2006)]. Verwendet das Vergleichsverfahren ausschließlich Objekte, entfällt die Umwandlung von einer Repräsentation in eine andere und der Entwickler kann direkt auf den Objekten in der Programmiersprache arbeiten. Dies hat zurfolge, dass die Werte nur innerhalb der Programmiersprache lesbar sind. Weiterhin sind Objekte ein Konzept aus der Informatik, das sie für Anwender ohne Informatikkenntnisse schwer verständlich macht. Die Abbildung 3.2 veranschaulicht die Bestandteile eines Objekts.

Objekt (komplexer Wert)				
	Attribut 1	**Attribut 2**	**Attribut n**	} Metaebene
Objekt 1	wert 1	wert 2	wert 3	
Objekt 2	wert 4	wert 5	wert 6	} Objekte mit Werten
Objekt n	wert 7	wert 8	wert 9	

→ *Ein Wert kann wieder ein Objekt sein.*

Abbildung 3.2.: Aufbau eines Objekts

Das Problem bei Objekten ist die Abhängigkeit von der Programmiersprache. Das Konzept von Unit-Test ist unabhängig von der Programmiersprache, ebenso wie die geplante Erweiterung durch den Prototyp. Dennoch wurde der Unit-Test in verschiedenen Programmiersprachen problemlos umgesetzt. Die Programmiersprache kann jedoch beim Vergleich von komplexen Werten zu Einschränkungen führen, wenn beispielsweise keine Metaprogrammierung unterstützt wird. Erklärend hierzu ein Beispiel:

Toni verdient sein Geld mit waschen, anders ausgedrückt: mit Geldwäsche. Um von der Mafia unentdeckt zu bleiben, hat er sich auf unscheinbare Münzen spezialisiert. Auf jeder Münze steht der Wert und das Jahr der Prägung. Jede Münze hat je nach Wert einen größeren Umfang, von dem auch das Gewicht abhängt. Um die Herstellung zu vereinfachen, sind Tonis Münzen alle gleich groß, egal welchen Wert sie haben. Außerdem verzichtet er auf die Jahresangabe. Die Zahl für den Wert sieht dafür sehr authentisch aus - es ist aber immer die 50. Zum Weltspartag bringt Toni seine Münzen zur Bank. Der Bankier nimmt die schweren Taschen entgegen und entleert sie über den Zählautomaten, der daraufhin geschäftig zu klimpern beginnt. Der Automat akzeptiert anstandslos sämtliche Münzen, da für ihn nur das Gewicht ausschlaggebend ist.

Der Zählautomat erkennt eine Münze am Gewicht. Er ignoriert den Umfang, das Jahr und den Wert. Um in einer Programmiersprache die Münzen zu vergleichen, ist die erste Hürde das Fehlen bestimmter Eigenschaften aufzudecken. Jede Münze hat eine Jahresangabe, Tonis Münzen aber nicht. Um dies festzustellen, muss eine Programmiersprache auf die Eigenschaften der Münze zugreifen können, die sogenannte Metaebene.

Ferner sind Vergleichsalgorithmen üblicherweise kein fester Bestandteil der Programmiersprache. Existiert für den Vergleich keine Bibliothek, dann ist der Aufwand zum funktionsfähigen Prototyp sehr hoch.

Die Vor- und Nachteile im Überblick:
+ Einheitliche Repräsentation
+ Einheitliche Technologie
+ Hohe Flexibilität durch die Programmiersprache
- Flexibilität abhängig von der Programmiersprache
- Plattform eventuell abhängig von der Programmiersprache
- Objekte nur durch die Programmiersprache lesbar
- Hoher Einstiegsaufwand

3.2.4. Repräsentation durch Tabellen mit relationalen Datenbanken

Anstatt Werte durch Objekte zu repräsentieren, kann man sie in Tabellen ablegen. In einer Tabelle steht zum Beispiel jede Zeile für eine Münze und jede Spalte für eine Eigenschaft (Umfang, Gewicht, Wert, Jahresangabe). Eine Repräsentation als Tabelle verbessert die lesbar und ist auch für Nichtinformatiker leicht verständlich. Abbildung 3.3 zeigt den Aufbau einer Tabelle.

Tabellen eignen sich für relationale Datenbanken. Eine Datenbank ist ein Programm, das darauf spezialisiert ist, Werte zu speichern und bei Bedarf wiederzufinden. Relational bedeutet, dass die Datenbank die Werte in Tabellen ablegt [Heuer und Saake (2000)]. Datenbanken

Tabelle: Münze				
Umfang	**Gewicht**	**Wert**	**Jahresangabe**	} Schema
0.5	10	50	1983	
0.3	7	30	1991	} Werte
0.4	7	40	1989	

Abbildung 3.3.: Aufbau einer Tabelle

verfügen über eine mächtige Abfragesprache zur Suche nach Werten, die besonders nützlich für den Vergleich (die Suche nach Werten) erscheint. Klassische Datenbanken bieten in der Regel keine Vergleichsmöglichkeiten, wie sie in Abschnitt 3 gefordert sind. Es existieren jedoch spezialisierte Datenbanken [Schmitt (2005)]. Für den Vergleich müsste die Abfragesprache erweitert werden. Alternativ kann man die Abfragesprache ausschließlich für die Suche verwenden und den Vergleich in einer Programmiersprache umsetzen, das erfordert aber eine Umwandlung zwischen Tabelle und Objekt. Diese Arbeit könnte beispielsweise ein objektrelationaler Mapper übernehmen.

Um auf die Eigenschaften einer Münze zuzugreifen, kann man bei Tabellen auf das sogenannte Schema zurückgreifen. Das Schema beinhaltet Information über die Tabelle selbst, wie die Namen der Spalten und den Namen der Tabelle. Diese Information bestimmt die Bedeutung der Werte. Falls eine echte Münze mit einer gefälschten Münze verglichen wird, so kann man über das Schema herausfinden, ob beide Münzen dieselben Eigenschaften besitzen, wie zum Beispiel Umfang, Gewicht und Wert, aber keine Jahrangabe.

Tabellen setzen keine relationalen Datenbanken voraus, aber die Fähigkeiten einer Datenbank selbst zu entwickeln ist unnötig in Anbetracht existierender Datenbanken. Anderseits wirkt der Einsatz einer Datenbank ausschließlich für Vergleichszwecke übertrieben, da einige Hauptfunktionen der Datenbank ungenutzt bleiben, wie zum Beispiel Transaktionen.

Die Vor- und Nachteile im Überblick:
+ Tabellen ermöglichen die Nutzung von relationalen Datenbanken
+ Verwaltung der Werte durch die Datenbank
+ Ausgereifte, verbreitete Technologie
+ Mächtige Abfragesprache für die Suche nach Werten
+ Tabellen sind unabhängig von Programmiersprache und Plattform
+ Tabellen sind verständlich und gut lesbar
- Funktionalität der Datenbank wird nicht ausgereizt
- Abhängigkeit zu einer herstellerspezifischen Datenbank
- Wechsel der Repräsentation zwischen Datenbank und Programmiersprache erforderlich
- Vergleichsfunktionen nur in spezialisierten Datenbanken verfügbar
- Aufwand: vorhandene Funktionen nutzbar, Anpassungen sind aber nötig

3.2.5. Repräsentation durch XML

Die E**x**tensible **M**arkup **L**anguage, abgekürzt XML, ist eine verbreitete Sprache, um strukturierte Daten zu beschreiben [W3C (b)]. In erster Linie handelt es sich bei XML um eine Repräsentation für Werte. Hinzu kommen Standards, die unter anderen folgende Funktionen beinhalten: eine Sprache zur Beschreibung des Schemas, eine Abfragesprache (XPath) und eine Transformationssprache (XSLT). Die meisten Programmiersprachen unterstützen diese Standards von Haus aus.

Liegen Werte in XML vor, sind sie ohne weiteres lesbar. Erreicht wird dies durch die selbstbeschreibende Beschaffenheit, die besonders für Menschen geeignet ist. Sogenannte Bezeichner (Englisch: “tags”) beschreiben die Bedeutung der Werte. Falls vorhanden, legt ein Schema-Dokument die erlaubte Struktur der Bezeichner fest.

Eine Münze kann man zum Beispiel durch den Bezeichner <Münze> darstellen. Die Eigenschaften der Münze werden ebenfalls durch Bezeichner dargestellt: <Umfang>, <Gewicht>, <Wert>, <Jahr>. Um die Eigenschaften der Münze zuzuordnen, werden sie durch den Bezeichner <Münze> umschlossen. Ein konkreter Wert wird durch den jeweiligen Bezeichner umschlossen. Die Werte selbst sind keine Bezeichner. Abbildung 3.4 demonstriert den Aufbau einer Münze in XML.

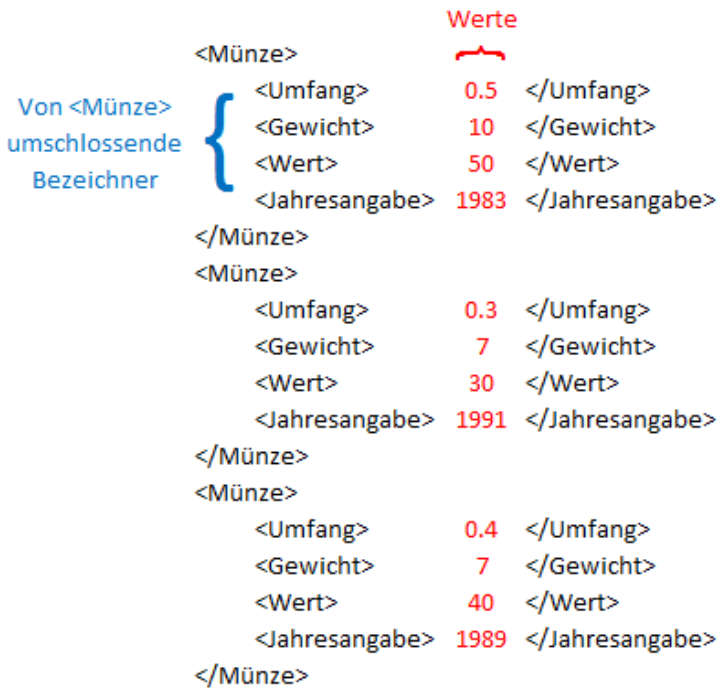

Abbildung 3.4.: Aufbau von XML

XPath ist eine Sprache für die Suche nach Werten und Eigenschaften. Allerdings werden keine Vergleichsverfahren angeboten; Vergleichsalgorithmen für XML befinden sich zurzeit in der Forschung. Entweder erweitert man XPath um Vergleichsverfahren oder man verwendet XPATH als reine Abfragesprache, um den Vergleich in einer Programmiersprache zu erleichtern. Letzteres erfordert eine Umwandlung zwischen XML und Objekten, also zusätzlichen Aufwand.

Die Vor- und Nachteile im Überblick:
+ Verbreiteter Standard
+ Zusätzliche Funktionen wie XML-Schema, XPATH und XSLT
+ Unabhängig von der Programmiersprache und Plattform
+ Lesbarkeit durch selbstbeschreibende Beschaffenheit
- Vergleichsverfahren sind nicht vorhanden
- Wechsel der Repräsentation zwischen XML und Programmiersprache erforderlich

3.2.6. Fazit

Die Wahl der Repräsentation entscheidet über verfügbare Technologien und über den erforderlichen Aufwand.

In Programmiersprachen werden die Werte als Objekte dargestellt, die ausschließlich in der Programmiersprache zur Verfügung stehen. Darunter leidet die Lesbarkeit. Einfache Suchverfahren sind in den meisten Programmiersprachen bereits vorhanden, wahrscheinlich sind aber Vergleichsverfahren zu ergänzen.

Tabellen und XML haben ähnliche Merkmale: Beide sind unabhängig von Programmiersprache und Plattform. Weiterhin sind sie selbstbeschreibend und damit für den Menschen lesbar. Der größte Unterschied sind die jeweils verfügbaren Technologien. Relationale Datenbanken speichern Werte in Tabellen, um sie anschließend mittels Abfragesprache effizient zu suchen. Bei XML hingegen gibt es Standards, wie Abfrage- und Transformationssprache. Vergleichsfunktionen fehlen, können aber durch eine Programmiersprache ergänzt werden, womit allerdings die Umwandlung in Objekte erforderlich wird.

Jede vorgestellte Technologie ist auf eine Repräsentation spezialisiert. Alle Repräsentationen sind geeignet für den Vergleich von komplexen Werten. Aber keine der vorgestellten Technologien verfügt über die geforderten Vergleichsverfahren.

Dennoch scheint XML der geeignete Kandidat zu sein. Der Grund lautet, dass XML technologieunabhängig ist. Zudem bietet XML dieselben Vorteile wie Tabellen, ohne auf relationale Datenbanken eines bestimmten Herstellers zurückgreifen zu müssen. Bei XML sind die zusätzlichen Standards optional, der Entwickler hat demnach die Freiheit, sie einzusetzen. Die Unabhängigkeit ist wichtig, um den Unit-Test allgemein zu erweitern.

Die Lesbarkeit ist ein weiteres wichtiges Kriterium, da sowohl Entwickler als auch Anwender darauf angewiesen sind, die Werte festlegen oder prüfen zu können. Objekte haben hier das Nachsehen, da sie nur über die Programmiersprache lesbar sind und zudem Informatikkenntnisse voraussetzen. XML und Tabellen sind hingegen für den Menschen ohne Probleme lesbar. Allerdings unterscheiden sich beide, sobald komplexe Werte ineinander geschachtelt vorliegen. Abbildung 3.5 veranschaulicht den Unterschied.

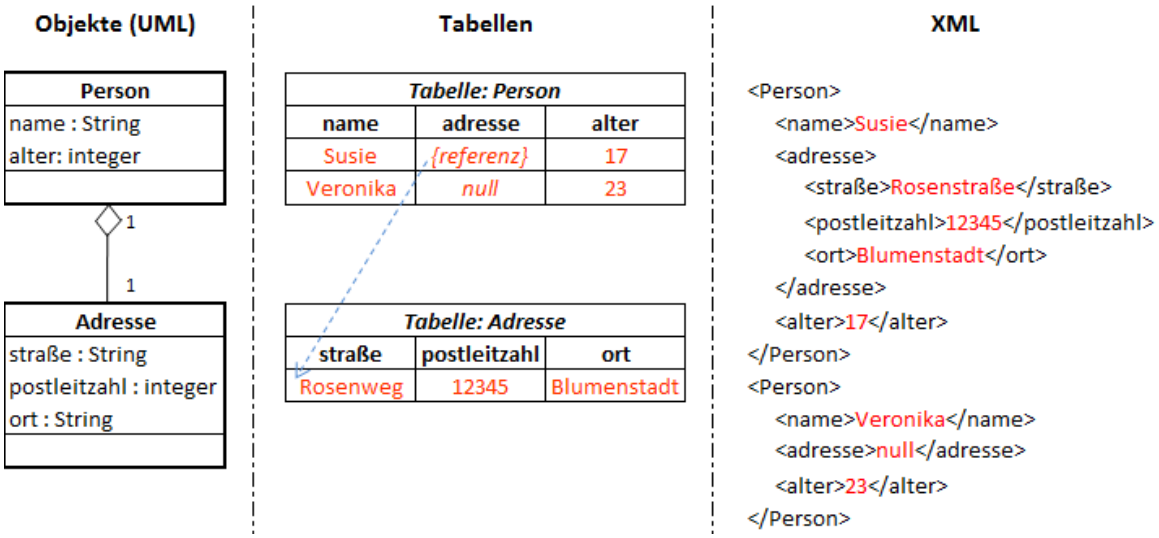

Abbildung 3.5.: Repräsentation geschachtelter Werte

Besteht ein komplexer Wert aus komplexen Werten, sind in einer Tabelle weitere Tabellen einzufügen. Obwohl man geschachtelte Tabellen darstellen könnte, verzichtet man im Zuge der Übersichtlichkeit darauf. Stattdessen wird eine Referenz auf eine andere Tabelle eingefügt, die den enthaltenen komplexen Wert stellvertretend darstellt. Dies hat zur Folge, dass zusammenhängende Werte an verschiedenen Orten stehen. Für Objekte in einer Programmiersprache wird dasselbe Prinzip verwendet.

Bei XML bildet man geschachtelte komplexe Werte durch unterschiedliche Ebenen ab, indem ein komplexer Wert einen anderen komplexen Wert umschließt. Dadurch sind beliebig komplexe Werte ineinander darstellbar ohne optische Anpassungen.

Der Prototyp verwendet also XML, weil XML unabhängig von Programmiersprache und Plattform ist. Außerdem ist XML für den Menschen lesbar und kann geschachtelte komplexe Werte in einem Dokument darstellen.

3.3. Marktanalyse

3.3.1. Einleitung

Bevor die Anforderungen im Prototyp umgesetzt werden, ist die Frage zu klären, ob bereits Bibliotheken existieren, die Funktionen bieten, um Unit-Tests zu automatisieren. Denkbar ist es, vorhandene Bibliotheken als Grundlage für die Entwicklung zu nehmen.

3.3.2. Vorhandene Bibliotheken

Zunächst gilt es, komplexe Werte (Objekte) aus einer Programmiersprache nach XML umzuwandeln und wieder zurück. Die Open Source Bibliothek XStream [Walnes (2007)] erfüllt diese Aufgabe: Sie wandelt die Eigenschaften in Bezeichner um und setzt die konkreten Werte zwischen den Bezeichnern. Der Objekttyp wird ebenfalls zum Bezeichner und umschließt alle anderen Bezeicher. Referenzen auf andere Objekte werden in XML-Attribute übertragen. Abbildung 3.6 veranschaulicht die Umwandlung.

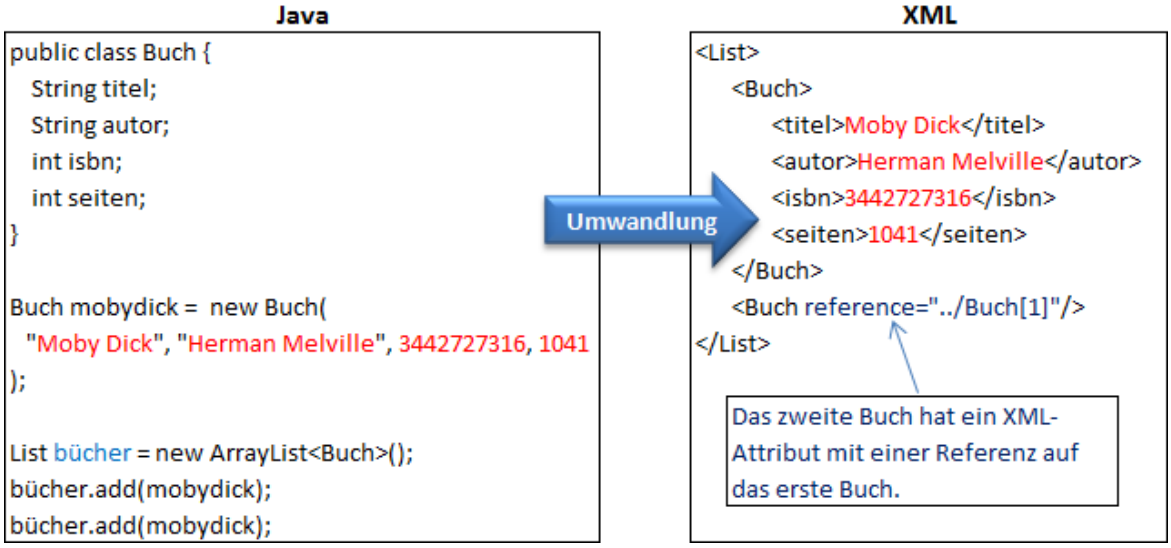

Abbildung 3.6.: Java-Objekt “bücher” in XML umgewandelt

Liegen die Werte in XML vor, fehlt noch ein Programm, das XML-Dokumente vergleicht. Die Open Source Bibliothek XMLUnit [Bacon und Martin (2007)] erfüllt diese Aufgabe. Es bietet ähnliche Funktionen wie ein Unit-Test: Es garantiert die Gleichheit zwischen zwei XML-Dokumenten.

Im folgenden Kapitel 4 “Prototyp” ab Seite 27 werden die Schwierigkeiten erläutert, die XMLUnit beim Vergleich von komplexen Werten hat und wie sie gelöst werden. Da komplexe Werte in einer Programmiersprache die Regel sind, handelt es sich hierbei um ein ernstes Problem.

XMLUnit ist ursprünglich enstanden, um Nachrichten für Webservices [W3C (c)] darauf zu prüfen, ob sie korrekt erzeugt werden. Normalerweise enthalten die Nachrichten einfache Werte und sind daher ohne Probleme zu vergleichen. Sobald verschachtelte komplexe Werte vorkommen, wird der Vergleich schwieriger. Zusätzlich mangelt es XMLUnit an Konfigurierbarkeit, wie es die fachlichen Anforderungen verlangen. Dennoch bietet XMLUnit eine solide Grundlage für einen Prototyp.

Ein großer Pluspunkt von XMLUnit ist, dass es Unterschiede klassifiziert. Damit fällt ein erheblicher Aufwand weg, um zum Beispiel Strukturunterschiede von Wertunterschiede zu unterscheiden.

3.3.3. Framework for Integrated Test (kurz: FIT)

Bei FIT gibt der Anwender die Testwerte für den Unit-Test vor [Cunningham (2007)]. Der Anwender benötigt hierzu keine Programmierkenntnisse: Er trägt die Erwartungswerte für einen Testfall in eine für ihn verständliche Tabelle ein. Die Tabelle erhält der Anwender vom Entwickler. Sie beinhaltet Zellen für die Eigenschaften der Objekte, die im Testfall vorkommen. Damit sind Tests realisierbar, die auf den Erwartungen der Anwender zugeschnitten sind; sie ergänzen die restlichen Tests der Entwickler um eine weitere Facette.

Der Prototyp unterstützt ebenfalls FIT. Der Unterschied besteht darin, dass der Anwender anstelle von Tabellen strukturierte XML-Dokumente ausfüllt. Ein FIT-Test mit XML bekommt den Namen FIX: “Fix It with XML”. Wie man aus einem Objekt ein XML-Dokument erzeugt, zeigt das Listing 3.5 in Pseudocode.

```
xml-Datei = XmlDiff.writeObject(objekt);
speichere xml-Datei;
```

Listing 3.5: Objekt ins XML-Format umwandeln

Als Eingabe dient ein beliebiges Objekt. Aus den Eigenschaften des Objekts wird die Struktur für das XML-Dokument erzeugt. Anschließend wird das XML-Dokument als Datei gespeichert und dem Anwender übergeben. Der trägt die Werte in das XML-Dokument ein, speichert es ab und gibt es zurück zum Entwickler. Schließlich wird der Test ausgeführt:

```
sollObjekt = lade xml-Datei; // Testwerte vom Anwender
XmlDiff.assertNoDifferences(sollObjekt, istObjekt);
```

Listing 3.6: FIX-Testwerte vergleichen

Abbildung 3.7 zeigt die Schritte eines FIX-Tests mitsamt aller Akteure und anfallender Dokumente. Vom Ablauf gleicht FIX einem Regressionstest, einziger Unterschied ist die Herkunft der Sollwerte.

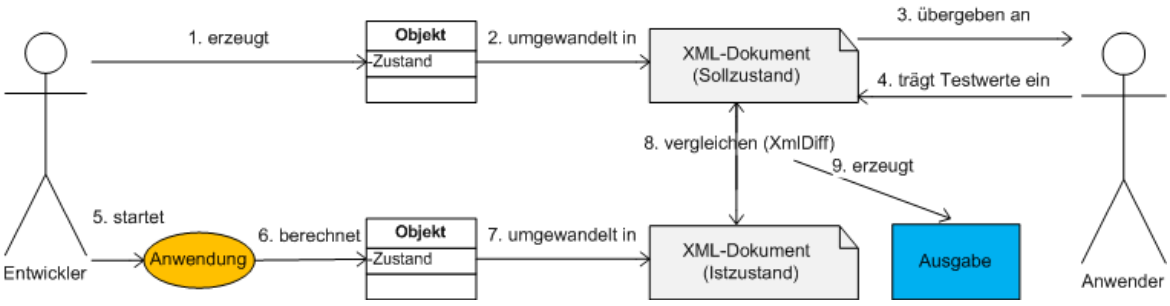

Abbildung 3.7.: Ablauf eines FIX-Tests

FIX bietet gegenüber FIT den Vorteil, dass jedes Objekt in einem einzigen XML-Dokument darstellbar ist. Bei FIT ist für jeden Objekttyp eine neue Tabelle anzulegen. Die Tabellen sind dann untereinander abhängig. Ein weiterer Vorteil gegenüber FIT ist die Transformationssprache XSLT, mit der das XML-Dokument ohne großen Aufwand für den Anwender beliebig angepasst werden kann. Nachteil gegenüber FIT ist die bisher begrenzte Werkzeugunterstützung, um zum Beispiel Quelltextdateien in XML-Dokumente umzuwandeln.

3.3.4. Fazit

Kombiniert man vorhandene Bibliotheken, kommt man dem automatisierten Unit-Test sehr nah: XStream wandelt Objekte um in XML-Dokumente und XMLUnit vergleicht sie anschließend. Allerdings sind Anpassungen notwendig, damit die Symbiose auch mit Objekten praxistauglich funktioniert.

Darüber hinaus ist es möglich, FIT-Tests mit XML-Dokumenten statt mit Tabellen durchzuführen: Ein Anwender trägt die Testwerte für einen Testfall in ein vorgefertigtes XML-Dokument ein. Der Entwickler kann anschließend die Werte im XML-Dokument mit den Istwerten der Anwendung testen. Angelehnt an FIT wird ein Unit-Test mit XML-Dokumenten FIX genannt.

Abschließend ist festzuhalten, dass derzeit keine Bibliothek existiert, die Unit-Tests so automatisiert, dass der Aufwand wegfällt, Testfälle zu schreiben. Diese Lücke schließt der nachfolgend beschriebene Prototyp.

4. Prototyp: XmlDiff

4.1. Einleitung

In diesem Kapitel geht es um die Entwicklung des Prototyps XmlDiff. Den Anfang bildet eine ABC-Analyse der *Ziele*. Anschließend folgt in der *Spezifikation* ein Überblick über die Bestandteile des Prototyps sowie über den internen Ablauf. Daraufhin geht der Abschnitt *Konfiguration* auf Modelle ein, durch die man den automatischen Vergleich steuern kann. In der *Analyse* werden gesammelte Unterschiede ausgewertet, um mit dem Wissen ein *Vergleichsverfahren* zu entwickeln. Dabei wird sich herausstellen, dass *Suchverfahren* eine entscheidene Rolle spielen. In der *Evaluation* werden die getroffenen Maßnahmen schließlich bewertet.

4.2. Ziele

Dieser Abschnitt zeigt in einer Tabelle die Ziele, die maßgebend für die Entwicklung des Prototyps sind. Sie gehen aus Gesprächen mit Entwicklern hervor sowie aus den Anforderungen aus Kapitel 3 ab Seite 12. Während der Entwicklung sind weitere Ideen der Entwickler eingeflossen. Eine ABC-Analyse für jedes Ziel gibt Aufschluss über die Priorität, wobei A für eine hohe Priorität steht und C für eine niedrige.

Ziel	**ABC-Analyse**
Fachlich	
Zugängliche Schnittstelle für den Entwickler bereitstellen.	A
Automatische Verwaltung der Objekte für den Regressionstest.	A
Konfiguration des Vergleichs durch möglichst wenig zusätzlichen Programmcode.	A
Die gefundenen Unterschiede übersichtlich anzeigen.	B
Die Sprache der Ausgabe orientiert sich an die Programmiersprache, nicht an XML.	B
Ein Eclipse-Plugin entwickeln, das die Handhabung zur Darstellung der Unterschiede erleichtert.	C
Ein Werkzeug entwickeln, das XML-Unterschiede grafisch darstellt.	C
Technisch	
Vergleichsverfahren für komplexe Objekte optimieren.	A
Suchverfahren zur Ähnlichkeitsbestimmung von Objekten entwickeln.	A
Objekte für den Regressionstest dauerhaft speichern und verwalten.	A
Erkennung bestimmter Unterschiede in einem abgestuften Vergleichsverfahren.	B
Integration in JUnit Version 3 und 4.	B
Architektur	
Minimale Schnittstelle für den Entwickler.	A
Bereitstellen expliziter Funktionen für einen Regressionstest.	A
Einheitliche Konfiguration über ein Konfigurationsobjekt.	A
Vererbung der Schnittstelle für den Entwickler vermeiden.	B

4.3. Spezifikation

Der Prototyp XmlDiff ist eine Bibliothek, die in der Programmiersprache Java geschrieben und ausschließlich für Unit-Tests in Java anwendbar ist. Durch die Verwendung von XML sind theoretisch andere Programmiersprachen möglich.

XmlDiff setzt für Unit-Tests die frei verfügbare Bibliothek JUnit [Gamma und Beck (2007)] voraus. Ohne JUnit kann XmlDiff ebenfalls verwendet werden, allerdings mit eingeschränkten Funktionsumfang. XmlDiff baut auf den beiden Bibliotheken XMLUnit und XStream auf, sie sind daher zwingend erforderlich.

Es werden keine Änderungen an der Bibliothek JUnit vorgenommen. Stattdessen bietet XmlDiff eigene Funktionen an, die denen von JUnit ähneln. Dadurch wird verhindert, dass für jede neue JUnit-Version Anpassungen an XmlDiff vorgenommen werden müssen.

Abbildung 4.1 auf Seite 30 stellt die Bestandteile des Prototyps dar sowie den internen Programmablauf.

Die Schnittstelle für den Entwickler (“API”) nimmt zwei Objekte entgegen. Alternativ können XML-Dokumente übergeben werden, zum Beispiel für einen Regressionstest. Die Bibliothek XStream wandelt die Objekte um in XML-Dokumente. Die XML-Dokumente werden dann für den Vergleich an die Bibliothek XMLUnit übergeben. XMLUnit greift auf ein bereitgestelltes Suchverfahren zurück. Optional kann der Entwickler eine Konfiguration übergeben, um das Suchverfahren und den Vergleich zu steuern. Stellt der Entwickler keine Konfiguration zur Verfügung, verwendet der Prototyp vordefinierte Einstellungen. Ist der Vergleich abgeschlossen, kann der Prototyp verschiedene Ergebnisse zurückliefern: die XML-Dokumente der umgewandelten Objekte, ein aufbereitetes HTML-Dokument oder eine Ausgabe der Unterschiede auf der Konsole. Handelt es sich um einen Unit-Test, dann erhält die Bibliothek JUnit das Ergebnis des Vergleichs.

In den folgenden Abschnitten werden die wichtigsten Bestandteile ausführlich erläutert.

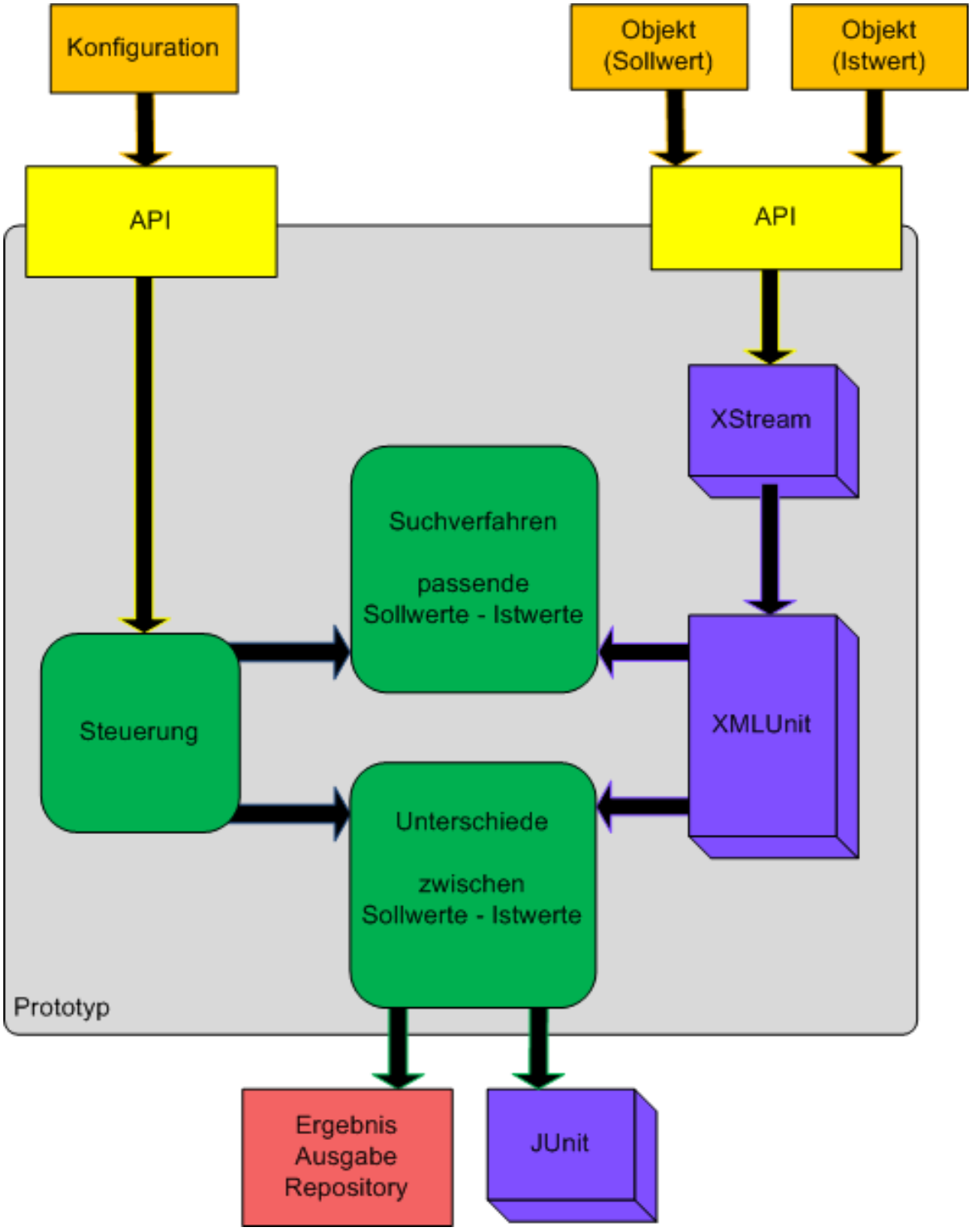

Abbildung 4.1.: Ablauf und Bestandteile des Prototyps

4.4. Konfiguration

4.4.1. Einleitung

Akzeptanz, Erwartungskonformität und Zugänglichkeit sind wichtige Faktoren, damit der Prototyp mehr als einmal eingesetzt wird. Der Entwickler ist nicht nur vom praktischen Nutzen des Prototyps zu überzeugen, sondern auch von dessen Bedienbarkeit. Für einen schnellen Einstieg sorgt ein einziger Funktionsaufruf. Zudem ist der Prototyp so vorkonfiguriert, dass keine weiteren Eingaben notwendig sind. Manche Testfälle erfordern jedoch zusätzliche Einstellungen. Der dafür notwendige Quelltext soll dann möglichst minimal ausfallen.

Der minimale Funktionsaufruf ohne Einstellungen in Listing 4.1 dient fortan als Maßstab:

```
assertNoDifferences(objekt1, objekt2);
```

Listing 4.1: Funktionsaufruf zum Vergleich zweier Objekte

Während der Entwicklung sind mehrere Konfigurationsmodelle entstanden. Sie ermöglichen es, dieselben Einstellungen auf unterschiedliche Weise vorzunehmen. Zu jedem Modell gibt es ein explorativen Prototyp, der von den Entwicklern getestet und bewertet wurde.

4.4.2. Funktionalorientiertes Konfigurationsmodell

Das erste Modell ist ein funktionaler Ansatz. Ein Konfigurationsobjekt enthält alle Einstellungen sowie das Objekt, das getestet wird. Die Einstellungen werden über statische Funktionen verändert. Der erste Funktionsaufruf nimmt das Objekt entgegen und wandelt es in ein Konfigurationsobjekt um. Jede weitere Funktion nimmt das Konfigurationsobjekt entgegen, verändert es und gibt es dann als Rückgabewert zurück. Auf diese Weise lassen sich beliebig geschachtelte Funktionsaufrufe verketten, weil jeder Aufruf die Einstellungen weiterreicht. Die Einstellungen werden schließlich vor dem Vergleich aus dem Konfigurationsobjekt entnommen. Jede denkbare Kombination ist somit möglich. Ein Beispiel zeigt das Listing 4.2:

```
assertNoDifferences(ignoreContent(ignoreOrder(object1)), object2);
```

Listing 4.2: Konfiguration mit geschachtelten Funktionsaufrufen

Das Modell hat bei Entwicklern zu Irritationen geführt: Es ist nicht klar geworden, warum sich die Funktionen nur auf ein Objekt beziehen, um den Vergleich für beide Objekte steuern. Die Konfiguration des Vergleichs ist zu indirekt.

Ein weiteres Problem sind Funktionen mit mehreren Parametern. Zum Beispiel lassen sich bestimmte Eigenschaften ignorieren, der Entwickler muss hierzu die gewünschten Eigenschaften übergeben. Listing 4.3 zeigt das Beispiel:

```
assertNoDifferences(ignoreField(object1,"timestamp"),object2);
```

Listing 4.3: Konfiguration mit mehreren Parametern

Durch den zusätzlichen Parameter “timestamp” wird die Verschachtelung unübersichtlich. Daraus folgt das nächste Problem: Je länger die Funktionsaufrufe, desto schlechter lesbar der Quelltext.

Der Charme des funktionalorientierten Modells liegt in seiner Einfachheit. Es spielt seine Stärke bei wenig verschachtelten Funktionsaufrufen aus. Der Entwickler kann jederzeit Funktionen hinzufügen, ohne Anpassungen am vorhandenen Quelltext vorzunehmen.

Dennoch stellt das Modell einen Bruch zur objektorientierten Programmiersprache da. Der Entwickler ist gewohnt, Objekte direkt zu bearbeiten, um sie anschließend zu übergeben. Im funktionalorientierten Modell wird das Konfigurationsobjekt hingegen indirekt bearbeitet.

4.4.3. Objektorientiertes Konfigurationsmodell

Das zweite Modell orientiert sich deshalb an objektorientierten Programmiersprachen. Der Entwickler nimmt alle Einstellungen an einem Konfigurationsobjekt vor. Für jede mögliche Einstellung bietet das Konfigurationsobjekt eine Funktion. Der Entwickler ruft beliebig Funktionen auf und übergibt schließlich das fertige Konfigurationsobjekt als Parameter. Listing 4.4 demonstriert den Ablauf:

```
config = new Config();   // Konfigurationsobjekt erzeugen
config.ignoreOrder();    // 1. Einstellung setzen
config.ignoreContent(); // 2. Einstellung setzen
assertNoDifferences(object1,object2,config);
```

Listing 4.4: Das Konfigurationsobjekt in Aktion

Der Quelltext bleibt übersichtlich, auch wenn viele Einstellungen hinzukommen. Zudem muss der vorhandene Quelltext nicht angepasst werden. Der Nachteil zum funktionalorientierten Modell ist der Verlust des Einzeilerfaktors: Sobald eine Einstellung benötigt wird, muss der Entwickler zuerst ein Konfigurationsobjekt erzeugen und anschließend die gewünschten Funktionen aufrufen.

4.4.4. Vereinfachtes objektorientiertes Konfigurationsmodell

Im vereinfachten objektorientierten Konfigurationsmodell gilt es, das Konfigurationsobjekt und die Funktionsaufrufe im Testfall zu vermeiden. Hierzu werden spezielle Funktionen be-

reitgestellt, die ein fertiges Konfigurationsobjekt zurückgeben. Häufig benötigte Einstellungen werden vordefiniert und sind somit wiederverwendbar. Ein Beispiel zeigt Listing 4.5:

```
assertNoDifferences(object1, object2, Config.ignoreOrderAndContent());
```

Listing 4.5: Konfigurationsobjekt aus vordefinierter Funktion

Mit dieser Variante sind weiterhin alle Einstellungen möglich. Im Testfall reicht eine Zeile, um den Vergleich zu konfigurieren, womit die Lesbarkeit bewahrt bleibt. Als Nachteil müssen die Funktionen vordefiniert sein. Um den Aufwand für den Entwickler gering zu halten, sind häufig benötigte Einstellungen im Prototyp bereits umgesetzt.

4.4.5. Fazit

Konfiguration dient dazu, den Vergleich für den Testfall anzupassen. Dadurch lässt sich der Einsatzbereich des Prototyps vergrößern. Viele Einstellungen sind kombinierbar, deshalb wird darauf Wert gelegt, dass im Testfall möglichst wenig Quelltext entsteht. Zwei unterschiedliche Modelle haben sich vorgestellt:

Das funktionalorientierte Modell nutzt beliebig viele schachtelbare Funktionen zur Konfiguration. Einstellungen lassen sich so sehr kurz formulieren. Je komplexer allerdings die Einstellungen, desto unübersichtlicher wird es. Darüber hinaus wirkt das funktionalorientierte Modell in einer objektorientierten Programmiersprache wie ein Fremdkörper.

Beim objektorientierten Modell wird ein Konfigurationsobjekt für den Vergleich übergeben. Im Testfall ist viel Quelltext nötig, um das Konfigurationsobjekt zu erzeugen und einzustellen. Daher stellt das vereinfachte objektorientierte Modell spezielle Funktionen zur Verfügung, die fertige Konfigurationsobjekte zurückgeben. Dadurch ist lediglich ein zusätzlicher Parameter erforderlich.

Der folgenden Abschnitt 4.5 “Analyse” beschreibt häufig aufgetretene Unterschiede. Mit passender Konfiguration lassen sich für alle Unterschiede Ausnahmen definieren.

4.5. Analyse

4.5.1. Einleitung

Die Analyse soll helfen, das Problem einzuschränken, um bessere Ergebnisse beim automatischen Vergleich zu erzielen.

In einem Zeitraum von zwei Monaten werden Testwerte gesammelt. Die Testwerte stammen aus den Unit-Tests einer Anwendung, die sich im produktiven Einsatz befindet, womit die Testwerte eine realitätsnahe Stichprobe repräsentieren. Zur Stichprobe zählen insgesamt 849 fehlgeschlagene Vergleiche. Bei jedem Vergleich werden zwei XML-Dokumente erzeugt: eines für den Sollwert und eines für den Istwert. Beide Dokumente werden anschließend von Hand durchgesehen und ausgewertet. Daraufhin werden die Unterschiede klassifiziert und ihre Häufigkeit festgehalten.

4.5.2. Unterschiede zwischen Struktur und Wert

Unterschiede in XML-Dokumenten lassen sich in zwei Arten einteilen: Struktur- und Wertunterschiede. Ein Strukturunterschied liegt vor, sobald ein Bezeichner in einem Dokument auftritt und im anderen Dokument nicht. Dies tritt ein, wenn neue Bezeichner hinzugefügt, existierende entfernt oder umbenannt werden. Auch die Reihenfolge der Bezeichner ändert die Struktur. Zwei Dokumente sind strukturgleich, wenn sie dieselben Bezeichner in derselben Reihenfolge aufweisen. In der Regel bleibt die Struktur zur Laufzeit gleich; Strukturunterschiede treten auf, wenn der Entwickler den Quelltext ändert. Ausnahmen bilden Listen und sich zur Laufzeit selbst verändernde Anwendungen. Abbildung 4.2 zeigt Beispiele für Strukturunterschiede.

Wertunterschiede treten immer zur Laufzeit auf. Die Anwendung berechnet einen Wert und weist ihn einer Eigenschaft zu. Dies sind immer einfache Werte: Zahlen, Buchstaben oder Wahrheitswerte. Abbildung 4.3 stellt Beispiele für Wertunterschiede dar.

4.5.3. Strukturunterschiede: Bezeichner und Attribute

XML bietet zwei gleichwertige Möglichkeiten, die Struktur eines XML-Dokuments aufzubauen. Sie lassen sich beliebig kombinieren:

1. Eigenschaften durch Bezeichner darstellen
2. Eigenschaften durch Attribute darstellen

SOLL	IST
<Person name="Susie"> <adresse>Rosenstraße 12345 Blumenstadt</adresse> <alter>17</alter> <farbe>blond</farbe> </Person>	<Person> <alter>17</alter> <adresse> <straße>Rosenstraße</straße> <plz>12345</plz> <ort>Blumenstadt</ort> </adresse> <haarfarbe>blond</haarfarbe> <name>Susie</name> </Person>

Unterschiede:

1. Reihenfolge des Bezeichners <alter> hat sich geändert: Position 1 statt 2.

2a. Der Bezeichner <adresse> hat Kindbezeichner bekommen: <straße>, <plz>, <ort>.

2b. Reihenfolge des Bezeichners <adresse> hat sich geändert: Position 2 statt 1.

3. Bezeichner <farbe> in <haarfarbe> umbenannt.

3a. Fehlendes Attribut "name" in Person.

3b. Bezeichner <name> in Person hinzugefügt

Abbildung 4.2.: Strukturunterschiede

SOLL	IST
<Person name="Susie"> <alter>17</alter> <adresse> <straße>Rosenstraße</straße> <plz>12345</plz> <ort>Blumenstadt</ort> </adresse> <haarfarbe>blond</haarfarbe> </Person>	<Person name="Susanne"> <alter>18</alter> <adresse> <straße>Rosenstraße</straße> <plz>23456</plz> <ort>Blumenstadt</ort> </adresse> <haarfarbe>brünett</haarfarbe> </Person>

Abbildung 4.3.: Wertunterschiede

Ein Attribut repräsentiert eine Eigenschaft und gehört immer zu genau einem Bezeichner. Ein Bezeichner kann beliebig viele Attribute besitzen. Bezeichner und Attribute stellen dieselbe Information unterschiedlich dar. Da beide Konzepte sich ähneln, zählen alle Attributunterschiede zu Strukturunterschieden.

4.5.4. Unterschiede in Listen

51% der 849 fehlgeschlagenen Testfälle sind Unterschiede in Listen. Eine Liste ist ein komplexer Wert; sie enthält viele einfache oder komplexe Werte (Elemente), die in der Regel alle eine ähnliche Struktur aufweisen. Unterschiede in Listen sind genau genommen Strukturunterschiede: Da aber Listen sehr häufig in Programmiersprachen vorkommen, werden sie gesondert betrachtet. Unterschiede in einer Liste treten wiederholt auf, wenn ein Element mehrmals in der Liste enthalten ist und es sich geändert hat. Dieser Unterschied geht nur einmal in die Zählung ein, da seine Häufigkeit von der Anzahl der Elemente in der Liste abhängt.

Mit 71% sind hinzugefügte oder entfernte Werte die häufigste Ursache für Unterschiede in Listen. Eine andere Reihenfolge der Elemente ist mit 29% die zweite Ursache. Besonders eine spezielle Form der Liste, die Menge, ist anfällig für eine unterschiedliche Reihenfolge: Laut Definition spielt die Reihenfolge einer Menge in der Programmiersprache Java keine Rolle, beim Vergleich des XML-Dokuments ist dies zu berücksichtigen. Abbildung 4.4 zeigt Beispiele für Unterschiede in Listen.

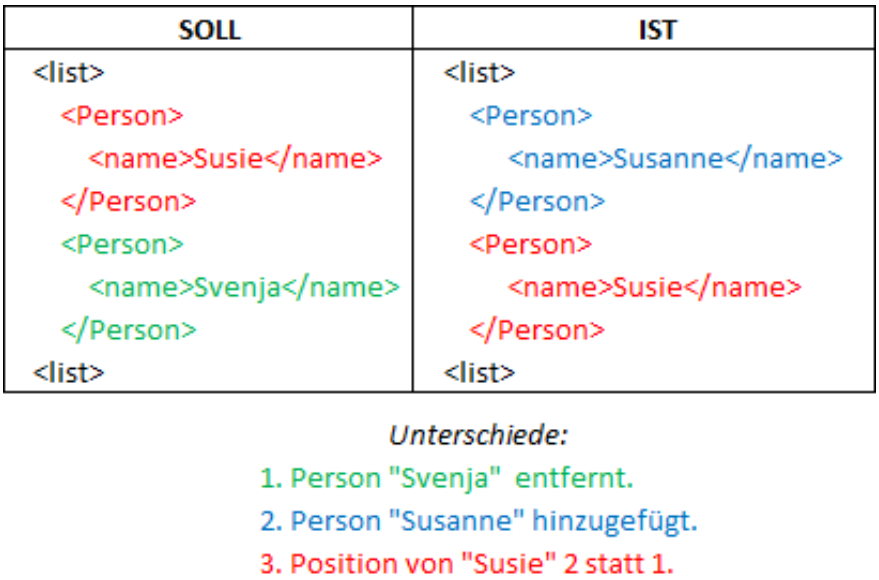

Abbildung 4.4.: Unterschiede in Listen

4.5.5. Unterschiede der Struktur

Unterschiede in der Struktur treten zu 22% in den 849 fehlgeschlagenen Testfällen auf. Hiervon sind 45% umbenannte Eigenschaften, 38% hinzugefügte Eigenschaften und 17% entfernte Eigenschaften.

Auffällig ist, dass Unterschiede an der Struktur selten, aber dann gehäuft auftreten. Dies liegt daran, dass in regelmäßigen Abständen strukturverbessernde Wartung an der Anwendung vorgenommen wird.

4.5.6. Unterschiede der Werte

Weiterhin treten von den 849 fehlgeschlagenen Testfällen unterschiedliche Werte zu 26% auf. Die Wertunterschiede sind spezifisch für die Anwendung und werden nicht genauer betrachtet. Allgemein lässt sich sagen, dass Zeitstempel häufig die Ursache sind, da ein Zeitstempel sich bei jeder erneuten Abfrage ändert.

Ein Problem bei der Stichprobe ist, dass Wertunterschiede durch veränderte Testwerte in der Testdatenbank und verändertes Programmverhalten auftreten. Die tatsächliche Ursache zu ermitteln, ist sehr aufwendig, da jeder Wertunterschied es erfordert, den Quelltext auf Änderungen zu prüfen.

4.5.7. Unterschiede der Attribute

Die Bibliothek XStream bevorzugt Bezeichner statt Attribute bei der Umwandlung eines Objekts nach XML. Prinzipiell ist jeder Bezeichner, bis auf den Wurzelbezeichner, durch ein Attribut ersetzbar. XStream verwendet nur Attribute, um ein anderes Objekt zu referenzieren. Die Struktur der Attribute hat sich in der Stichprobe nie verändert, aber ihr Wert in 1% aller fehlgeschlagener Testfälle. Der eine Grund für die Wertänderung ist eine Änderung des Objekttyps, auf den die Referenz im Attribut zeigt. Der andere Grund tritt ein, wenn man referenzierte Elemente innerhalb einer Liste verschiebt. Abbildung 4.5 zeigt hierzu ein Beispiel.

4.5.8. Fazit

Beim Vergleich von XML-Dokumenten gibt es sowohl Struktur- als auch Wertunterschiede. Strukturunterschiede treten auf durch Änderungen am Quelltext oder durch veränderte Listen. In XML stellen Attribute spezielle Bezeichner dar und gehören daher zur Struktur.

SOLL	IST
<Liste> <Buch> <titel>Moby Dick</titel> </Buch> <Buch reference="../Buch[1]"/> </Liste>	<Liste> <Buch> <titel>Der Steppenwolf</titel> </Buch> <Buch> <titel>Moby Dick</titel> </Buch> <Buch reference="../Buch[2]"/> </Liste>

1. Buch "Der Steppenwolf" zur Liste hinzugefügt.
2. Referenz auf das Buch "Moby Dick" hat sich verändert.

Abbildung 4.5.: Unterschiede bei Attributen

Wertunterschiede treten auf, wenn sich das Programmverhalten verändert hat oder der Testwert.

Die häufigsten Änderungen treten in Listen auf. Dies mag daran liegen, dass in dieser Stichprobe besonders viele Listen getestet werden. Auf der anderen Seite sind Listen äußerst instabil, weil bereits eine Änderung eines enthaltenen Werts ausreicht, damit sich die Liste von einer anderen unterscheidet. Beachtet man die Reihenfolge der Elemente, treten besonders oft Unterschiede beim Vergleich von Mengen auf. Unterschiede bei Attributen sind selten, da Bezeichner bei der Umwandlung von Objekt nach XML den Vorzug erhalten.

4.6. Vergleichen von XML-Dokumenten

4.6.1. Einleitung

In der Diplomarbeit [Falk (2005)] werden Vergleichsverfahren für XML-Dokumente ausgiebig beschrieben. Folgender Abschnitt handelt über den speziellen XML-Vergleich mit Objekten aus einer Programmiersprache.

Ein reiner Struktur- oder Wertvergleich kommt für Objekte nicht in Betracht, da sich beides ändern kann - in ungünstigen Fällen sogar gleichzeitig. Ein reiner Wertvergleich ignoriert die Bedeutung der Werte, dies führt zu Verwechslungen der Objekte: Zwei Objekte mit denselben Werten, aber unterschiedlichen Eigenschaften werden als gleich angesehen. Ein Hund sieht dann einer Katze zum Verwechseln ähnlich. Abbildung 4.6 veranschaulicht diese Situation.

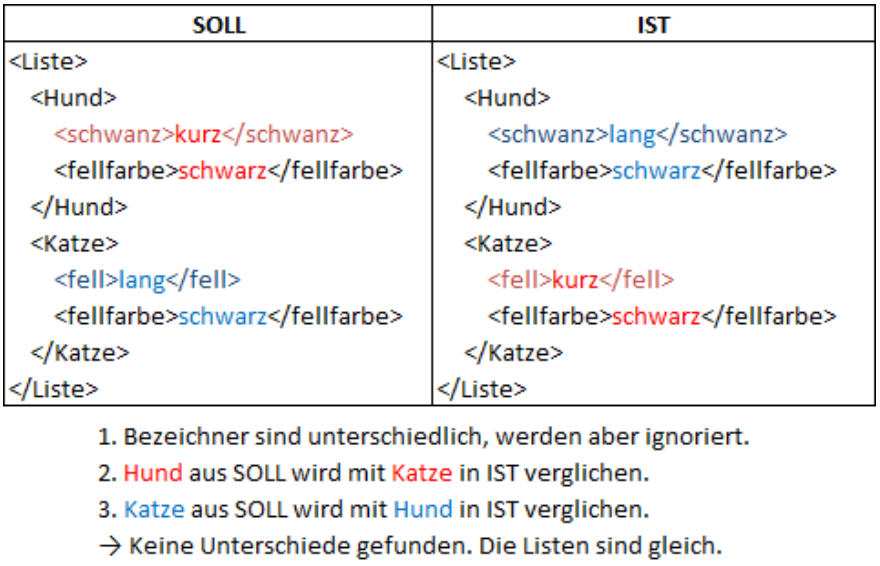

Abbildung 4.6.: Reiner Wertvergleich führt zu Verwechslungen

Ein reiner Strukturvergleich ignoriert die Werte, die ein Objekt ausmachen. Das Objekt verliert seine Identität und ist von anderen Objekten nicht mehr unterscheidbar. Ein Beispiel wäre ein Korb voll Mäuse, die alle so ähnlich aussehen, dass es unmöglich ist, eine bestimmte Maus aus dem Korb herauszunehmen. Abbildung 4.7 zeigt den Vergleich für zwei Körbe.

Ein automatisches Vergleichsverfahren für Objekte muss demnach sowohl die Struktur als auch die Werte berücksichtigen, sonst lassen sich Testfälle konstruieren, die falsche Objekte miteinander vergleichen. Werden Struktur und Werte berücksichtigt, kann es den-

SOLL	IST
<Korb> <Maus> <name>Ismael</name> </Maus> <Maus> <name>Queequeg</name> </Maus> <Maus> <name>Ahab</name> </Maus> </Korb>	<Korb> <Maus> <name>Starbuck</name> </Maus> <Maus> <name>Stubb</name> </Maus> <Maus> <name>Ismael</name> </Maus> </Korb>

1. Werte sind unterschiedlich, werden aber ignoriert.
2. Ismael wird mit Starbuck verglichen.
3. Queequeg wird mit Stubb verglichen.
4. Ahab wird mit Ismael verglichen.

→ Keine Unterschiede gefunden, Körbe sind gleich.

Abbildung 4.7.: Reiner Strukturvergleich führt zu Verlust der Identität

noch zu Verwechslungen kommen, wenn die Unterschiede zu vielseitig sind. Zum selben Schluss kommt auch die Autorin in ihrer Diplomarbeit für den allgemeinen Vergleich von XML-Dokumenten.

Die Diplomarbeit [Falk (2005)] stellt darüber hinaus Vergleichsverfahren vor, die das Schema eines XML-Dokuments verwenden, um den Vergleich zu verbessern. Beim Unit-Test liegen allerdings keine Schemata vor, weil die XML-Dokumente zur Laufzeit aus den Objekten erzeugt werden, wozu kein Schema notwendig ist. Es ist unmöglich, ein allgemeingültiges Schema für alle Objekte zu definieren, da sie zu vielfältig und zu abhängig von der Anwendung sind; und selbst wenn ein solches Schema existiert, dann wäre es zu umfassend. Aus der Diplomarbeit sind daher nur die Verfahren interessant, die kein Schema verwenden.

4.6.2. Optimistische und pessimistische Vorgehensweise

Beim Vergleich sind zwei unterschiedliche Vorgehensweisen möglich: optimistisch oder pessimistisch. Bei der optimistischen Vorgehensweise geht der Entwickler davon aus, keine Unterschiede zu finden, während er bei der pessimistischen Vorgehensweise das Gegenteil erwartet.

Eine Unterscheidung der Vorgehensweise ist hilfreich, um den Aufwand zu verringern: Es ist einfacher, zunächst optimistisch davon auszugehen, dass kein Unterschied vorliegt und auf

identische Dokumente zu prüfen. Ein optimistisches Verfahren lässt sich leicht umsetzen, indem man die Dokumente bitweise vergleicht, sodass jeder Unterschied, egal wie klein, entdeckt wird. Die Art des Unterschieds spielt dabei keine Rolle, entscheidend ist nur, ob ein Unterschied existiert. Wegen seiner Effizienz wird das optimistische Verfahren deshalb vor jedem Vergleich als erstes durchgeführt.

Stellt das optimistische Verfahren Unterschiede fest, müssen die Dokumente intensiver betrachtet werden. Hierzu dient die pessimistische Vorgehensweise, die darauf spezialisiert ist, Unterschiede zu erkennen. Auch mit einer pessimistischen Vorgehensweise ließe sich feststellen, ob überhaupt Unterschiede in den Dokumenten existieren, allerdings muss das pessimistische Verfahren dazu einen höheren Aufwand betreiben.

4.6.3. Objekterkennung bei pessimistischer Vorgehensweise

Der direkte Vergleich zweier Bezeichner allein reicht nicht aus, da der Zusammenhang verloren geht. Jeder Bezeichner steht in einen bestimmten Zusammenhang, der sich aus der Struktur des XML-Dokuments ergibt. Die Struktur hängt wiederum ab vom umgewandelten Objekt. Abbildung 4.8 veranschaulicht das Problem.

SOLL	IST
<Schiff> <name>Pequod</name> <Kapitän> <name>Ahab</name> </Kapitän> </Schiff>	<Kapitän> <name>Ahab</name> <Schiff> <name>Pequod</name> </Schiff> </Kapitän>

→ Die Objekte können zugeordnet werden, wenn die Struktur der Bezeichner berücksichtigt wird.

Abbildung 4.8.: Objekterkennung mit Berücksichtigung übergeordneter Bezeichner

Durch die Umwandlung eines Objekts ins XML-Format bleiben alle Informationen erhalten, allerdings muss das Verfahren in der Lage sein, das Objekt mitsamt seinen Eigenschaften im XML-Dokument zu erkennen, um es mit einem anderen Objekt vergleichen zu können. Beliebig verschachtelte Objekte erschweren die Erkennung.

Da im Allgemeinen beliebige XML-Dokumente mit beliebigen Objekten auftreten, gilt es ein Verfahren zu finden, das Objekte und deren Eigenschaften selbstständig erkennt. Hierzu

muss das Verfahren zu einem Bezeichner alle nachfolgenden Bezeichner untersuchen. Im XML-Dokument entspricht dies einer Tiefentraversierung.

Das Verfahren beginnt mit der Wurzel des XML-Dokuments, also dem ersten Bezeichner und allen seinen Nachkommen. Wenn im anderen XML-Dokument ein Bezeichner mit identischen oder ähnlichen Nachkommen existiert, dann werden beide Bezeichner als passendes Objekt betrachtet. Existiert jedoch zum ersten Bezeichner kein passender Bezeichner, wird der nächste Bezeichner herangezogen und wieder eine Tiefentraversierung mit dessen Nachkommen durchgeführt. Da das Verfahren mit dem obersten Bezeichner beginnt und mit dem nächst tieferen voranschreitet, ist sichergestellt, dass immer die größten, passenden Objekte gefunden werden. Ist zu einem Bezeichner im anderen Dokument kein passender Bezeichner vorhanden, wird der Bezeichner als Objekt betrachtet, das hinzugefügt oder entfernt wurde - je nachdem in welchem der beiden XML-Dokumente sich der Bezeichner befindet. Abbildung 4.9 auf Seite 43 veranschaulicht das Verfahren anhand eines vereinfachten Beispiels.

4.6.4. Suche nach vergleichbaren Objekten

Ein Vergleichsverfahren besteht aus zwei Phasen: Zuerst muss zu einem Objekt ein vergleichbares anderes Objekt gefunden werden. Anschließend werden die konkreten Unterschiede der beiden vergleichbaren Objekte ermittelt. Die Suche nach vergleichbaren Objekten ist schwieriger, da ein Objekt sich bis zur Unkenntlichkeit verändern kann. Den konkretem Unterschied zweier Objekte zu ermitteln, ist hingegen einfach, da sich die Unterschiede klassifizieren lassen und damit abzählbar sind. Ein zuverlässiges Suchverfahren ist also entscheidend für die Qualität des Vergleichsverfahren. Die Trennung zwischen Suche und Vergleich hat zudem den Vorteil, dass man jeweils ein Teilproblem betrachtet und damit die Komplexität verringert.

Das Hauptproblem bei der Suche nach vergleichbaren Objekten demonstriert die Abbildung 4.10 auf Seite 44. Ohne Kenntnisse über Änderungen am Quelltext oder der Anwendungsdaten ist nicht entscheidbar, ob die Personen ursprünglich dieselben waren und damit vergleichbar sind. Zwei Fälle sind denkbar:

1. Es existieren zwei unterschiedliche Personen mit dem Namen “Susie”, die an unterschiedlichen Orten wohnen.
2. Es existiert eine Person mit dem Namen “Susie”, die vor kurzem umgezogen ist.

Je nach Interpretation fällt das Ergebnis des Vergleichs unterschiedlich aus. Das hier entwickelte Verfahren triff die Entscheidung anhand der in Abschnitt 4.5 ermittelten Häufigkeiten.

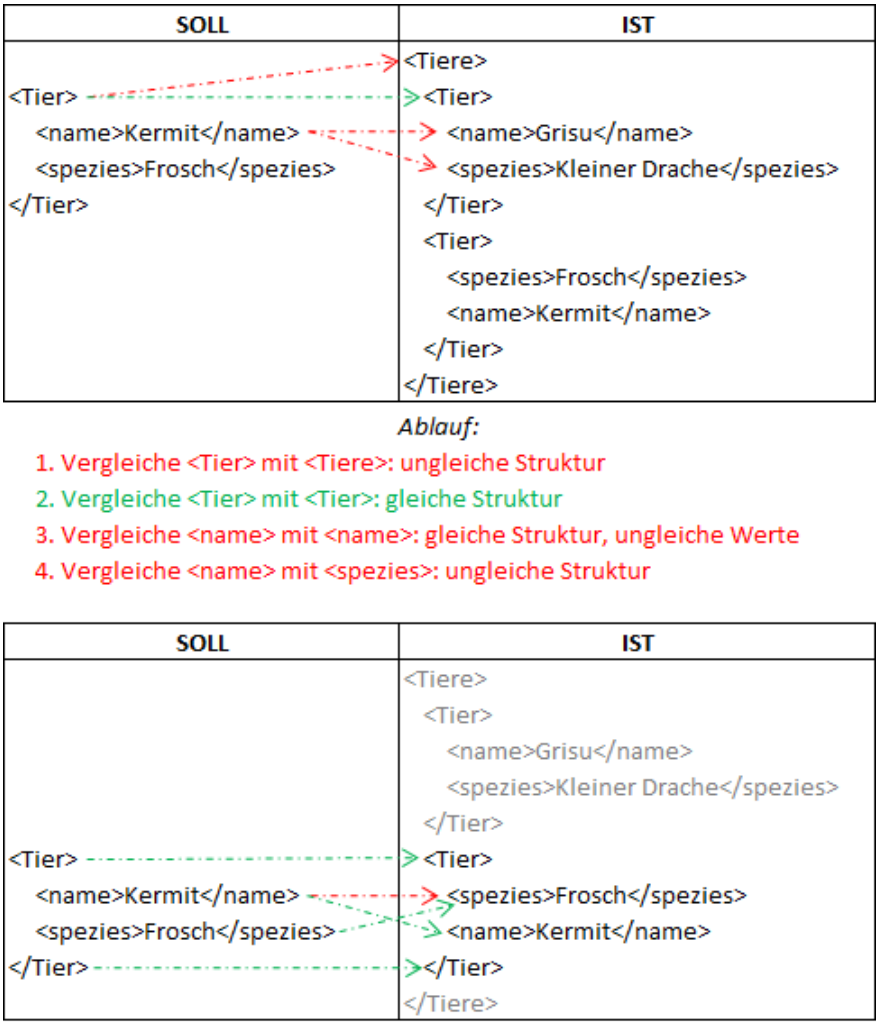

Abbildung 4.9.: Ablauf der Objekterkennung im XML-Dokument mittels Tiefentraversierung

SOLL	IST
<Liste> ... <Person> <name>Susie</name> <alter>17</alter> <adresse> <straße>Rosenstraße</straße> <plz>12345</plz> <ort>Blumenstadt</ort> </adresse> </Person> ... </Liste>	<Liste> ... <Person> <name>Susie</name> <alter>17</alter> <adresse> <straße>Nelkenstraße</straße> <plz>54321</plz> <ort>Tulpenstadt</ort> </adresse> </Person> ... </Liste>

1. Adresse unterscheidet sich.
2. Persondaten sind gleich.
→ Problem: Sind es vergleichbare Personen?

Abbildung 4.10.: Vergleichbare Objekte ohne Hintergrundinformation erkennen

4.6.5. Ähnlichkeitsbestimmung durch Suchverfahren

Um ein vergleichbares Objekt zu finden, werden Struktur und Werte zweier Bezeichner herangezogen. Ist bereits deren Struktur unterschiedlich, ist die Wahrscheinlichkeit hoch, dass es keine vergleichbaren Objekte sind. Allerdings existiert die Möglichkeit, dass sich der Objekttyp oder die Eigenschaft geändert hat. Die Ursache lässt sich nicht sicher feststellen, dazu sind Kenntnisse über die durchgeführten Änderungen am Quellcode nötig. Aufgrund der Analyse in Abschnitt 4.5 geht das Vergleichsverfahren davon aus, dass sich Werte regelmäßig ändern und die Struktur gleich bleibt.

Existiert kein passender Bezeichner im anderen Dokument, setzt die Suche mit dem nachfolgenden Bezeichner fort. Sind zwei Bezeichner vergleichbar, werden alle ihre Nachkommen geprüft. Sind ihre Nachkommen und deren Nachkommen vergleichbar, ist der ursprüngliche Bezeichner ein Objekt, zu dem ein vergleichbares anderes Objekt gefunden wurde. Das Verfahren merkt sich alle beteiligten Bezeichner und schließt sie von der weiteren Suche aus, damit gibt es zu jedem Objekt genau ein vergleichbares Objekt.

Es bleiben die Bezeichner übrig, denen kein vergleichbarer anderer Bezeichner zugeordnet werden kann. Hierfür sind zwei Ursachen möglich:

1. Zwei Bezeichner waren ursprünglich gleich, wurden aber aufgrund großer Änderungen nicht als vergleichbar erkannt.
2. Bezeichner wurden in den beiden XML-Dokumenten hinzugefügt oder entfernt.

Die genaue Ursache zu ermitteln ist schwierig, da wiederum Kenntnisse über die Änderungen erforderlich sind.

Aufgrund der Analyse in Abschnitt 4.5 sind hinzugefügte oder entfernte Elemente aus einer Liste die häufigste Ursache, daher geht das Verfahren von diesem Fall aus.

Reihenfolge der Bezeichner

Ist die Reihenfolge der Bezeichner von Bedeutung, wird nur der Bezeichner im anderen Dokument betrachtet, der sich an derselben relativen Position zur Wurzel befindet. Der Aufwand entspricht nach der Groß-O-Notation $O(n)$: Jeder Bezeichner im Dokument wird einmal überprüft. Abbildung 4.11 zeigt einen Vergleich mit Beachtung der Reihenfolge.

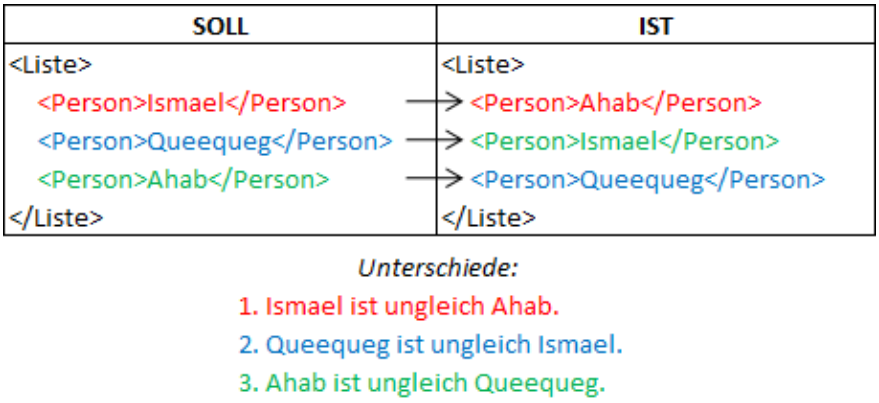

Abbildung 4.11.: Vergleich mit Beachtung der Reihenfolge

Spielt die Reihenfolge keine Rolle, muss das Verfahren jeden Bezeichner mit jedem Bezeichner im anderen Dokument vergleichen, bis ein passender Bezeichner gefunden wird. Im Idealfall beträgt der Aufwand $O(n)$, wenn die Reihenfolge identisch ist. Im schlechtesten Fall wird jeder Bezeichner mit jedem anderen übrig gebliebenen Bezeichner verglichen, dies entspricht einem Aufwand von $O(n*(n-i))$, zusammengefasst $O(n^2)$. Der schlechteste Fall liegt vor, wenn der passende Bezeichner sich immer an der letzten durchsuchten Position im anderen Dokument befindet. Abbildung 4.12 stellt den Ablauf dar.

Alternativ kann man die Reihenfolge der Bezeichner ignorieren, indem beide XML-Dokumente sortiert werden. Alle Bezeichner werden nach dem Alphabet sortiert, wobei die Tiefenstruktur erhalten bleibt: Ein Bezeichner hat nach der Sortierung dieselben Nachkommen wie vor der Sortierung. Anschließend werden gleiche Bezeichner auf derselben Ebene nach ihrem Wert sortiert. Sind beide XML-Dokumente sortiert, befinden sich die passenden Bezeichner immer an gleicher Position relativ zur Wurzel - der Aufwand für die Suche nach passenden Bezeichnern entspricht dann $O(n)$. Das Problem der Reihenfolge

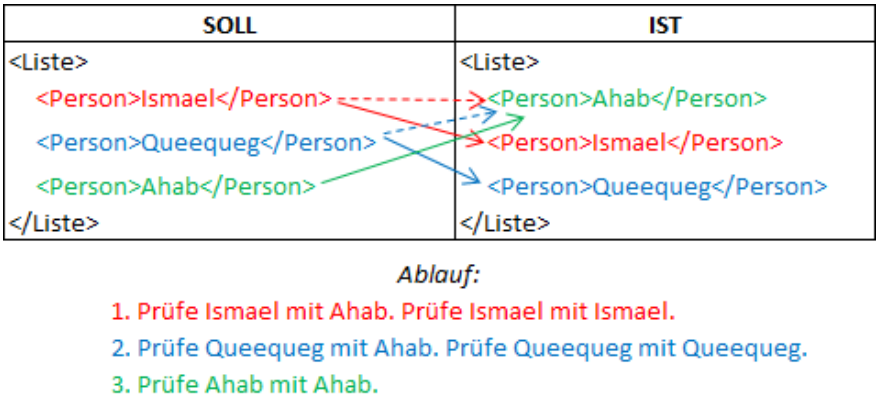

Abbildung 4.12.: Vergleich ohne Beachtung der Reihenfolge

ist auf diese Weise effizient gelöst, allerdings dürfen keine Bezeichner hinzugefügt oder entfernt werden, da sonst die Zuordnung mittels der Position verloren geht. Auch unglückliche Wertänderungen können dies bewirken, wenn dadurch die Reihenfolge beeinflusst wird. Dies hat zur Folge, dass trotz Sortierung immer noch passende Bezeichner gesucht werden müssen. Abbildung 4.13 veranschaulicht das Problem anhand eines Beispiels. Es bleibt zu prüfen, ob eine Vorsortierung den Aufwand für die Suche erleichtert.

4.6.6. Auflösen der Objektreferenzen

Unter bestimmten Umständen kann die Umwandlung von Objekten ins XML-Format dazu führen, dass der Vergleich Unterschiede findet, obwohl die Objekte gleich sind. Die Ursache ist, dass im XML-Dokument die Objektrefenzen abhängig von der Reihenfolge sind. Das Problem tritt auf, wenn Listen in unterschiedlicher Reihenfolge vorliegen und ein Objekt mehrmals darin enthalten ist. Abbildung 4.14 veranschaulicht das Problem.

Die Lösung lautet, das referenzierte Objekt statt die Referenz während des Vergleichs zu prüfen.

4.6.7. Stufenweises Vergleichsverfahren

In Abschnitt 4.5 Analyse hat sich herausgestellt, dass einige Unterschiede besonders häufig auftreten. Anhand der ausgewerteten Daten ensteht ein stufenweises Vergleichsverfahren. Ziel des stufenweisen Verfahrens ist es, bestimmte Unterschiede zu identifizieren, die besonders häufig auftreten. Der Entwickler erhält daraufhin eine vorgefertigte Nachricht, in der

SOLL (unsortiert)	IST (unsortiert)
<Liste> <Person>Ismael</Person> <Person>Queequeg</Person> <Person>Ahab</Person> </Liste>	<Liste> <Person>Queequeg</Person> <Person>Ismael</Person> <Person>Ahab</Person> <Person>Flask</Person> </Liste>

↓

SOLL (sortiert)	IST (sortiert)
<Liste>	<Liste>
<Person>Ahab</Person> →	<Person>Ahab</Person>
<Person>Ismael</Person> →	<Person>Flask</Person>
<Person>Queequeg</Person> →	<Person>Ismael</Person>
</Liste>	<Person>Queequeg</Person>
	</Liste>

Ergebnis:

1. Ahab ist gleich Ahab.
2. Ismael ist ungleich Flask.
3. Queequeg ist ungleich Ismael.
4. Queequeq wurde hinzugefügt?

Abbildung 4.13.: Vergleich mit Vorsortierung

SOLL	IST
<Liste> <Person> <name>Susie</name> <Adresse> <ort>Blumenstadt</ort> </Adresse> </Person> <Person> <name>Susanne</name> <Adresse reference="../../Adresse[1]"/> </Person> </Liste>	<Liste> <Person> <name>Susanne</name> <Adresse> <ort>Blumenstadt</ort> </Adresse> </Person> <Person> <name>Susie</name> <Adresse reference="../../Adresse[1]"/> </Person> </Liste>

Unterschiede:

1. <Adresse> ist ungleich <Adresse reference="../../Adresse[1]"/>
2. <Adresse reference="../../Adresse[1]"/> ist ungleich <Adresse>

Abbildung 4.14.: Schwieriger Vergleich durch Objektreferenzen

er über den Typ des entdeckten Unterschieds benachrichtigt wird. Dies soll dem Entwickler helfen, den Unterschied einzuordnen und dadurch schneller zu entscheiden, ob er von Bedeutung ist.

Als erstes prüft ein optimistisches Vergleichsverfahren, ob Unterschiede in beiden XML-Dokumenten vorhanden sind. Ist dies der Fall, werden in der ersten Stufe ausschließlich Strukturunterschiede gesucht. Ist die Struktur gleich, weiß der Entwickler, dass am Quelltext keine Änderungen vorgenommen wurden. In der zweiten Stufe werden die Werte auf Unterschiede untersucht. Sind alle Werte gleich, dann weiß der Entwickler, dass keine Änderung am Programmverhalten vorliegt. In der dritten Stufe wird die Reihenfolge der Objekte überprüft. Unterschiede treten häufig auf, weil Mengen verwendet werden oder eine Datenbankabfrage das Ergebnis in einer anderen Reihenfolge zurückliefert. In der vierten Stufe werden schließlich Attribute auf Änderungen geprüft. Dieser Fall tritt selten ein und bedeutet in der Regel, dass im Quelltext eine Klasse umbenannt wurde.

4.6.8. Fazit

Da die XML-Dokumente Objekte einer Programmiersprache enthalten, muss der Zusammenhang bei jedem Bezeichner betrachtet werden. Durch eine Tiefentraversierung wird das gesamte Objekt mit seinem Eigenschaften berücksichtigt.

Es gibt zwei Vorgehensweisen beim Vergleich: die optimistische, die keine Unterschiede erwartet und die pessimistische, die Unterschiede erwartet. Die pessimistische Vorgehensweise erfordert einen höheren Aufwand, findet aber die konkreten Unterschiede.

Um die Komplexität zu verringern, werden Such- und Vergleichverfahren getrennt. Das Suchverfahren hat die Aufgabe, zu einem gegebenen Objekt das passende - eventuell veränderte - Objekt zu finden. Die passenden Objekte werden anschließend verglichen.

Die Qualität des Vergleichverfahrens hängt entscheidend von der Güte des Suchverfahrens ab. Allerdings ist nicht immer entscheidbar, ob zwei Objekte vergleichbar sind: Die Antwort hängt von der Interpretation der Unterschiede ab.

Ein stufenweises Vergleichsverfahren hilft den Entwickler, typische Unterschiede schnell festzustellen, indem in einer bestimmten Reihenfolge nach Unterschieden gesucht wird.

4.7. Suchverfahren zur Ähnlichkeitsbestimmung

4.7.1. Einleitung

Es gibt verschiedene Möglichkeiten, die Suche nach vergleichbaren Objekten durchzuführen; sie werden nachfolgend vorgestellt. Weitere Varianten sind möglich: Die hier vorgestellten Verfahren stellen eine Auswahl dar, die umgesetzt, getestet und bewertet werden. Der Quelltext für die Verfahren befindet sich im Anhang A ab Seite 74.

4.7.2. Gemischtes-Suchverfahren

Wie bereits beschrieben, ist ein reiner Wert- oder Strukturvergleich problematisch. Das Gemischte-Suchverfahren kombiniert beide Ansätze. Zunächst arbeitet es wie ein reiner Strukturvergleich: Es prüft die Struktur zweier Bezeichner und aller Nachkommen. Sobald das Verfahren auf einen Bezeichner trifft, der keine Nachkommen besitzt, handelt es sich um ein Blatt in einem Baum. Der Wert im Blatt wird mit dem Blatt des anderen Baumes verglichen. Vorbedingung ist, dass der Pfad zum Blatt in beiden Bäumen identisch ist, also gleiche Äste zu denselben Blätter führen.

Das Verfahren funktioniert, wenn die Struktur und die Werte unverändert bleiben, also wenn die Reihenfolge unverändert bleibt und keine Bezeichner entfernt oder hinzugefügt werden. Passende Objekte werden nur gefunden, wenn sie identisch sind. Treten jedoch Unterschiede auf, kann das Verfahren nicht mit Sicherheit sagen, welches Objekt sich verändert hat.

Das Verfahren ist geeignet, wenn folgende Bedingungen erfüllt sind (je mehr desto besser):

- die Reihenfolge bleibt unverändert.
- die Struktur bleibt unverändert.
- die Werte bleiben unverändert.

Diese Bedingungen können bei pessimistischer Herangehensweise nicht vorausgesetzt werden, daher sind weitere Verfahren notwendig.

4.7.3. Schlüsselwert-Suchverfahren

Einige Listen erfordern eine Schlüsseleigenschaft im Objekt, um das Objekt eindeutig innerhalb der Liste zu identifizieren. Zum Beispiel haben Bücher als Schlüssel die ISBN, mit der ein Buch weltweit eindeutig bestimmbar ist. Es können auch mehrere Eigenschaften zusammen einen Schlüssel bilden, wie beispielsweise der Vor- und Nachname bei Schülern in einer Klasse. Der Schlüssel muss in seinem Umfeld immer eindeutig sein: Die Suche darf als Ergebnis nur ein Objekt zurückliefern. Wird diese Bedingung nicht erfüllt, müssen eventuell weitere Eigenschaften zum Schlüssel hinzugefügt werden. Mithilfe des Schlüssels lassen sich veränderte Objekte eindeutig zugeordnen.

Das Hauptproblem ist, den Schlüssel automatisch zu erkennen. XML bietet zwar die Möglichkeit Bezeichner über das Schema als Schlüssel zu markieren, jedoch muss das Suchverfahren ohne Schema auskommen und die Schlüssel nach der Umwandlung des Objekts selbstständig erkennen. Dies ist ohne Kenntnisse der Anwendung nicht allgemein möglich. Der Entwickler muss aushelfen und dem Suchverfahren die Information geben.

4.7.4. Schwellwert-Suchverfahren

Wenn die Unterschiede zwischen dem ursprünglichen Objekt und dem veränderten Objekt so groß sind, dass es nicht mehr entscheidbar ist, ob sie ursprünglich dasselbe Objekt waren, dann helfen Wahrscheinlichkeiten. Ein einfacher Ansatz ist es, den prozentualen Anteil der Übereinstimmung zweier Objekte zu nehmen. Angenommen ein Gast bestellt beim Wirt eine Flasche Wodka. Um die gewünschte Flasche auszuwählen, prüft der beschwipste Wirt per Augenmaß den Füllstand, die Marke und die Brüchigkeit der Flasche.

- Die erste Flasche ist voll, von der Marke “Wodka Gorbatschow” und hat keine Risse.
- Die zweite Falsche ist voll, von der Marke “Molotov Cocktail” und hat keine Risse.

Zwei der drei Merkmale stimmen überein, für den getrübten Blick des Barkeepers sind somit beide Flaschen gleich - womöglich sieht er aber auch nur doppelt. Deshalb wählt er zufällig eine Flasche aus.

Natürlich war es die verkehrte Flasche, woraufhin eine wilde Schlägerei ausbricht und einer der Gäste die Flasche der Marke “Molotov Cocktail” als verlängten Arm einsetzt; die Flasche erhält daraufhin eine große Kerbe. Nachdem wieder etwas Ruhe eingekehrt ist, wollen sich die Gäste bei einem Cocktail versöhnen und bestellen folglich. Der Wirt betrachtet erneut beide Flaschen:

- Die erste Flasche ist voll, von der Marke “Wodka Gorbatschow” und hat keine Risse.
- Die zweite Flasche ist voll, von der Marke “Molotov Cocktail” und hat jetzt Risse.

Er stellt fest, dass nur eines der drei Merkmale überstimmt. Er beschließt daraufhin, dass es sich um unterschiedliche Flaschen handelt und schaut sich deshalb die Flaschen genauer an, bevor er ausschenkt.

Beim Schwellwert-Suchverfahren wird ein Schwellwert bestimmt, der eine feste Grenze darstellt. Der Wirt hat in dem Beispiel einen Schwellwert von 50%: Zwei der drei Eigenschaften müssen übereinstimmen, damit der Wirt von gleichen Flaschen ausgeht. Mögliche Ergebnisse sind:

1. Der Schwellwert wird überschritten und beide Flaschen sind vergleichbar.
2. Der Schwellwert wird unterschritten und beide Flaschen sind nicht vergleichbar.
3. Der Schwellwert wird genau getroffen und Fall 1 oder Fall 2 wird angewendet.

Das Problem des Schwellwertverfahrens ist es, den richtigen Schwellwert zu finden. Er hängt vom Testfall ab.

Im ersten Testfall wird von wenigen Änderungen ausgegangen, der Schwellwert ist entsprechend hoch auf 80% gesetzt. Dennoch lässt sich ein Testfall konstruieren, bei dem die Suche scheitert, wie Abbildung 4.15 demonstriert. Der Schlüssel <Artikelnummer> kennzeichnet eindeutig einen Artikel. Wenn sich zwei Artikelnummern unterscheiden, dann sind die Artikel nicht vergleichbar.

SOLL	IST
<Artikelliste> <artikelnummer>4711</artikelnummer> <Wodka> <marke>Gorbatschow</marke> <hersteller>Henkell & Söhnlein KG</hersteller> <alkoholgehalt>50%</alkoholgehalt> <liter>0,7</liter> <farbe>Schwarz</farbe> </Wodka> </Artikelliste>	<Artikelliste> <artikelnummer>1983</artikelnummer> <Wodka> <marke>Gorbatschow</marke> <hersteller>Henkell & Söhnlein KG</hersteller> <alkoholgehalt>50%</alkoholgehalt> <liter>0,7</liter> <farbe>Schwarz</farbe> </Wodka> </Artikelliste>

Ablauf:

1. Nur die Artikelnummern sind unterschiedlich.
2. Schwellwert beträgt 80%.
3. Übereinstimmung: 5 von 6 Werten sind gleich, das macht 83,3%.
4. Übereinstimmung > Schwellwert.
5. Die Artikel sind vergleichbar.

Abbildung 4.15.: Suche nach vergleichbaren Objekten mit hochem Schwellwert

Im zweiten Testfall wird ein niedriger Schwellwert von 20% gewählt, weil mit vielen Änderungen gerechnet wird oder weil ein Schlüssel ein Objekt eindeutig identifiziert. Abbildung 4.16

demonstriert diesen Fall. Dort werden fälscherlicherweise beide Artikel mit derselben Artikelnummer als nicht vergleichbar erkannt, weil die Änderungen zu groß sind.

SOLL	IST
<Artikelliste> <artikelnummer>4711</artikelnummer> <Wodka> <marke>Gorbatschow</marke> <hersteller>Henkell & Söhnlein KG</hersteller> <alkoholgehalt>50%</alkoholgehalt> <liter>0,7</liter> <farbe>Schwarz</farbe> </Wodka> </Artikelliste>	<Artikelliste> <artikelnummer>4711</artikelnummer> <Wodka> <marke>Moskovskaya</marke> <hersteller>Simex GmbH & Co KG</hersteller> <alkoholgehalt>40%</alkoholgehalt> <liter>0,5</liter> <farbe>Grün</farbe> </Wodka> </Artikelliste>

Ablauf:

1. Nur die Artikelnummern sind gleich.
2. Schwellwert beträgt 20%.
3. Übereinstimmung: 1 von 6 Werten sind gleich, das macht 16,6%.
4. Übereinstimmung < Schwellwert.
5. Die Artikel sind nicht vergleichbar.

Abbildung 4.16.: Suche nach vergleichbaren Objekten mit niedrigem Schwellwert

Das Schwellwert-Suchverfahren lässt sich sinnvoll einsetzen, wenn detailierte Kenntnisse über die Änderungen bekannt sind. Die Änderungen hängen vom Testfall ab. Benötigt wird eine Heuristik, die automatisch eine Eigenschaft als Schlüssel erkennt.

4.7.5. Fazit

Ziel war es, ein autonomes Suchverfahren zu entwickeln, das selbstständig veränderte Objekte erkennt. Dies ist leider äußerst schwierig, da dem Suchverfahren in vielen Fällen die Änderungen bekannt sein muss, damit es die richtigen Schlüsse ziehen kann. Drei entwickelte Suchverfahren zeigen, wie sich die Probleme in der Praxis auswirken. Schwierigkeiten bereiten vor allem Änderungen der Werte, der Struktur durch Hinzufügen oder Entfernen von Bezeichnern und eine veränderte Reihenfolge. Je zahlreicher die Änderungen, desto schwieriger die Suche.

Damit ein Suchverfahren korrekt arbeitet, muss es das gesamte Umfeld berücksichtigen. Dies steht mit dem Ziel in Konflikt, ein automatisches Verfahren zu entwickeln, das ohne Hilfe des Entwicklers auskommt. Entweder es wird ein spezielles Verfahren entwickelt, das genau auf eine Anwendung zugeschnitten ist oder ein allgemeines Verfahren, das nicht korrekt arbeitet. Da das Ziel ein allgemeingültiges Verfahren ist, muss auf die Korrektheit verzichtet werden.

Tatsächlich hat sich das Gemischte-Suchverfahren mit Tiefentraversierung zur Objekterkennung als bestes Suchverfahren herausgestellt, obwohl es in einigen - aber wenigen - Fällen falsche Ergebnisse liefert. Dennoch garantiert das Verfahren, unterschiedliche Dokumente zu erkennen. So hat der Entwickler zumindest einen Anhaltspunkt, um den Unterschieden auf den Grund zu gehen. Zu diesem Zweck muss der Entwickler auf ein Werkzeug zurückgreifen, das XML-Dokumente gegenüberstellt.

4.8. Evaluation

4.8.1. Einleitung

Dieser Abschnitt bewertet zunächst die umgesetzten Maßnahmen im Prototyp. Herkömmliche Unit-Tests werden herangezogen, um die Verbesserungen durch den Prototyp zu ermitteln. Anschließend werden die Bedingungen genannt, die für einen sinnvollen Einsatz des Prototyps sprechen.

4.8.2. Unit-Test

Aufwandsersparnis beim Unit-Test

Ein Unit-Test läuft immer auf dasselbe hinaus: Vergleiche den tatsächlichen Wert mit dem erwarteten Wert. Stimmen sie überein, ist der Test erfolgreich, falls nicht, liegt ein Softwarefehler vor.

Damit ein Objekt vergleichbar wird, muss der Entwickler eine Vergleichsfunktion schreiben, in der die Gleichheit des Objekts definiert wird. Angenommen der Entwickler schreibt einen Test für den Hamburger Hafen, um Container zu vergleichen. Um einen Container zu identifizieren, muss ein Hafenarbeiter den Container öffnen und den Inhalt mit einer Bestandsliste prüfen - bei mehreren Millionen Schrauben eine langwierige Angelegenheit. Ein Entwickler geht wie der Hafenarbeiter vor: Er veranlasst den Computer, den Inhalt eines Containers mit dem gesuchten Container zu vergleichen. Ein Container kann dabei mehrere Kartons enthalten, die wiederum Verpackungen enthalten, bis hin zum tatsächlichen Produkt. Die Konsequenz für den Entwickler ist, für jeden Artikel im Container eine Vergleichsfunktion zu schreiben, was nicht weniger aufwendig ist, als die Aufgabe des Hafenarbeiters.

Solche Unit-Tests erfordern einen hohen Aufwand, der keine besonderen Fertigkeiten oder Kenntnisse voraussetzt, sondern das mechanische Abarbeiten von ähnlichen Vergleichsfunktionen. Typischerweise werden zwei Objekte als gleich betrachtet, wenn sie denselben Zustand haben, also alle ihre Eigenschaften dieselben Werte besitzen. Folglich prüft die Vergleichsfunktion jede Eigenschaft. Ist eine Eigenschaft ein Objekt, so ist ein weiterer Vergleich für dieses Objekt durchzuführen. Dies kann sich für komplexe Objekthierarchien immer tiefer fortsetzen. In den meisten Projekten existieren solche Objekte, die als reine Datenhalter dienen. Sie zeichnen sich durch viele Eigenschaften aus, aber besitzen keine Anwendungslogik. Ein Unit-Test mit solchen Objekten ist besonders geeignet für den Prototyp, da mit einer einzigen Zeile Code, viele Zeilen für Vergleichsfunktionen gespart werden.

Folgendes Beispiel nimmt an, dass der Container elektronische Artikel enthält: Fernseher, Mikrowellen (das Haushaltsgerät) und Waschmaschinen.

In einem herkömmlichen Unit-Test schreibt der Entwickler zuerst die Vergleichsfunktionen:

1. Schreibe Vergleichsfunktion für Fernseher
    a) Schreibe Vergleichsfunktion für Hersteller
        i. Vergleiche Herstellername
        ii. Vergleiche Produktbezeichnung
        iii. Schreibe Vergleichsfunktion für Firmensitz
            A. Vergleiche Stadt
            B. Vergleiche Postleitzahl
            C. Vergleiche Straße
            D. Vergleich Hausnummer
2. Schreibe Vergleichsfunktion für Mikrowellen
    a) Vergleiche Seriennummer
3. Schreibe Vergleichsfunktion für Waschmaschinen
    a) Schreibe Vergleichsfunktion für Hersteller (siehe 1a)
    b) Schreibe Vergleichsfunktion für Waschfunktionen
        i. Vergleiche Unterstützung für Trockenwäsche
        ii. Vergleiche Unterstützung für Schaumwäsche

Anschließend schreibt der Entwickler den Unit-Test:

```
// 1. Container erzeugen
// 2. Container mit Vergleichsfunktion vergleichen:
assertEquals(containerSOLL, containerIST);
```

Listing 4.6: Herkömmlicher Unit-Test

Derselbe *vollständige* Testfall mit XmlDiff lautet wie folgt:

```
// 1. Container erzeugen
// 2. Container ohne Vergleichsfunktion vergleichen:
assertNoDifferences(containerSOLL, containerIST);
```

Listing 4.7: Unit-Test mit XmlDiff

Das Beispiel verdeutlicht den Aufwand für einen herkömmlichen Unit-Test. Mit XmlDiff sind keine Vergleichsfunktionen notwendig. Damit bleibt der Testfall übrig, der nahezu identisch zum herkömmlichen Testfall ist.

Der ersparte Aufwand ist berechenbar: Angenommen für ein Objektvergleich sind zehn Vergleichsfunktionen erforderlich. Jedes Objekt hat im Schnitt zehn Eigenschaften. Für jede Eigenschaft sind etwa vier Zeilen Vergleichscode erforderlich. Daraus ergibt sich eine Ersparnis von 10 Objekte $*$ 10 Attribute $*$ 4 Zeilen $=$ 400 Zeilen. Weiter angenommen ein Entwickler schreibt 10 Zeilen pro Minute, dann ergibt sich eine Zeitersparnis von 400 Zeilen $/10$ Zeilen pro Minute $=$ 40 Minuten.

Nutzen außerhalb von Unit-Test

XmlDiff kann auch für einen anderen Zweck außerhalb von Unit-Tests eingesetzt werden, um den Programmfluss zu steuern. Erst wird eine Bedingung geprüft, dann wird abhängig vom Ergebnis entweder der eine oder der andere Programmpfad ausgeführt:

```
wenn objekt1 gleich objekt2
dann führe Programmpfad 1 aus
sonst führe Programmpfad 2 aus
```

Listing 4.8: Programmfluss mit Bedingungen steuern

XmlDiff übernimmt den Vergleich, der sonst vom Entwickler in einer Vergleichsfunktion bereitgestellt wird. Diese Funktion des Prototyps hat sich bisher nicht durchgesetzt. Der Grund dafür ist, dass die Entscheidung für den Programmpfad stark von der Anwendung abhängt: Der Vergleich enthält in der Regel Anwendungslogik, die nicht automatisiert erzeugt wird.

4.8.3. Regressionstest

Im Regressionstest wird ein Sollzustand mit einem Istzustand verglichen, mit dem Ziel keine Unterschiede zu finden. Soll- und Istzustand werden aus demselben Objekt entnommen, aber zu verschiedenen Zeitpunkten: Der Sollzustand ist älter als der Istzustand.

Regressionstests können unterteilt werden, indem man die Erzeugung des Sollzustands berücksichtigt. Nachfolgend wird zwischen dauerhaften und flüchtigen Regressionstests unterschieden.

Dauerhafter Regressionstest

Ein dauerhafter Regressionstest sichert den Sollzustand auf einem Datenträger. In der Zwischenzeit werden Änderungen an der Anwendung vorgenommen, die mehrere Stunden, Tage oder Wochen andauern können. Nachdem die Arbeiten erledigt sind, wird die veränderte Anwendung erneut ausgeführt und man erhält einen Istzustand. Der Vergleich zwischen Soll- und Istzustand stellt jetzt sicher, dass das Programmverhalten trotz Änderungen gleich geblieben ist.

Ziel ist die strukturverbessernde Wartung der Anwendung (Englisch: “refactoring”), ohne Änderungen am Verhalten oder der Funktionalität vorzunehmen - ein Regressionstest wäre sonst ungeeignet, da Soll- und Istzustand dann nicht vergleichbar wären.

Angenommen der Entwickler schreibt einen dauerhaften Regressionstest für eine Webseite. Anwender haben die Möglichkeit, sich auf der Webseite anzumelden und ein Profil von sich zu erstellen. Aufgrund der großen Beliebtheit der Seite und des Besucheransturms leidet die Ladezeit der Webseite enorm. Der Entwickler hat ermittelt, dass die Ladefunktion der Profile aus der Datenbank für die Wartezeiten verantwortlich ist. Er beschließt die Funktion umzuschreiben, um die Webseite zu beschleunigen. Die neue Funktion soll die Profile wie die alte Funktion aus der Datenbank holen, nur schneller. Damit sich bei der Änderung keine Fehler einschleichen, beschließt er, einen Regressionstest zu schreiben.

Bevor der Entwickler Änderungen am Programm vornimmt, muss er dafür sorgen, dass er den Sollzustand wiederherstellen kann, weil dieser für den Regressionstest benötigt wird. Zwei Vorgehensweisen stehen zur Wahl:

1. Die Profile (Sollzustand) dauerhaft sichern
2. Die alte Funktion unverändert beibehalten

In der ersten Alternative wird der Sollzustand mithilfe der alten Funktion erzeugt und anschließend dauerhaft gesichert. Daraufhin wird die alte Funktion überarbeitet.

In der zweiten Alternative bleibt die alte Funktion unverändert erhalten, damit sie jederzeit aufgerufen werden kann, um neue Sollzustände zu erzeugen. Die neue Funktion zum Auslesen der Datenbank wird unabhängig entwickelt und der Anwendung hinzugefügt.

Bewertet man die Alternativen, so hat die zweite den Vorteil, dass bei Schwierigkeiten in der neuen Funktion weiterhin auf die alte Funktion zurückgegriffen werden kann. Außerdem können jederzeit beliebige Sollzustände erzeugt werden.

Anderseits kann es unerwünscht sein, dass die Anwendung veraltete Funktionen enthält. Unter Umständen ist es einfacher, die alte Funktion abzuändern, anstatt die neue Funktion neu einzubauen.

Der Entwickler entscheidet sich für die erste Alternative. Nachdem der Sollzustand gespeichert ist, kann er den herkömmlichen Regressionstest schreiben:

1. Lade SOLL-Profile von einem Datenträger
2. Lade IST-Profile mit neuer Funktion aus der Datenbank
3. Vergleiche SOLL-Profile aus Schritt 1 mit IST-Profile aus Schritt 2

Schritt 1 setzt voraus, dass die Profile zuvor einmalig mit der alten Funktion erzeugt wurden. Die alte Funktion wird daraufhin umgebaut. Sobald die Arbeit abgeschlossen ist, führt der Entwickler den Regressionstest durch und prüft, ob das Verhalten der Webseite dasselbe ist, wie zuvor.

Der Prototyp erleichtert das Verfahren, indem der Schritt 1 für die einmalige Erzeugung des Sollzustandes wegfällt. Außerdem übernimmt der Prototyp automatisch die Verwaltung der Dateien für die Sollzustände. Der Entwickler muss keine Funktionen zum Speichern und Laden zur Verfügung stellen:

1. Lade IST-Profile mit neuer Funktion aus der Datenbank
2. Vergleiche SOLL-Profile mit IST-Profile aus Schritt 1

In Schritt 2 sorgt der Prototyp dafür, dass die SOLL-Profile automatisch geladen werden, indem der Entwickler eine Identifikation für den gewünschten Sollzustand angibt. Der Aufwand für die Verwaltung der Sollzustände entfällt.

Flüchtiger Regressionstest

Der flüchtige Regressionstest findet innerhalb eines Testlaufs statt. Während des Testlaufs wird nachgewiesen, dass ein Objekt zustandslos ist: Durch den Aufruf einer Funktion bleibt der Zustand des Objekts unverändert. Die erste Zustandsaufnahme findet vor dem Funktionsaufruf statt, der zweite danach. Anschließend werden sie verglichen. Einen weiteren Anwendungsfall für einen flüchtigen Regressionstest zeigt das Webseiten-Beispiel, bei dem der Zustand mit der alten und der neuen Funktion erzeugt wird:

1. Lade SOLL-Profile mit der alten Funktion aus der Datenbank
2. Lade IST-Profile mit der neuen Funktion aus der Datenbank
3. Vergleiche beide Profile

Der Prototyp erleichtert den flüchtigen Regressionstest nicht, aber übernimmt weiterhin den automatischen Vergleich der Testwerte.

4.8.4. Trennung von Anwendung und Test

In der Praxis besteht ein konzeptuelles Problem, wie Vergleichsfunktionen eingesetzt werden. Die Vergleichsfunktion (zum Beispiel equals() in Java) wird herkömmlich für die Anwendung geschrieben und beinhaltet daher Anwendungslogik. So sind zum Beispiel zwei Artikel eines Kaufhauses gleich, wenn die Eigenschaft “Artikelnummer” denselben Wert hat. Diese Vergleichsfunktion wird im Unit-Test wiederverwendet. Das Problem hierbei ist, dass nur eine Eigenschaft getestet wird. Besteht der Artikel aus weiteren Eigenschaften, bleiben diese Eigenschaften ungetestet. In einigen Fällen kann das erwünscht sein, in anderen nicht. Das Problem lässt sich lösen, indem eine weitere Vergleichsfunktion speziell für den Unit-Test geschrieben wird.

Ausgehend von der Richtigkeit dieser Annahme liefert der Prototyp eine spezielle Vergleichsfunktion für den Unit-Test, mit der sich eine bessere Trennung zwischen Anwendung und Unit-Test erreichen lässt.

4.8.5. Randbedingungen

Es bleibt zu klären, welche Bedingungen ein Unit-Test erfüllen muss, um mit dem Prototyp einen Mehrwert gegenüber herkömmlichen Unit-Tests zu erzielen.

Konstante Testwerte

Für dauerhafte Regressionstests gilt die Einschränkung, dass Testwerte konstant bleiben müssen. Sonst verschwimmt die Aussagekraft des Regressionstests, da nicht mehr eindeutig ist, ob der Test aufgrund von Änderungen in der Anwendung fehlschlägt oder aufgrund geänderter Testwerte. Das Problem tritt auf, wenn beispielsweise eine gemeinsame Testdatenbank für Unit-Tests verwendet wird und jeder Entwickler die Werte darin ändert. Der Prototyp kann die Symptome durch entsprechende Konfiguration bekämpfen, aber die Erkrankung selbst nicht heilen.

Keine Anwendungslogik

Für den Prototyp sind alle Unit-Tests ungeeignet, die Anwendungslogik voraussetzen. Hier kann er nur unterstützend eingesetzt werden, um einfache Vergleiche durchzuführen. Von der Anwendung abhängige Vergleiche muss der Entwickler von Hand schreiben.

Ein Beispiel wäre eine Anwendung, um den Preis eines Neuwagens zu ermitteln: Laut Geschäftsführung ist ein Auto mit Ledersitzen genauso viel wert wie ein Auto mit Klimaanlage, da der Kunde beim Kauf zwischen einen der beiden auswählen darf.

- Vergleiche Wert zwischen Auto mit Klimaanlage und Auto mit Ledersitzen

Ein generierter Test schlägt fehl, weil er die unterschiedlichen Werte von Klimaanlage und Ledersitz in den Vergleich einbezieht. Dies verletzt die Regel der Geschäftsführung, der Test ist damit falsch.

Vergleichen ist subjektiv

Damit Objekte vergleichbar werden, schreibt der Entwickler eine Vergleichsfunktion für jedes beteiligte Objekt. Diese Vergleichsfunktion prüft die ausgewählten Eigenschaften, die der Entwickler - oder die Anwendung - für die Gleichheit als relevant betrachtet.

Ein Beispiel ist eine Bücherei, die alle Bücher mit einem Computer verwaltet. Es gibt Bücher, die haben denselben Titel und stammen vom selben Autor, sind jedoch unter verschiedenen Verlagen veröffentlicht. Jeder Verlag hat ein anderes Druckformat, daher unterscheiden sich zwei gleiche Bücher aus unterschiedlichen Verlagen in ihrer Seitenzahl, obwohl beide Bücher Wort für Wort identisch sind. Wie ist in diesem Fall Gleichheit zu definieren? Dem Besucher der Bibliothek ist der Verlag womöglich nebensächlich, Hauptsache er findet das gewünschte Buch. Der Bibliothekar unterscheidet hingegen sehr wohl zwischen den Verlagen, da der eine Verlag günstiger ist als der andere. Auch im Unit-Test muss festgelegt werden, welche Eigenschaften für die Gleichheit von Bedeutung sind.

Der Entwickler kann mithilfe des Prototyps Ausnahmen festlegen. Je mehr Ausnahmen nötig werden, desto geringer ist sein Nutzen. Nehmen die Ausnahmen überhand, sollte der Entwickler die Vergleichfunktion auf herkömmliche Weise schreiben. Der Entwickler muss selbst entscheiden, wann der Aufwand mit dem Prototyp geringer ist.

4.8.6. Fazit

Der Prototyp erleichtert den Objektvergleich und den Regressionstest, indem der Aufwand entfällt, Vergleichsfunktionen zu schreiben.

Für Dauerhafte Regressionstests übernimmt der Prototyp die Verwaltung der gespeicherten Sollwerte. Der Nutzen ist eingeschränkt, wenn sich die Testwete häufig ändern. Ein flüchtiger Regressionstest erzeugt die Testwerte zur Laufzeit, daher gibt es keine Testwerte zu verwalten und der Prototyp kann nicht unterstützen.

Es gibt kein allgemeingültiges, automatisiertes Vergleichsverfahren. Der entwickelte Prototyp ist am Besten für Unit-Tests geeignet, die keine Anwendungslogik enthalten. Durch Konfiguration kann der Entwickler Ausnahmen festlegen, wodurch die Abhängigkeit von der Anwendungslogik aufgeweicht, aber nicht vollständig beseitigt wird. Der Entwickler muss abwägen, wann es aufwändiger ist, den Prototyp einzusetzen.

Rückblickend wurden die Anforderungen aus Kapitel 3 weitesgehend erfüllt:

1. Das Konzept Unit-Test wurde durch Strukturähnlichkeit mit JUnit beibehalten
2. Der Aufwand für einen Unit-Test wurde durch Automatisierung reduziert
3. Listen wurden durch geeignete Suchverfahren automatisch vergleichbar (Ausnahmen benötigen weitere Information vom Entwickler)
4. Der Regressionstest wurde durch Verwaltung der Testwerte vereinfacht
5. Die Konfiguration wurde ohne Einschränkungen in das Konzept eingegliedert
6. Anpassungen wurden erforderlich, um beim Vergleich von XML-Dokumenten praxistaugliche Ergebnisse zu erzielen

Die Einbindung des Prototyps in den Unit-Test hat darüber hinaus den positiven Nebeneffekt, dass er ohne Anpassungen in einem bestehenden Continuous Integration System funktioniert.

5. Architekturkonzept

5.1. Einleitung

Dieses Kapitel beschreibt die Architekturrichtlinien, die maßgebend für die Entwicklung des Prototyps sind. Die Architektur des Prototyps wird vorgestellt und anhand wichtiger Klassen erläutert. Die Architektur wurde entsprechend den Anforderungen aus Kapitel 3 angepasst. Alle Maßnahmen verfolgen das Ziel, Unit-Tests für den Entwickler effizienter zu gestalten.

5.2. Architekturrichtlinien

Das bestehende Konzept von JUnit soll fortgeführt werden, um die Erwartungshaltung des Entwicklers zu erfüllen. Der Prototyp orientiert sich daher an JUnit 4 und verzichtet auf Vererbung. Dadurch behält der Entwickler mehr Freiheit beim Design seiner Testklassen.

Die Klassen XmlDiff und XmlConfig bilden zusammen die Schnittstelle zum Entwickler. Alle anderen Klassen sind für den internen Gebrauch. Die Architektur des Prototyps ist so ausgerichtet, dass die Schnittstelle zum Entwickler möglichst schlank bleibt.

Für einen Testfall befinden sich alle notwendigen Funktionen in der Klasse XmlDiff. Die Funktionen sind wie bei der Klasse Assert von JUnit statisch. Die Klasse XmlDiff fasst viele interne Funktionen zu wenigen Funktionen zusammen, dadurch muss sich der Entwickler nur in diese Klasse einarbeiten. Es handelt sich um das Fassademuster [Gamma u. a. (2005)]. In Abbildung 5.1 kann man das Fassademuster daran erkennen, dass alle Abängigkeiten von der Klasse XmlDiff zu den internen Paketen führen.

Bei jeder internen Klasse wird darauf wert gelegt, dass sie genau einen Zweck erfüllt, streng nach dem Architekturprinzip “Entwurf nach Zuständigkeit” [Dijkstra (1982)]. Komplexe Funktionen werden durch die Kombination mehrerer Funktionen zusammengesetzt, ähnlich wie in Unix-Betriebssystemen: Dort existieren viele kleine Programme, die über sogenannte Pipes kombiniert werden, um so komplexe Programme zu erhalten. Damit sind weniger Abhängigkeiten zwischen den Klassen erforderlich. Auch die Wiederverwendbarkeit der Klassen erhöht sich.

5.3. Architektur auf Paketebene

Die Abbildung 5.1 zeigt die Pakete und Abhängigkeiten im Prototyp. Zusätzlich werden die Klassen XmlDiff und XmlConfig aufgeführt, da sie für den Entwickler die Schnittstelle zur Bibliothek bilden. Der Begriff Framework hat in der Abbildung die Bedeutung von Bibliothek.

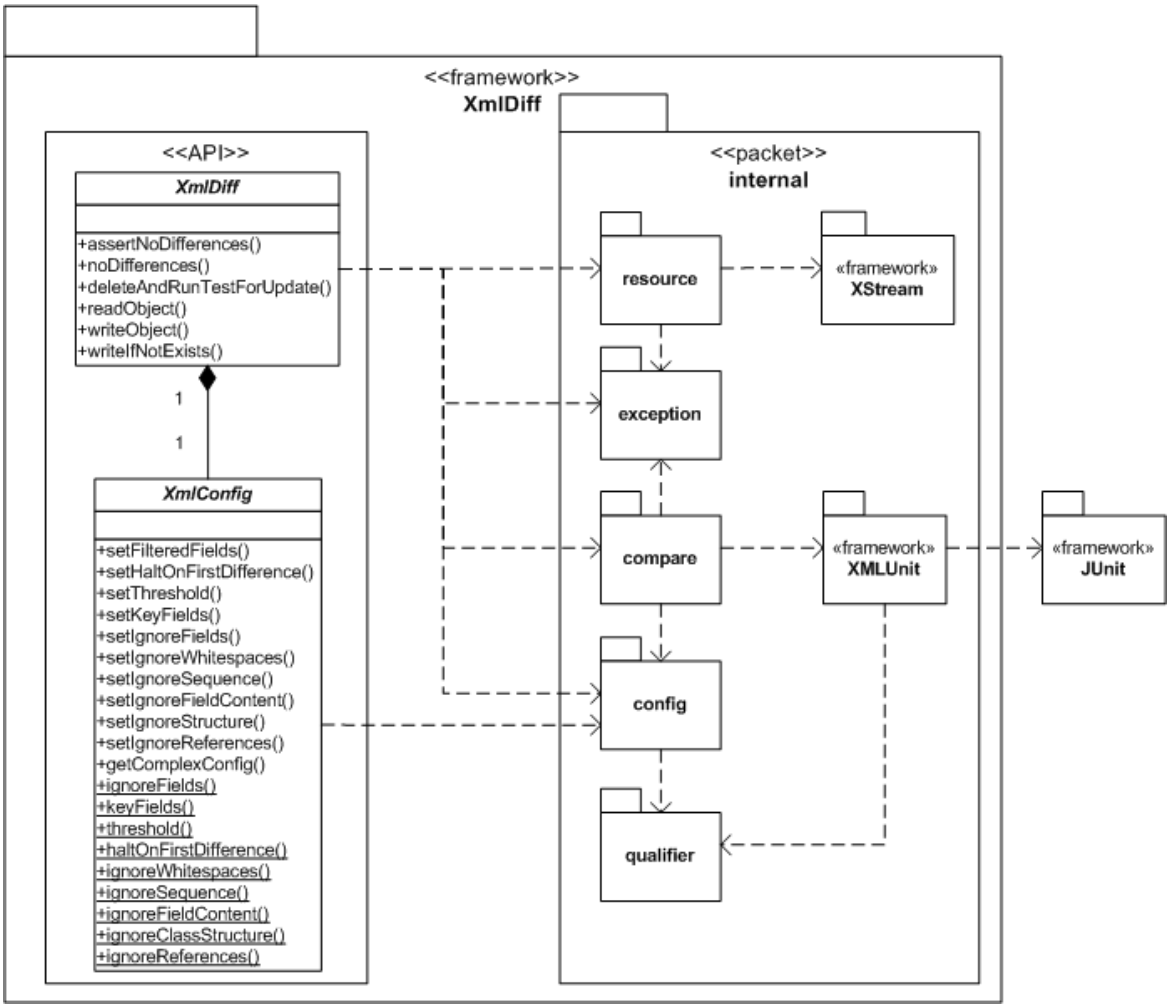

Abbildung 5.1.: Die Pakete von XmlDiff und ihre Abhängigkeiten

Das Paket “internal” enthält folgende Pakete:

- resource: Verwaltet Testwerte sowie die Ausgabe von XML- und HTML-Dateien.
- exception: Enthält alle selbst definierten Ausnahmen für die Fehlerbehandlung.
- compare: Startet abhängig von den Einstellungen den Vergleich und sorgt für Kombatibiltät zu JUnit.

- config: Enthält Zustand und Konstanten, um den Vergleich zu steuern.
- qualifier: Stellt die Suchverfahren für den Vergleich zur Verfügung.

Das Fundament des Prototyps bilden die Bibliotheken XMLUnit und XStream. XStream nimmt die Umwandlung zwischen Objekt und XML-Dokument vor. XMLUnit vergleicht die XML-Dokumente mit den zur Verfügung gestellten Suchverfahren. Die Bibliothek JUnit ist optional und daher nicht im Prototyp enthalten. JUnit garantiert die Gleichheit bei einem JUnit-Test.

5.4. API zu XmlDiff

Ein JUnit-Test erfordert den Aufruf einer Funktion, die mit den Präfix “assert” beginnt. Um in dieser Tradition fortzufahren, heißt die Funktion für den automatisierten Vergleich “assertNoDifferences”. Sie ähnelt der Funktion “assertEquals” von JUnit und erwartet als Parameter zwei Objekte, deren Gleichheit garantiert wird. Optional kann eine Nachricht übergeben werden, die dann ausgegeben wird, wenn der Unit-Test fehlschlägt. Solange keine Einstellungen notwendig sind, verhalten sich “assertEquals” und “assertNoDifferences” nahezu identisch. Der Unterschied zwischen den beiden Funktionen ist, dass der Entwickler bei “assertNoDifferences” keine “equals”-Funktionen bereitstellen muss. Alternativ kann der automatisierte Vergleich ohne JUnit mit der Funktion “noDifferences” durchgeführt werden.

```
testPersonen() {
    // Testdaten
    Person jekyll = new Person("Jekyll");
    Person hyde = new Person("Hyde");

    // JUnit-Test schlägt fehl
    XmlDiff.assertNoDifferences(jekyll,hyde);
}
```

Listing 5.1: Einfacher Unit-Test mit XmlDiff

5.5. Konfiguration

Will der Entwickler den automatischen Vergleich steuern, dann muss er ein Objekt von Typ IXmlConfig an “assertNoDiffferences” übergeben.

Für das Interface IXmlConfig gibt es zwei Implementationen: XmlComplexConfig und XmlConfig. XmlComplexConfig enthält alle Einstellungen - auch die für interne Zwecke. Bei der Klasse XmlConfig handelt es sich um eine vereinfachte Variante der Klasse XmlComplexConfig. Die Klasse XmlConfig kapselt hierzu ein XmlComplexConfig-Objekt und bietet den Entwickler ausschließlich für ihn relevante Funktionen an. XmlConfig ist für den Entwickler übersichtlicher und zugänglicher. Alle Funktionsaufrufe an XmlConfig werden an XmlComplexConfig weitergeleitet, sodass kein Quelltext dupliziert wird.

Intern wird über das Interface IXmlConfig immer die Klasse XmlComplexConfig verwendet, es ist also keine Unterscheidung der beiden Klassen erforderlich. Abbildung 5.2 stellt den Zusammenhang der Klassen dar.

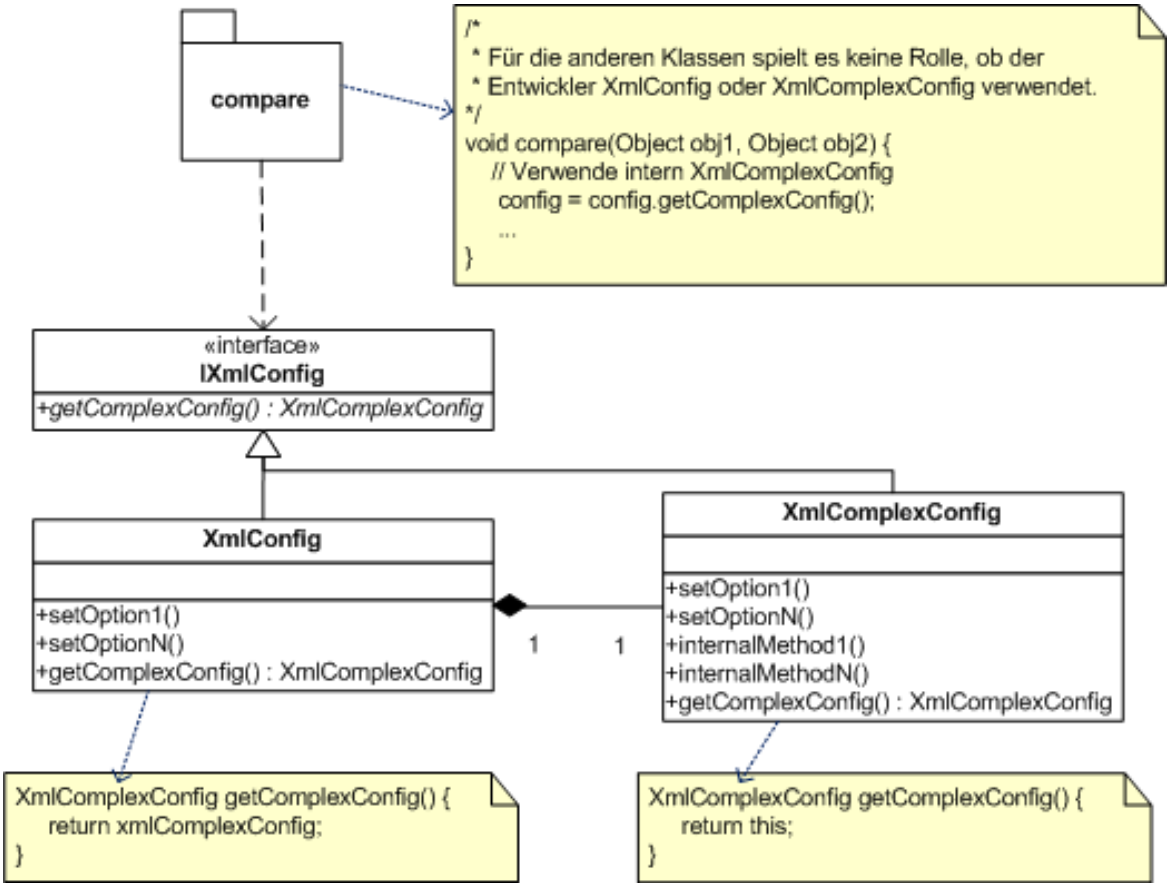

Abbildung 5.2.: Schnittstelle für den Entwickler vereinfachen durch Kapselung

```
testPersonen() {
    // Testdaten
    Person jekyll = new Person("Jekyll");
    Person hyde = new Person("Hyde");

    // JUnit-Test ist jetzt erfolgreich
    XmlDiff.assertNoDifferences(jekyll,hyde,XmlConfig.ignoreFieldContent
        ());
}
```

Listing 5.2: Einfacher Unit-Test mit konfiguriertem XmlDiff

5.6. Resourcenverwaltung

Die Funktion “assertNoDifferences” erhält zwei Objekte als Parameter. Für Regressionstests kann der Entwickler stattdessen ein Objekt von Typ XmlResource übergeben. Die Klasse XmlResource ist ein Fachwert, der einen String enthält. Dieser String dient als Identifikator zu den gespeicherten Testwerten. Beim Vergleich prüft XmlDiff, ob eine XmlResource vorliegt und lädt anhand des Identifikators das Objekt aus dem gespeicherten XML-Dokument. Der Identifikator ist abhängig vom Testfall: In zwei unterschiedlichen Testklassen führt derselbe Identifikator zu unterschiedlichen Testwerten. Listing 5.3 zeigt den Ablauf für einen Regressionstest mit dem Fachwert XmlResource.

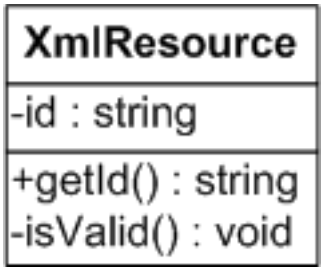

Abbildung 5.3.: Der Fachwert XmlResource

```
public void testPersonAusDatenbankLesen() {
    // 1. Soll-Wert von der Festplatte über XmlResource holen
    XmlResource jekyllSoll = new XmlResource("Jekyll");
    // 2. Ist-Wert aus der Datenbank holen
    Person jekyllIst = Datenbank.select("Jekyll");
    // 3. Regressionstest
    XmlDiff.assertNoDifferences(jekyllSoll, jekyllIst);
}

// Wird vom Entwickler manuell aufgerufen
public void main(String[] args) {
    // Testdaten erzeugen bzw. aktualisieren:
    // Führt den Test aus und schreibt das Objekt jekyllIst in ein XML-
       Dokument mit der ID "Jekyll".
    XmlDiff.deleteAndRunTestForUpdate();
}
```

Listing 5.3: Regressionstest mit XmlResource als Parameter

5.7. Fazit

Durch das Architekturprinzip “Entwurf nach Zuständigkeiten” entstehen spezialisierte Klassen, die einen bestimmten Zweck erfüllen. Das Fassademuster wird für die Klasse XmlDiff verwendet, um die Funktionen der internen Klassen zu bündeln. Die Schnittstelle zum Entwickler besteht so aus einer einzigen Klasse mit wenigen Funktionen. Um die Konfiguration zu erleichtern, existiert eine vereinfachte Klasse XmlConfig, die ein XmlComplexConfig-Objekt kapselt. Intern verwendet XmlDiff immer die Klasse XmlComplexConfig. Durch das Interface IXmlConfig ist keine Unterscheidung zwischen den beiden Klassen notwendig. Mittels Delegation wird kein Quelltext dupliziert. Die Verwaltung der Testdaten für einen Regressionstest übernimmt der Fachwert XmlResource. Eine XmlResource enthält für jeden Testfall einen eindeutigen Identifikator zu den gespeicherten Testwerten.

Alle Maßnahmen haben den Zweck, die Schnittstelle zwischen Entwickler und Bibliothek möglichst schlank und zugänglich zu halten.

6. Schluss

6.1. Zusammenfassung

Diese Arbeit hat gezeigt, dass sich Unit-Tests weiter automatisieren lassen. Mit dem entwickelten Prototyp entfällt es, Vergleichsfunktionen für den Unit-Test zu schreiben. Weiterhin können Regressionstests vereinfacht werden, indem der Prototyp die Testwerte verwaltet. Beide Maßnahmen minimieren den Aufwand für den Entwickler: Er muss weniger Code schreiben. Dadurch wird Zeit gespart, die der Entwickler für andere Aufgaben nutzen kann.

Für den automatischen Vergleich wird die Tatsache ausgenutzt, dass sich jeder Unit-Test auf einen Wertvergleich reduzieren lässt. Der Wertvergleich wird dann automatisiert. Allerdings gibt es Einschränkungen, sobald ein Testfall spezielles Wissen über die Anwendung enthält.

Da es letztlich um Werte geht, bietet sich XML an, sie zu repräsentieren. Hierzu muss das Objekt aus einer Programmiersprache in ein XML-Dokument umgewandelt werden. XML ist unabhängig von der Plattform und der Programmiersprache, wodurch ein Prototyp unabhängig von der konkreten Implementation des Unit-Tests umsetzbar ist. Weiterhin spielt die Lesbarkeit von XML-Dokumenten eine wichtige Rolle, besonders wenn Testwerte von Anwendern durchgesehen oder erstellt werden. Dadurch sind FIT-Tests (hier FIX genannt) auf Basis von XML statt von Tabellen möglich.

Das Problem ist, ein automatisches Vergleichsverfahren für XML-Dokumente zu entwickeln. Die Grenze ist erreicht, sobald der gefundene Unterschied von der Interpretation abhängt. Eine unterschiedliche Adresse bei einer Person lässt sich beispielsweise als Umzug interpretieren oder als weiterer Untermieter. Eine korrekte Entscheidung ist von einem Programm nicht länger zu erwarten. Ähnliche Schwierigkeiten bereitet es, Schlüsselwerte automatisch zu erkennen. Daher haben Suchverfahren zur Ähnlichkeitsbestimmung einen hohen Stellenwert für die Qualität des gesamten Vergleichsverfahrens. Ein optimales Suchverfahren erkennt zusammengehörige Objekte, auch bei massiven Unterschieden. Leider hat sich herausgestellt, dass ein optimales Suchverfahren nicht existiert. Als Konsequenz muss ein Entwickler dem Prototyp zusätzliches Wissen bereitstellen, sodass der Vergleich wieder korrekt durchführbar ist. Weitere Einstellungen helfen dem Entwickler, Ausnahmen zu definieren, um Einschränkungen zu umgehen.

6.2. Stand

Dass der Prototyp mit sich selbst getestet keine Fehler entdeckt hat, fasst der Autor als gutes Zeichen auf. Der Prototyp umfasst den automatischen Vergleich, die Verwaltung von Testdaten in Regressionstests sowie zahlreiche Konfigurationsmöglichkeiten. Die Kompatibilität zu JUnit Version 3 und 4 ist gewährleistet. Darüber hinaus kann der Prototyp HTML-Dateien erzeugen, um Unterschiede grafisch gegenüberzustellen.

6.3. Offene Punkte

Die verwendete Bibliothek XMLUnit verwendet DOM-Bäume [W3C (a)], um XML-Dokumente einzulesen. Für jedes XML-Dokument befindet sich ein DOM-Baum vollständig im Hauptspeicher. Mit üblichen Einstellungen kann die virtuelle Maschine von Java maximal 256 MB Speicher reservieren. Beim Vergleich großer XML-Dokumente stößt der Prototyp daher an die Speichergrenze. Dies führt zu einen “Heap Overflow”. Der Ausweg ist, SAX statt DOM zu verwenden. SAX ist ereignisgesteuert und liest XML-Dokumente ein, ohne einen Baum im Speicher zu erzeugen [Megginson (2000)]. Eine Umstellung bedeutet jedoch, dass XMLUnit komplett umgeschrieben werden muss - eine heikle und aufwendige Angelegenheit, die den Rahmen dieser Arbeit sprengen würde.

Die grafische Gegenüberstellung von Unterschieden als HTML-Dokument ist rudimentär umgesetzt. Für jeden möglichen Unterschied ist eine besondere Verarbeitung notwendig. Weiterhin ist die Frage zu klären, wie Unterschiede am besten grafisch dargestellt werden können.

FIX fehlt es an Werkzeugen, die ähnlich wie bei FIT es erleichtern, XML-Dokumente und Ergebnisse ohne Programmierkenntnisse zu erzeugen.

6.4. Ausblick

Die gesammelten Erfahrungen lassen sich nutzen, um eine Musterarchitektur für das Vergleichsverfahren zu entwickeln. Die Musterarchitektur ist in sechs Schichten unterteilt. Höhere Schichten dürfen auf untere Schichten zugreifen, aber nicht umgekehrt. Jede Schicht ist unabhängig von den anderen Schichten austauschbar. Abbildung 6.1 stellt die Musterarchitektur vor.

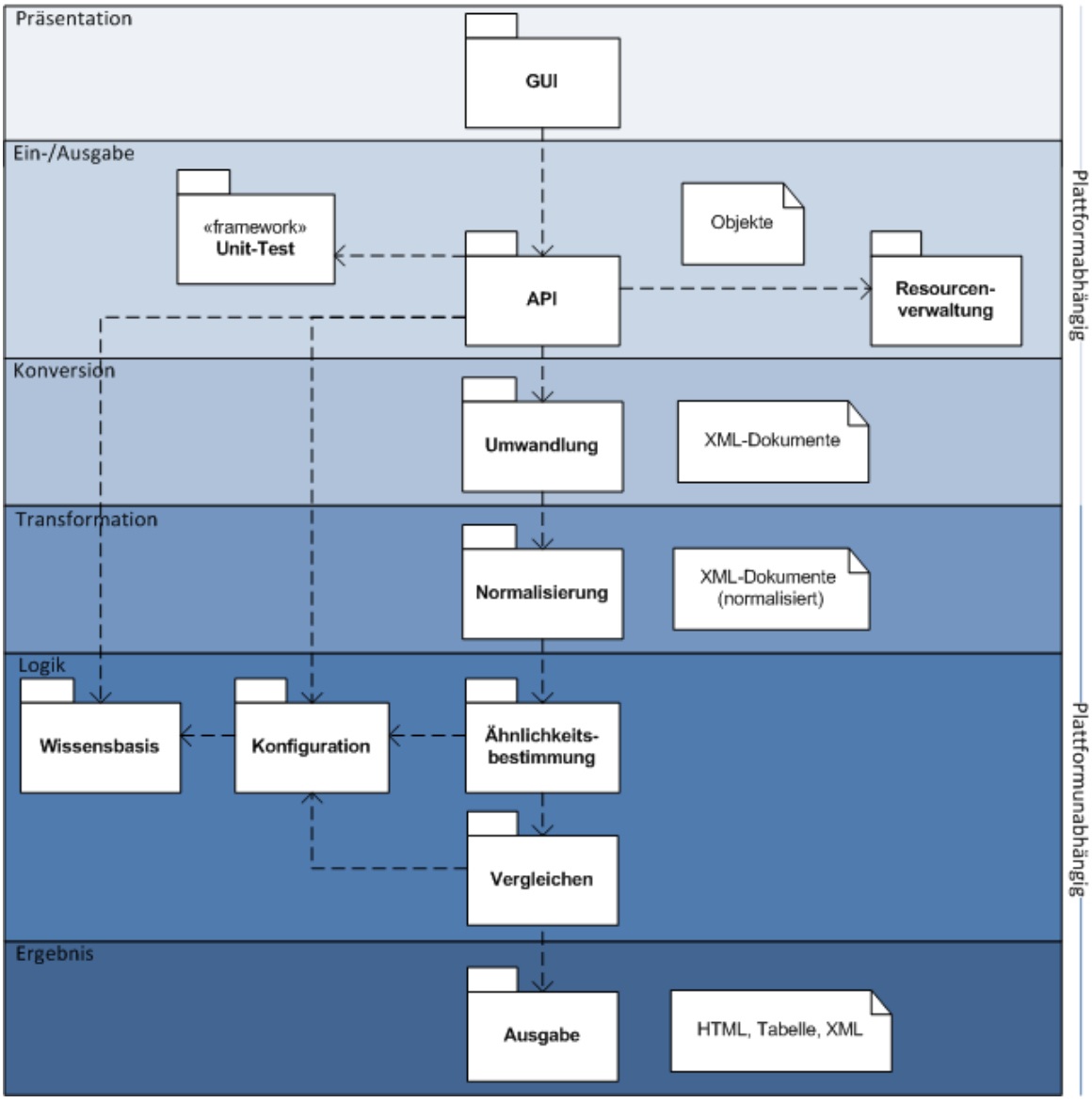

Abbildung 6.1.: Überarbeitete allgemeine Musterarchitektur

Die drei oberen Schichten sind plattformabhängig und je nach Programmiersprache oder Betriebssystem anzupassen. In der Präsentationsschicht befindet sich die grafische Oberfläche für Einstellungen und die Anzeige der Ergebnisse. Anwendungen greifen über die API in der Ein-/Ausgabeschicht auf die Bibliothek zu. Die API benachrichtigt zudem die Unit-Test-Bibliothek über das Testergebnis. Die Eingliederung in ein Continuous Integration System erfolgt über die Unit-Test-Bibliothek. Weiterhin findet in der Ein-/Ausgabeschicht die Verwaltung gespeicherter Objekte für den Regressionstest statt. Die Konversionsschicht wandelt die Objekte in XML-Dokumente um. Die folgenden drei Schichten sind somit plattformunabhängig und wiederverwendbar. Die Tranformationsschicht normalisiert die XML-Dokumente, um den Vergleich zu vereinfachen. Zum Beispiel können die XML-Dokumente sortiert oder bestimmte Bezeichner gefiltert werden. Die Logikschicht vergleicht die transformierten XML-Dokumente auf Grundlage der Konfiguration. Zunächst werden ähnliche Objekte identifiziert und anschließend ihre Unterschiede ermittelt. Zusätzlich merkt sich eine Wissensbasis die Einstellungen des Anwenders. Die Wissensbasis hilft, zukünftige Vergleiche automatisch zu konfigurieren. Sie lernt vom Anwender, welche Einstellungen bei welchen Testwerten gemacht werden. Damit sind automatische Vergleiche mit Anwendungslogik möglich, womit das bisherige Hauptproblem kompensiert wird. In der Ergebnisschicht werden die Unterschiede schließlich in der gewünschten Repräsentationen zurückgegeben.

Abschließend ist eine weite Verbreitung der hier vorgestellten Ideen zur Automatisierung des Unit-Tests wünschenswert.

Literaturverzeichnis

[Bacon und Martin 2007] BACON, Tim ; MARTIN, Jeff: *XMLUnit*. 2007. – URL http://xmlunit.sourceforge.net

[Cunningham 2007] CUNNINGHAM, Ward: *Fit: Framework for Integrated Test*. Januar 2007. – URL http://fit.c2.com

[Dijkstra 1982] DIJKSTRA, Edsger W.: *On the role of scientific thought*. Springer-Verlag New York, Inc., 1982. – 60–66 S. – ISBN 0387906525

[Falk 2005] FALK, Steffi: *Algorithmus zur Ähnlichkeitsbestimmung von XML-Dokumenten und -Schemata*, Universität Rostock, Diplomarbeit, 2005. – URL http://dbis.informatik.uni-rostock.de/Studium/Diplomarbeiten/1117629545.18_0/Elektronische_Version/DiplomarbeitSteffiFalk.pdf

[Freese 2007] FREESE, Tammo: *EasyMock*. Dezember 2007. – URL http://www.easymock.org/

[Gamma und Beck 2007] GAMMA, Erich ; BECK, Kent: *JUnit*. 2007. – URL http://www.junit.org

[Gamma u. a. 2005] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-oriented Software*. 3. Addison Wesley, Oktober 2005. – 426 S. – ISBN 1405837306

[Heuer und Saake 2000] HEUER, Andreas ; SAAKE, Gunter: *Datenbanken Konzepte und Sprachen*. 2. mitp, 2000. – 704 S. – ISBN 3826606191

[Lahres und Rayman 2006] LAHRES, Bernhard ; RAYMAN, Gregor: *Praxisbuch Objektorientierung. Von den Grundlagen zur Umsetzung*. 1. Galileo Press, August 2006. – 609 S. – ISBN 3898426246

[Massol und Husted 2004] MASSOL, Vincent ; HUSTED, Ted: *JUnit in Action*. Manning Publications Co., 2004. – 359 S. – ISBN 1930110995

[Megginson 2000] MEGGINSON, David: *SAX*. Mai 2000. – URL http://www.saxproject.org/

[Schmitt 2005] SCHMITT, Ingo: *Ähnlichkeitssuche in Multimedia-Datenbanken. Retrieval, Suchalgorithmen und Anfragebehandlung*. Oldenbourg, November 2005. – 445 S. – ISBN 348657907X

[Spillner und Linz 2005] SPILLNER, Andreas ; LINZ, Tilo: *Basiswissen Softwaretest*. 3. dpunkt.verlag, 2005. – 276 S. – ISBN 3898643581

[W3C a] W3C, World Wide Web C.: *Document Object Model (DOM)*. – URL http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/

[W3C b] W3C, World Wide Web C.: *Extensible Markup Language (XML)*. – URL http://www.w3.org/XML

[W3C c] W3C, World Wide Web C.: *Simple Object Access Protocol (SOAP)*. – URL http://www.w3.org/TR/soap/

[W3C d] W3C, World Wide Web C.: *XML Path Language (XPath)*. – URL http://www.w3.org/TR/xpath

[W3C e] W3C, World Wide Web C.: *XSL Transformations (XSLT)*. – URL http://www.w3.org/TR/xslt

[Walnes 2007] WALNES, Joe: *XStream*. 2007. – URL http://xstream.codehaus.org

A. Quelltexte der vorgestellten Verfahren

Anmerkung: Die Quelltexte sind lediglich Ausschnitte. Der vollständige Quelltext des Prototyps befindet sich auf der CD-ROM, die dieser Bachelorarbeit beiliegt.

A.1. Stufenvergleich

```
public static boolean noDifferencesStages(final Object controlObject,
    final Object testObject,
        final Appendable diffDescription) throws AppendException {
    boolean stillSimilar;

    IXmlCompare compare;
    StringBuilder message = new StringBuilder();

    XmlComplexConfig config;

    message = new StringBuilder("Structure of fields has been changed.\n"
        );
    config = new XmlComplexConfig();
    config.removeIgnore(XmlDifference.FIELD_STRUCTURE.getDifferences());
    config.addIgnore(XmlDifference.FIELD_CONTENT.getDifferences());
    config.addIgnore(XmlDifference.SEQUENCE.getDifferences());
    config.addIgnore(XmlDifference.REFERENCE_CHANGE.getDifferences());
    compare = new XmlCompare(config);
    stillSimilar = compare.noDifferences(controlObject, testObject,
        message);

    if (stillSimilar) {
        message = new StringBuilder("Content of fields has been changed.\
            n");
        config.removeIgnore(XmlDifference.FIELD_CONTENT.getDifferences())
            ;
        compare = new XmlCompare(config);
```

```
            stillSimilar = compare.noDifferences(controlObject, testObject,
                message);

            if (stillSimilar) {
                message = new StringBuilder("Sequence of fields/objects has
                    been changed.\n");
                config.removeIgnore(XmlDifference.SEQUENCE.getDifferences());
                compare = new XmlCompare(config);
                stillSimilar = compare.noDifferences(controlObject,
                    testObject, message);

                if (stillSimilar) {
                    message = new StringBuilder("References have been changed
                        .\n");
                    config.removeIgnore(XmlDifference.REFERENCE_CHANGE.
                        getDifferences());
                    compare = new XmlCompare(config);
                    stillSimilar = compare.noDifferences(controlObject,
                        testObject, message);

                    if (stillSimilar) {
                        message.delete(0, "References have been changed.\n".
                            length());
                    }
                }
            }
        }

        try {
            diffDescription.append(message);
        } catch (final IOException e) {
            throw new AppendException(e.getMessage());
        }

        return stillSimilar;
    }
```

Listing A.1: Stufenvergleich

A.2. Gemischtes-Suchverfahren

```
public class MultiLevelNameAndTextQualifier implements ElementQualifier {
    public boolean qualifyForComparison(final Element control, final
        Element test) {
        return control.isEqualNode(test);
    }
}
```

Listing A.2: Gemischtes-Suchverfahren

A.3. Schwellwert-Suchverfahren

```
public class MultiLevelThresholdQualifier implements ElementQualifier {
    private static final ElementQualifier NAME_QUALIFIER = new
        ElementNameQualifier();
    private final float threshold;

    public MultiLevelThresholdQualifier(final float threshold) {
        this.threshold = threshold;
    }

    public boolean qualifyForComparison(final Element control, final
        Element test) {
        boolean result = NAME_QUALIFIER.qualifyForComparison(control,
            test);

        if (result && control.hasChildNodes() && test.hasChildNodes()) {
            result = false;
            int matches = 0;

            for (int controlIndex = 0; controlIndex < control.
                getChildNodes().getLength(); controlIndex++) {
                final Node controlNode = control.getChildNodes().item(
                    controlIndex);
                boolean foundCorrespondent = false;

                for (int testIndex = 0; !foundCorrespondent && testIndex
                    < test.getChildNodes().getLength(); testIndex++) {
                    final Node testNode = test.getChildNodes().item(
                        testIndex);
                    boolean similar = false;
```

```
                    // Check only nodes of the same type.
                    if (controlNode.getNodeType() == testNode.getNodeType
                        ()) {
                        if (controlNode.getNodeType() == Node.
                            ELEMENT_NODE) { // node
                            similar = qualifyForComparison((Element)
                                controlNode, (Element) testNode); //
                                recursion
                        } else if (controlNode.getNodeType() == Node.
                            TEXT_NODE) { // leaf
                            similar = controlNode.isEqualNode(testNode);
                        } else { // Fehler
                            Assert.isTrue(false, "Internal error:
                                Unexspected node type.");
                        }

                        if (similar) {
                            foundCorrespondent = true;
                            matches++;
                        }
                    }
                }
            }

            // Calculate how many children of both nodes are equal (
                percental).
            if (matches > 0) {
                final float similarValue = matches / (float) control.
                    getChildNodes().getLength();
                if (similarValue > threshold) {
                    result = true;
                }
            }
        }

        return result;
    }
}
```

Listing A.3: Schwellwert-Suchverfahren

A.4. Schlüsselwert-Suchverfahren

```
public class MultiLevelKeyElementQualifier implements ElementQualifier {
    private static final ElementQualifier NAME_QUALIFIER = new
        ElementNameAndTextQualifier();
    private final String[] xpaths;

    public MultiLevelKeyElementQualifier(final String... tagnames) {
        Assert.notNull(tagnames, "Vorbedingung verletzt: tagnames != null
            .");
        xpaths = tagnames;
    }

    public boolean qualifyForComparison(final Element control, final
        Element test) {
        boolean result = NAME_QUALIFIER.qualifyForComparison(control,
            test);

        if (result) {
            for (final String xpath : xpaths) {
                final NodeList controlNodes = XPathUtil.getNodesByXpath(
                    control, xpath);
                final NodeList testNodes = XPathUtil.getNodesByXpath(test
                    , xpath);
                if (controlNodes.getLength() == testNodes.getLength() &&
                    controlNodes.getLength() > 0) {
                    for (int index = 0; result && index < controlNodes.
                        getLength(); index++) {
                        final Node controlNode = controlNodes.item(index)
                            ;
                        final Node testNode = testNodes.item(index);
                        if (!controlNode.isEqualNode(testNode)) {
                            result = false;
                        }
                    }
                }
            }
        }
        return result;
    }
}
```

Listing A.4: Schlüsselwert-Suchverfahren

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 6. Februar 2008	
Ort, Datum	Unterschrift