checkerberry test center

Benutzerhandbuch

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Beispiele

Kapitel 1. Checkerberry test center

1.1. Einleitung

Automatisierte Tests bilden das Herzstück eines guten Software-Entwicklungsprozesses. Sie ermöglichen den Einsatz von Continuous Integration Produkten, die dem Entwicklungsteam permanent Feedback über die Qualität des Systems liefern. Dies erhöht die Sicherheit bei der Software-Entwicklung, da Auswirkungen von Änderungen oder komplexen Refactorings schnell sichtbar werden. Dies steigert die Qualität der Software und somit auch die Kundenzufriedenheit. Continuous Integration Umgebungen sind mittlerweile den Kinderschuhen entwachsen und lassen sich auch ohne viel Erfahrung schnell einrichten. Eine Test-Umgebung ohne Tests liefert keine Fehler - allerdings auch keine Informationen. Der Grund für fehlende Tests ist vielfältig. Oft werden Zeitgründe vorgeschoben oder Komponenten als untestbar eingestuft.

Die Art und Weise wie Software getestet wird, befindet sich im Wandel. Während früher eigene Testteams und Testphasen vorgesehen waren, integriert sich der Prozess immer weiter in die Software-Entwicklung. Diese Entwicklung hat zwei Ursachen. Zum einen propagieren agile Vorgehensmodelle wie Scrum oder eXtreme Programming (XP), dass Software-Komponenten erst fertig entwickelt sind, wenn sie auch umfangreich getestet wurden. Durch die starke Verbreitung agiler Prozesse wandert somit immer mehr Testverantwortung in die Richtung des Software-Entwicklers. Zum anderen wächst das Verständnis im Management, dass die Behebung von Fehlern teurer wird, je später sie entdeckt werden. Oder andersherum: Je früher Fehler entdeckt werden, umso günstiger ist deren Behebung. Diese beiden Faktoren führen dazu, dass sowohl das Management als auch das Projektteam das Testen während der Software-Entwicklung fordern und fördern.

Bei der Betrachtung von Tests muss man zwischen manuellen und automatisierten Tests unterscheiden. Manuelle Tests werden durch eine Person durchgeführt, die direkte Aktionen am Rechner durchführt. Im Gegensatz dazu werden automatisierte Tests von einem eigenen Software-Programm durchgeführt. Automatisierte Tests haben den entscheidenden Vorteil, dass sie nur einmal erstellt werden müssen und dann beliebig oft ausführbar sind. Der Anspruch eines Software-Entwicklers besteht darin, eine hohe Testabdeckung durch automatisierte Tests zu erzielen. Durch die regelmäßige Ausführung der Tests z.B. in Continuous Integration Systemen bekommt der Entwickler ein schnelles Feedback zur Qualität der Software.

Das Herstellen der Testvoraussetzungen muss schnell möglich sein, damit der Fokus auf den eigentlichen Testfall gelegt werden kann. Mit dem checkerberry test center ist diese Voraussetzung geschaffen, wodurch die Produktivität des Software-Entwicklers bei der Testerstellung massiv gesteigert werden kann.

Das checkerberry test center ist eine Sammlung von Bibliotheken und Werkzeugen für die Optimierung der Entwicklung von automatisierten, funktionalen Integrationstests. Das checkerberry test center fokussiert sich auf spezielle Bereiche, die sich zwar gut testen lassen, die jedoch einen hohen Aufwand bei der Testerstellung erfordern. Das checkerberry test center kapselt diese komplexen und aufwändigen Aspekte der Testerstellung und stellt dem Software-Entwickler eine einfache Schnittstelle für die Testerstellung zur Verfügung. Dieses Benutzerhandbuch richtet sich an Software-Entwickler und liefert einen vollständigen Überblick über die Bibliotheken checkerberry db, und checkerberry web sowie über checkerberry business view und checkerberry cockpit. Die Beschreibungen umfassen die Funktionsweise, die Architektur und die Installation der einzelnen Komponenten. Zusätzlich geben die Best Practices und die Beantwortung häufig gestellter Fragen einen Einblick in die Praxis.

1.2. Release-Notes

1.2.1. Änderungen in Version 3.1.5

Tabelle 1.1. Änderungen in Version 3.1.5

Ticket-Nr	Beschreibung
255	Anpassung auf Selenium 2.40 Checkerberry wurde auf Selenium 2.49 aktualisiert. Ab dieser checkerberry Version ist Selenium 2.40 oder höher erforderlich.

1.2.2. Änderungen in Version 3.1.4

Tabelle 1.2. Änderungen in Version 3.1.4

Ticket-Nr	Beschreibung
252	Aktivierung auf VM schlägt fehl Die Aktivierung der checkerberry-Lizenz wurde so angepasst, dass sie nun auch in den beobachteten Konstellationen erfolgreich ist. Bereits bestehende Installationen sind von der Umstellung nicht betroffen.
253	java.awt.HeadlessException bei Lizenzprüfung Vor dem Öffnen eines Dialogs wird geprüft, ob eine grafische Oberfläche zur Verfügung steht. Ist dies nicht der Fall, wird eine Exception geworfen.

1.2.3. Änderungen in Version 3.1.3

Tabelle 1.3. Änderungen in Version 3.1.3

Ticket-Nr	Beschreibung
223	Darstellung der Ergebnisse in einer unabhängigen HTML-Seite (ohne Hudson) Es wurde ein neues Modul eingefügt (checkerberry-atdd-html-maven-plugin), das den ATDD-Report unabhängig von einem CI-System als HTML-Report erstellt.

1.2.4. Änderungen in Version 3.1.2

Tabelle 1.4. Änderungen in Version 3.1.2

Ticket-Nr	Beschreibung
249	RemoteControlComponent.getSelected*-Methoden fehlen Die fehlenden Methoden (getSelectedId(), getSelectedIds(), getSelectedIndex(), getSelectedIndexes(), getSelectedValue(), getSelectedValues(), getSelectedLabel(), getSelectedLabels()) wurden in der Klasse RemoteControlComponent implementiert.
250	Spring4-Integration Für Spring4 wurden neue Hilfsprojekte angelegt (checkerberry-db-spring4-bridge, checkerberry-spring4-integration), die für eine einfache Anbindung an Spring4 verwendet werden können.

1.2.5. Änderungen in Version 3.1.1

Tabelle 1.5. Änderungen in Version 3.1.1

Ticket-Nr	Beschreibung
247	Hudson-Plugin hat Java-Inkompabilität Das Hudson-Plugin wird wieder mit der gleichen Java-Version wie die Basis-Bibliotheken von Hudson gebaut und ist dadurch wieder aktivierbar.

1.2.6. Änderungen in Version 3.1.0

Tabelle 1.6. Änderungen in Version 3.1.0

Ticket-Nr	Beschreibung
185	Logging Selenium-Server Konfiguration des Selenium Logging wurde in den FAQs dokumentiert.
209	Vereinheitlichtes Logging mit SLF4J Das Logging intern wurde auf slf4j-api umgestellt und commons-logging wurde durch jcl-over-slf4j ersetzt.
222	Anzeigen der Akzeptanz-Test-Identifikationsnummern im Hudson Die ID des Akzeptanztests wird jetzt in dem Hudson-Bericht mit angezeigt.
233	Javadoc in Beispielprojekten laden Im eclipse-Plugin wurde das Laden der Sourcen und des Javadocs aktiviert.
234	Selenium Version 2.28.0 einbinden Es wurde die aktuelle Selenium Version 2.39.0 eingebunden.
237	LocalDNS wirft Exception unter Java 7 Das Setzen eines lokalen DNS-Servers hat unter Java 7 und höher eine Exception ausgelöst. Für Java 7 und höher ist das Setzen eines lokalen DNS-Server nicht mehr erforderlich, da die Klasse InetAddress dort effizienter implementiert wurde. Dieser Bugfix deaktiviert das Setzen des lokalen DNS-Servers in Java 7 und verhindert so das Werfen der NoSuchField-Exception.
238	Xerces-Version erhöhen Die interne Xerces-Version wurde auf 2.4.0 erhöht. Mit der vorherigen Version konnten auf einigen Linux-Systemen die Lizenzen nicht aktiviert werden.
239	Dokumentation der Aktivierung über Kommandozeile Die Lizenzaktivierung über die Kommandozeile wurde dokumentiert und der Aufruf vereinfacht.
241	User-Stories aus Jira lesen Über das Projekt checkerberry-atdd-jira-userstory-retriever kann checkerberry business view die Informationen zu User-Stories direkt aus Jira einlesen.
242	Konkretes Logging entfernen Das Logging intern wurde auf slf4j-api umgestellt und commons-logging wurde durch jcl-over-slf4j ersetzt.
243	Reports verwenden Validatoren nicht

Ticket-Nr	Beschreibung
	Bei der Überprüfung der Werte in den Reports werden jetzt die konfigurierten Validatoren verwendet.

1.2.7. Änderungen in Version 3.0.6

Tabelle 1.7. Änderungen in Version 3.0.6

Ticket-Nr	Beschreibung
230	Firefox 16 wird in den Beispielprojekten nicht unterstützt Es wurde auf die aktuelle Selenium-Version (2.25.0) aktualisiert.
231	Zentrales Maven-Repository fehlt in Beispielprojekten Das zentrale Maven-Repository wurde zu allen Beispielprojekten hinzugefügt.

1.2.8. Änderungen in Version 3.0.0

Tabelle 1.8. Änderungen in Version 3.0.0

Ticket-Nr	Beschreibung
18	Die Buttons in checkerberry cockpit sollten Image-Buttons sein Umstellung der Buttons in checkerberry cockpit auf Image-Buttons.
98	Webdriver unterstützen Neben Selenium RC wird jetzt auch WebDriver als Implementierung der Fernsteuerung unterstützt. Die API der Modelle ändert sich dadurch nicht. Allerdings unterscheidet sich das Verhalten von SeleniumRC und WebDriver an der ein oder anderen Stelle.
146	Unterstützung von Firefox 10, Google Chrome, etc Checkerberry wurde auf Selenium 2.20.0 umgestellt, sodass die aktuellsten Browser jetzt bestmöglich unterstützt werden.
165	BHB sollte Feature-Orientiert sein Das Benutzerhandbuch wurde so umgestellt, dass die Features im Vordergrund stehen.
186	DateValidator unterstützt kein Time-Format Der Date-Validator unterstützt jetzt auch den Fall, dass in der Datenbank das Datum als Timestamp z.B. 2012-12-31 12:00:00.0 angegeben wird.
190	Dokumentation NumberValidator Das Benutzerhandbuch wurde um ein Beispiel erweitert, um diesen Fall klarzustellen.
192	Doku TimerService In der Funktion now wurde dokumentiert, wie ein eigener TimeService integriert werden kann, der nicht die Systemzeit zurückliefern muss.
201	Neueste Selenium-Version integrieren Selenium Version 2.13.0 integiert
206	Auflösung der Testdaten-Namen fehlerhaft bei Vererbung Wenn eine abstrakte Testklasse eine Testmehode enthält, wird bei der Ermittlung des zugehörigen Testklassennamens jetzt immer die konkrete Testklasse verwendet. Dies

Ticket-Nr	Beschreibung
	ermöglicht die Definition einer allgemeinen "abstrakten" Testmethode, die über verschieden konkrete Implementierungen mit unterschiedlichen Testdaten verwendet wird.
207	Hex-Buchstaben werden in BCD nicht unterstützt Die Angabe von BCD-Werten mit Hex-Zeichen ist mit diesem Release möglich z.B. X'2b5e02fb910c'. Dabei wird das Zeichen "a" bzw "A" mit 1010 kodiert, "b" / "B" mit "1011" etc.
208	Unzureichende Fehlermeldung bei nicht auflösbaren Auto-Parametern Für den Fall, dass die erwarteten Testdaten mindestens einen Auto-Parameter beinhalten, der nicht aufgelöst werden kann, wird jetzt eine eigene Fehlermeldung geworfen (DB-TBX-1058).
210	Metadaten als techn. Konzept aus API entfernen Das technische Konzept der Metadaten wurde gegen fachliche Kontexte (Parameter, Auto-Parameter, Validatoren) ausgetauscht.
212	Lookup-Keys fehlerhaft bei null Werten Die Verwendung von null-Werten in Lookup-Keys funktioniert wieder.
214	Angabe mehrerer Tabellen in @NoCache verbessern In der @NoCache-Annotation werden mehrere Tabellen jetzt als Liste von einzelnen String angegeben.
215	Langsame Startphase bei GUI-Tests Beim Starten der GUI-Tests verzögert die (überflüssige) Auflösung von Hostnamen in java.net.InetAddress den Start erheblich. Daher wurden alternative DNS Name Service Provider zur Verfügung gestellt, die den Start um Faktor 5-8 beschleunigen.
218	Keine extra Browserfenster bei GUI-Tests Beim Starten eines GUI-Tests wird jetzt nicht zusätzlich ein weiteres Browserfenster geöffnet. Dadurch verkürzen sich die Testlaufzeiten und die Komplexität in der Setup-Phase wird reduziert.
226	Minimal-invasive DTD-Änderungen Der Algorithmus zur Berechnung der DTD wurde neu implementiert. Der neue Algorithmus berücksichtigt die bisherige Reihenfolge der Datenbanktabellen und versucht diese Reihenfolge möglichst beizubehalten. Zudem ist die Berechnung einer neuen DTD dadurch schneller geworden.

1.3. Allgemeine Test-Konzepte

Dieser Abschnitt liefert einen kurzen Überblick über allgemeine Test-Begriffe und –Konzepte, um ein gemeinsames Grundverständnis für Tests zu vermitteln. Des Weiteren erfolgt die Einordnung des checkerberry test centers in die dargestellten Test-Landschaft.

1.3.1. Test-Arten

Bei der Erstellung von Tests werden verschiedene Test-Arten unterschieden. Die Arten beschreiben, was getestet werden soll. Im Folgenden werden die für das checkerberry test center relevanten Test-Arten beschrieben.

1.3.1.1. Unit-Tests

Komplexe Systeme wie z.B. ein CD-Player oder eine Web-Anwendung bestehen häufig aus vielen einzelnen Komponenten, die zu einem Gesamtsystem integriert werden. Das Ziel eines Unit-Tests besteht darin, die

korrekte Funktionsweise einer einzelnen Komponente sicherzustellen. Für das Beispiel des CD-Players könnte diese Komponente das Display sein. Das Testen von integrierten Komponenten ist hingegen nicht die Aufgabe eines Unit-Tests.

Die Granularität der zu testenden Komponente ist variabel. In der Software-Entwicklung kann sich der Unit-Test auf eine einzelne Java-Klasse beziehen. Es ist jedoch auch möglich, dass sich der Unit-Test auf eine Menge von Java-Klassen bezieht, die zusammen ein Modul bilden, das eine bestimmte Funktionalität zur Verfügung stellt.

1.3.1.2. Integrationstests

Die Aufgabe eines Integrationstests besteht im Gegensatz zu einem Unit-Test darin, das korrekte Zusammenspiel von verschiedenen Modulen in einer integrierten Umgebung sicherzustellen.

Als Beispiel kann eine einfache Software zum Speichern von Kreditkarten-Informationen dienen. Das Gesamtsystem besteht aus zwei Komponenten: Die erste Komponente validiert die Angaben der Kreditkarte, während die zweite Komponente die Information in der Datenbank speichert. Die korrekte Funktionsweise der ersten Komponente kann über einen Unit-Test gewährleistet werden.

Die Erstellung eines Unit-Tests für die zweite Komponente ist nicht möglich, da zu diesem Zweck eine externe Komponenten, die Datenbank, erforderlich ist. In diesem Fall muss ein Integrationstest erstellt werden, der die zweite Komponente mit integrierter Datenbank testet.

1.3.1.3. GUI-Tests

GUI-Tests werden verwendet, um grafische Oberflächen von Anwendungen zu testen. Da diese Tests eine laufende Anwendung benötigen, handelt es sich um eine spezielle Art von Integrationstests.

1.3.1.4. Abgrenzung Unit- und Integrationstests

Zwischen Unit- und Integrationstests gibt es nur theoretisch eine scharfe Abgrenzung. In der Praxis ist eine genaue Klassifizierung nicht immer möglich. Dies liegt daran, dass der Begriff „Integration" nicht eindeutig festgelegt ist.

In der Praxis spricht man von Unit-Tests, wenn eine Klasse oder eine Menge von Klassen in der selben Java Virtual Machine (JVM) getestet wird. Sobald weitere externe Systeme erforderlich sind wie z.B. Datenbanken, spricht man hingegen von Integrationstests.

1.3.2. Test-Durchführung

Während bei der Testerstellung die Frage „Was wird getestet?" im Vordergrund steht, stellt sich bei der Test-Durchführung die Frage „Wie wird getestet?". Bei der Test-Durchführung gibt es verschiedene Techniken, die im Folgenden beschrieben werden.

1.3.2.1. Manuelle Tests

Manuelle Tests zeichnen sich dadurch aus, dass sie von einer Person durchgeführt werden. Der Tester führt den Test manuell aus und prüft die erwarteten Ergebnisse.

Ein Vorteil von manuellen Tests besteht darin, dass der Tester ein intuitives Verständnis für die erwarteten Ergebnisse hat. Wird z.B. eine grafische Oberfläche getestet, sieht der Tester sofort, wenn das Layout der zu testenden Oberfläche fehlerhaft ist.

Ein Nachteil von manuellen Tests besteht darin, dass immer eine Person für den Test benötigt wird. Dadurch ist eine manuelle Testdurchführung langsam, ermüdend und somit auch fehleranfällig.

1.3.2.2. Automatisierte Tests

Automatisierte Tests werden im Gegensatz zu manuellen Tests von einem Software-Programm durchgeführt.

Ein Vorteil von automatisierten Tests besteht darin, dass sie beliebig oft wiederholt werden können. Des Weiteren ist die Testdurchführung schnell und weniger fehleranfällig.

Ein Nachteil von automatisierten Tests besteht darin, dass man dem ausführenden Software-Programm exakt die erwartetet Ergebnisse vorgeben muss. Wird z.B. eine grafische Oberfläche getestet, kann der automatisierte Test die korrekte Funktionsweise der Anwendung sicherstellen. Das Verständnis dafür, ob das Layout korrekt ist, lässt sich hingegen sinnvoller manuell testen.

1.3.2.3. Regressionstests

Unter Regressionstests versteht man die wiederholte Durchführung von Tests. Diese Test-Technik wird eingesetzt, um die Korrektheit des zu testenden Systems nach Änderungen an diesem System zu prüfen. In der Software-Entwicklung wird diese Test-Technik durch Continuous Integration Systeme wie z.B. Hudson [Hudson Homepage, 2010] unterstützt.

1.3.3. Testframeworks

JUnit [JUnit Homepage, 2010] ist ein Testframework, das den Software-Entwickler bei der Erstellung von Tests unterstützt und eine Möglichkeit zur automatisierten Test-Durchführung bereitstellt. Der Name „JUnit" suggeriert, dass das Framework in erster Linie der Erstellung von Unit-Tests dient. Dennoch ist es auch möglich Integrationstests mit JUnit zu erstellen und durchführen zu lassen.

Neben JUnit existiert mit TestNG [TestNG Homepage, 2010] ein weiteres Testframework, das viele Parallelen zu JUnit aufweist.

1.3.3.1. Test-Ablauf

Der Ablauf bei der Durchführung von Unit-Tests ist stets identisch. Die folgende Grafik beschreibt diesen Ablauf.

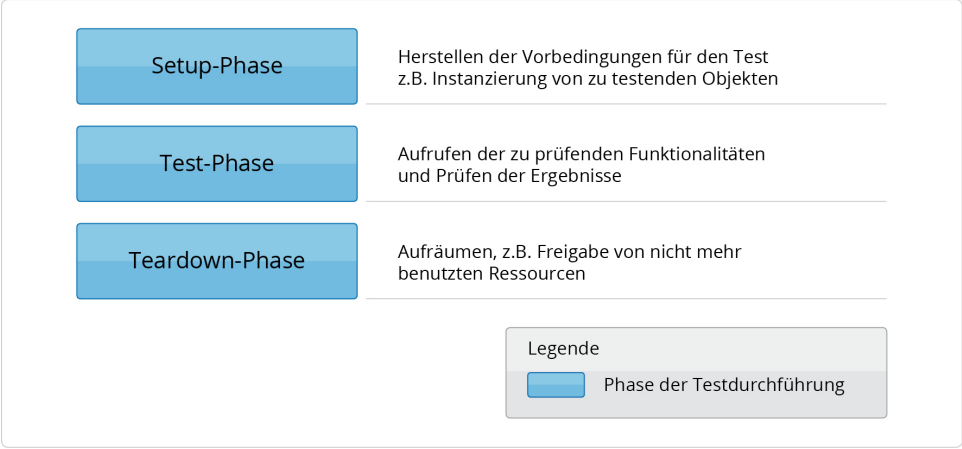


Abbildung 1.1. Ablauf automatisierte Test-Durchführung

Bei der Durchführung eines automatisierten Tests werden die drei Phasen Setup, Test und Teardown durchlaufen. Im konkreten Beispiel von JUnit3 bedeutet dies, dass zunächst die `setUp`-Methode, dann eine `test`-Methode und abschließend die `tearDown`-Methode der Testklasse aufgerufen wird. In anderen Testframeworks kann dieser Ablauf flexibler festgelegt werden, die drei Phasen werden jedoch auch dort eingehalten.

Die Setup-Phase erzeugt die Vorbedingungen, die der Test für die Ausführung benötigt. In der Test-Phase werden die zu testenden Funktionen ausgeführt. Nach der Ausführung wird die Teardown-Phase durchlaufen, um nicht länger benötigte Ressourcen wieder freizugeben.

1.3.4. Acceptance Test Driven Development (ATDD)

Das Thema Acceptance Test Driven Development (ATDD) gewinnt gerade in der agilen Welt immer stärker an Bedeutung. Der Kernaspekt von ATDD liegt in einer frühzeitigen Definition von Abnahmekriterien und -tests. Zu diesem Zweck setzen sich alle Projektbeteiligten, d.h. Fachabteilung, Software-Entwickler und Tester zusammen und besprechen, welche Kriterien für eine Abnahme der erstellte Funktionalität erfüllt sein müssen. Anhand dieser Kriterien werden konkrete Akzeptanztests definiert. Auch der umgekehrte Fall ist möglich: Zuerst werden konkrete Akzeptanztests besprochen, aus denen sich die Abnahmekriterien ergeben.

Die Vorteile von ATDD manifestieren sich an zwei Stellen innerhalb eines Projekts. Zum einen wird durch die ausgeprägte Kommunikation der Projektbeteiligten über konkrete Akzeptanztests der Spielraum für Missverständnisse stark eingeschränkt. Dadurch sinkt die Gefahr, dass fehlerhafte Umsetzungen erst in späten Projektphasen erkannt werden. Zum anderen bilden die definierten Akzeptanztests eine gute Grundlage für die Erstellung von automatisierten Tests, die in Regressionstests kontinuierlich ausgeführt werden können. In Kombination mit einer Continuous Integration Umgebung kann das korrekte fachliche Verhalten der Anwendung somit permanent sichergestellt werden. Das checkerberry test center bietet dabei eine technische Grundlage, um schnell und einfach Akzeptanztests zu automatisieren.

1.3.5. Positionierung des checkerberry test centers

Das checkerberry test center stellt mit checkerberry db und checkerberry web zwei Bibliotheken zur einfachen Erstellung von automatisierten Integrationstests zur Verfügung. Checkerberry db bietet dem Software-Entwickler die Möglichkeit, Tests mit einer integrierten Datenbank durchzuführen. Mit checkerberry web erhält der Software-Entwickler die Möglichkeit, grafische Oberflächen von Web-Anwendungen direkt im Web-Browser zu testen.

Die Integrationstests werden mit den Testframeworks JUnit oder TestNG entwickelt und lassen sich somit schnell und einfach in bestehende Prozesse integrieren. Das Testframework ist innerhalb des checkerberry test centers abstrahiert, sodass auch weitere Testframeworks unterstützt werden können.

Da das checkerberry test center ausschließlich die Erstellung von automatisierten Tests unterstützt, lassen sich die Tests beliebig oft wiederholen. Die Tests sind somit für die Durchführung von Regressionstests geeignet.

1.4. Checkerberry Maven-Repository

Die Verwendung des checkerberry test centers ist am sinnvollsten mit Maven möglich. Das Laden der benötigten Bibliotheken ist so am einfachsten möglich. Zu diesem Zweck existiert ein öffentliches Maven-Repository, das alle checkerberyy-Bibliotheken beinhaltet. Das folgende Code-Beispiel zeigt die Einbindung des Repositories in eine POM-Datei.

```
...
<!-- Im Abschnitt 'repositories' werden alle erforderlichen -->
<!-- Maven-Repositories definiert. Dort wird auch das -->
<!-- checkerberry-Repository eingetragen. -->
<repositories>
  ...
  <repository>
    <id>Checkerberry</id>
    <name>Checkerberry Test Center Repository</name>
    <url>http://repo.checkerberry.de/maven2/</url>
  </repository>
  ...
</repositories>
```

Beispiel 1.1. Beispiel checkerberry-Maven-Repository

Kapitel 2. Checkerberry db

2.1. Einleitung

Checkerberry db ist eine Bibliothek, die den Software-Entwickler bei dem Testen von Funktionen unterstützt, die Daten innerhalb einer Datenbank lesen oder manipulieren. Komplexe Aufgaben wie das Einspielen von Testdaten in die Datenbank werden innerhalb von checkerberry db gekapselt, sodass die zu entwickelnden Tests kurz, übersichtlich und wartbar bleiben. Die folgende Grafik gibt einen Überblick über die Funktionsweise von checkerberry db.

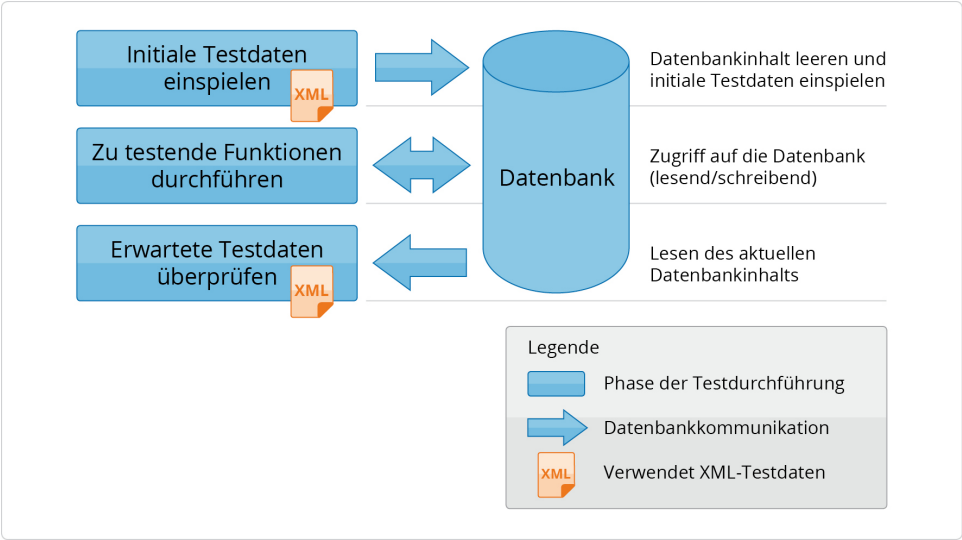


Abbildung 2.1. Überblick Testablauf checkerberry db

Bei der Durchführung eines Tests mit checkerberry db wird zunächst die Vorbedingung für den Test hergestellt. Zu diesem Zweck wird der Inhalt der Datenbank manipuliert. Der bestehende Inhalt wird gelöscht und initiale Testdaten werden eingespielt. Die einzuspielenden Testdaten werden dabei aus einer XML-Datei eingelesen. Nach dem Einspielen der Testdaten erfolgt der Aufruf der zu testenden Funktionen. Die Funktionen greifen auf die Testdaten in der Datenbank zu und können die Daten in der Datenbank manipulieren. Danach wird geprüft, ob die Datenbank die erwarteten Werte beinhaltet. Zu diesem Zweck wird der aktuelle Datenbankinhalt mit erwarteten Testdaten verglichen. Die erwarteten Testdaten werden wie die initialen Testdaten in einer XML-Datei definiert.

Checkerberry db setzt das Open-Source-Framework DbUnit [DbUnit Homepage, 2010] ein. DbUnit stellt grundlegenden Funktionalitäten zur Verfügung, um integrierte Datenbanktests durchzuführen. Allerdings erfordert das Einbinden von DbUnit in die eigenen Projekte einen hohen Anpassungsaufwand und umfangreiche Erfahrung. Ohne das notwendige DbUnit-Know-how kann dieser Weg sehr steinig sein. Im schlimmsten Fall stellt sich erst nach einigen Monaten heraus, dass der gewählte Weg langfristig nicht gangbar ist.

Innerhalb von checkerberry db wird DbUnit ausschließlich für die Kommunikation mit der Datenbank eingesetzt. Das bedeutet insbesondere, dass der in diesem Dokument beschriebene Funktionsumfang sich ausschließlich auf Funktionen von checkerberry db bezieht.

In checkerberry db stecken mehrere Jahre an Erfahrungen, um eine möglichst einfache und nachhaltige Testentwicklung zu ermöglichen. Schließlich besteht das Ziel des Software-Entwicklers in der Testentwicklung und nicht in der Entwicklung eines neuen Testframeworks.

2.2. Funktionsweise

Dieser Abschnitt beschreibt die Funktionsweise von checkerberry db anhand der drei Test-Phasen die in Abschnitt 1.3.3.1, „Test-Ablauf“beschrieben sind.

In den Code-Beispielen werden in der Regel die Spezifika der konkreten Testframeworks JUnit3, JUnit4 oder TestNG nicht aufgeführt, da sie in den meisten Fällen für das Verständnis des Beispiels nicht benötigt werden.

2.2.1. Setup-Phase

Checkerberry db wird eingesetzt, um Funktionen mit einer integrierten Datenbank zu testen. Generell benötigen Tests einen klar definierten Eingabezustand, um nach der Durchführung der zu testenden Komponente den erwarteten Ergebniszustand prüfen zu können. Im Fall von integrierten Datenbanktests bedeutet dies, dass der Inhalt der Datenbank vor dem Test in einen definierten Zustand versetzt werden muss. Zu diesem Zweck spielt checkerberry db in der Setup-Phase initiale Testdaten in die Datenbank ein.

Bevor initiale Testdaten in die Datenbank eingespielt werden können, muss checkerberry db die einzuspielenden Testdaten ermitteln. Die Definition der Testdaten erfolgt über XML-Dateien. Der Name der Datei ergibt sich implizit aus dem Namen der Testklasse und dem Namen der Testmethode: *<Name der Testklasse>_<Name der Testmethode>_initial.xml* z.B. *UserDaoTest_testInsertUser_initial.xml*. Die Ermittlung der Namen der Testdaten ist konfigurierbar (siehe Abschnitt 2.4.18, „Namenskonvention der XML-Testdaten anpassen“).

Die Testdaten werden im Klassenpfad, an der gleichen Stelle wie die zugehörige Testklasse gesucht.

```
package de.conceptpeople.sample.dao;

public class UserDaoTest extends AbstractSampleTestCase {

  /**
   * Testet das Anlegen eines neuen Users.
   * @throws Exception wenn der Test fehlschlug.
   */
  public void testInsertUser() throws Exception {
    // Neues User-Objekt erzeugen…
    User user = new User("Homer", "Simpson", "16.09.1946");
    // …und in der Datenbank speichern.
    boolean success = dao.save(user);
    // Prüfen über JUnit-Methode, ob das Speichern erfolgreich war.
    assertTrue(success);

    // Testhandler aus der Umgebung holen.
    DbTestHandler testHandler = getEnvironment().getTestHandler();

    // Die initialen Daten dürfen sich nicht geändert haben.
    testHandler.assertInitialDataUnchanged();
    // Der Datenbankinhalt muss die erwarteten Daten beinhalten.
    testHandler.assertEqualsExpected();
}
```

Beispiel 2.1. Beispiel UserDaoTest.testInsertUser

In obigen Code-Beispiel ist die Testklasse *UserDaoTest* dargestellt. Bei der Durchführung der Setup-Phase für die Testmethode *testInsertUser* werden die initialen Testdaten aufgrund von Klassen- und Methodenname in der Datei *UserDaoTest_testInsertUser_initial.xml* gesucht.

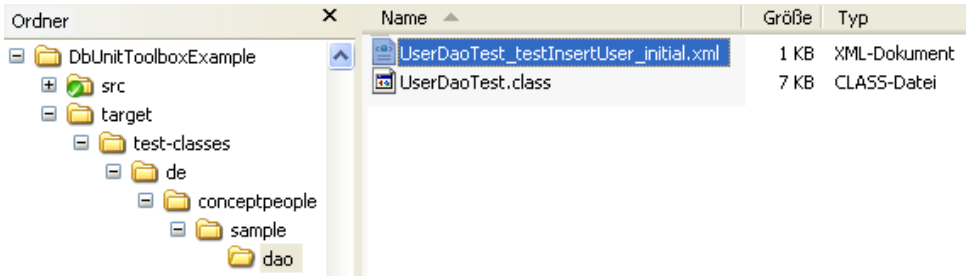


Abbildung 2.2. Screenshot - Testdaten im Klassenpfad

Aufgrund der Package-Spezifikation *de.conceptpeople.sample.dao* wird die Klasse für *UserDaoTest* bei der Kompilierung in dem Unterverzeichnis *de/conceptpeople/sample/dao* des Klassenpfads erstellt. In diesem Unterverzeichnis werden auch die Testdaten gesucht. In dem aktuellen Beispiel sind die Testdaten in der Datei *UserDaoTest_testInsertUser_initial.xml* definiert.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC "-//ConceptPeople//DTD sample-db 1.0//EN"
  "../sample-db.dtd">

<dataset>
  <USERS NAME="Bart" SURNAME="Simpson" BIRTHDATE="2009-03-18"/>
  <USERS NAME="Lisa" SURNAME="Simpson" BIRTHDATE="2009-03-18"/>
  <USERS NAME="Maggie" SURNAME="Simpson" BIRTHDATE="2009-03-18"/>
  <USERS NAME="Marge" SURNAME="Simpson" BIRTHDATE="2009-03-18"/>
</dataset>
```

Beispiel 2.2. Beispiel UserDaoTest_testInsertUser_initial.xml

Das oben angegebene Beispiel enthält eine Testdatendatei. Die Testdaten werden unter dem Tag *dataset* definiert. Innerhalb dieses Tags werden die einzuspielenden Datensätze angegeben. In dem obigen Beispiel werden vier Datensätze in die Datenbanktabelle *USERS* eingespielt. Zu diesem Zweck sind in den Testdaten vier *USERS*-Tags definiert, die die Werte der Spalten *NAME*, *SURNAME* und *BIRTHDATE* enthalten.

In vielen Situationen verwenden unterschiedliche Testmethoden einer Testklasse die gleichen initialen Testdaten. In diesen Situationen ist es sinnvoll, die Testdaten nicht auf Methoden-, sondern auf Klassenebene zu definieren. Für das obige Beispiel können die Testdaten somit ebenfalls in der Datei *UserDaoTest_initial.xml* gespeichert werden. Die Suche der initialen Testdaten erfolgt in checkerberry db in zwei Schritten: Zunächst werden die Testdaten auf Methodenebene unter dem Namen *UserDaoTest_testInsertUser_initial.xml* gesucht. Wenn diese Datei nicht vorhanden ist, werden die Testdaten auf Klassenebene unter dem Namen *UserDaoTest_initial.xml* gesucht.

Dieses Konzept minimiert den Verwaltungsaufwand und wird unter dem Namen „Convention over Configuration" in vielen Frameworks angewendet. Im konkreten Fall von checkerberry db ist nahezu kein Verwaltungsaufwand für das Einspielen der Testdaten erforderlich. Natürlich muss der Software-Entwickler die initialen Testdaten definieren. Diese Aufgabe ist jedoch in jedem Fall erforderlich. Das Testframework kann lediglich dafür Sorge tragen, dass das Einbinden der Testdaten keinen großen Aufwand erzeugt.

Wenn checkerberry db initiale Testdaten findet, wird zunächst der Inhalt aller Tabellen in der Datenbank gelöscht. Danach werden die gefundenen Testdaten in die Datenbank eingespielt. Wenn keine Testdaten gefunden werden, bleibt der Inhalt der Datenbank unverändert.

Das checkerberry test center stellt einen Caching-Mechanismus zur Verfügung, der das überflüssige Löschen und Einspielen von Tabellen verhindert. Die Funktionsweise des Caches ist in Abschnitt 2.4.14, „Performance-Optimierung durch Caching von Tabellen" beschrieben. Darüber hinaus besteht auch die

Möglichkeit, Tabellen vollständig aus der Verwendung von checkerberry db auszuschließen. Dies kann z.B. bei temporären Tabellen sinnvoll sein. Dieses Feature ist unter Abschnitt 2.4.6, „Ignorieren von Datenbanktabellen“ beschrieben.

Checkerberry db bietet einige Features, um das Löschen der gesamten Datenbanktabellen in der Setup-Phase zu umgehen (siehe Abschnitt 2.4.16, „Einspielen von Testdaten temporär unterdrücken“).

2.2.2. Test-Phase

In der Test-Phase wird der eigentliche Test in Form einer Methode aufgerufen. Da die initialen Testdaten zu diesem Zeitpunkt bereits in der Datenbank vorhanden sind, kann sich der Software-Entwickler auf die fachlichen Aspekte des Tests konzentrieren. In der Regel werden Methoden aufgerufen, die Daten aus der Datenbank lesen oder manipulieren. Die Aufgabe des Tests besteht somit zum einen darin, die Rückgabewerte der Methoden zu überprüfen. Zum anderen muss überprüft werden, ob der Zustand in der Datenbank korrekt ist.

In dem aktuellen Beispiel wird die Test-Methode `testInsertUser` aufgerufen. Diese Methode testet, ob die Methode `save` in der Klasse `UserDao` einen neuen Eintrag in der Datenbank anlegt. Zu diesem Zweck wird zunächst ein neues `User`-Objekt mit dem Namen „Homer Simpson“ und dem Geburtsdatum „16.09.1946“ erzeugt. Dieses Objekt wird an die `save`-Methode des `UserDao`-Objekts übergeben. Der Rückgabewert der `save`-Methode wird über die JUnit-Methode `assertTrue` validiert. Wenn diese Prüfung erfolgreich ist, wird die Korrektheit des Datenbankinhalts geprüft.

Zur Überprüfung des Datenbankinhalts bietet checkerberry db im Wesentlichen drei Methoden an: `assertInitialDataUnchanged`, `assertEqualsInitial` und `assertEqualsExpected`.

Die Methode `assertInitialDataUnchanged` stellt sicher, dass die initialen Testdaten in der Datenbank nicht verändert wurden. In dem aktuellen Beispiel prüft checkerberry db, ob die Einträge für Marge, Bart, Lisa und Maggie Simpson in der Datenbank mit den Werten aus den initialen Testdaten übereinstimmen. Dies ist wichtig beim Testen von lesenden Funktionen, da diese Funktionen den Datenbankinhalt nicht verändern dürfen. Gerade bei der Verwendung von Hibernate ist diese Überprüfung wichtig, da Änderungen in der Hibernate-Session auch bei lesenden Datenbankzugriffen eine Änderung in der Datenbank hervorrufen können. Das Hinzufügen neuer Datensätze wird von der Methode `assertInitialDataUnchanged` ignoriert. Es ist also zulässig, einen neuen Datensatz anzulegen und mit dieser Methode zu prüfen, ob die bereits bestehenden Datensätze von dieser Änderung unberührt blieben.

Um zu überprüfen, ob die Datenbank gar nicht verändert wurde, steht die Methode `assertEqualsInitial` zur Verfügung. Dies betrifft jedoch nur die in der mit „initial“ bezeichneten XML-File angegebenen Tabellen. Nicht aufgeführte Tabellen werden nicht überprüft.

Die Methode `assertEqualsExpected` vergleicht den aktuellen Datenbankinhalt mit den erwarteten Testdaten. Die erwarteten Testdaten werden analog zu den initialen Testdaten definiert. Allerdings wird für erwartete Testdaten das Suffix `result` anstelle von `initial` verwendet.

Für das aktuelle Beispiel werden die erwarteten Testdaten in der Datei `UserDaoTest_testInsertUser_result.xml` definiert. Diese Datei hat den folgenden Inhalt:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC "-//ConceptPeople//DTD sample-db 1.0//EN"
  "../sample-db.dtd">

<dataset>
  <USERS NAME="Bart" SURNAME="Simpson" BIRTHDATE="2009-03-18"/>
  <USERS NAME="Lisa" SURNAME="Simpson" BIRTHDATE="2009-03-18"/>
  <USERS NAME="Maggie" SURNAME="Simpson" BIRTHDATE="2009-03-18"/>
  <USERS NAME="Marge" SURNAME="Simpson" BIRTHDATE="2009-03-18"/>
  <USERS NAME="Homer" SURNAME="Simpson" BIRTHDATE="1946-09-16"/>
</dataset>
```

Beispiel 2.3. Beispiel UserDaoTest_testInsertUser_result.xml

Anhand der erwarteten Testdaten stellt checkerberry db sicher, dass die Datenbanktabelle *USERS* fünf Einträge mit den definierten Spaltenwerten beinhaltet.

Die Methode *assertEqualsExpected* überprüft nur Tabellen, die in den erwarteten Testdaten angegeben sind. Für das Beispiel bedeutet dies, dass in der Datenbank neben der *USERS*-Tabelle weitere Tabellen mit beliebigen Werten definiert sein können. Der Inhalt dieser Tabellen wird jedoch nicht geprüft, da sie nicht in den erwarteten Testdaten aufgeführt sind.

2.2.3. Teardown-Phase

In der Teardown-Phase erfolgen durch checkerberry db keine Aktionen. Checkerberry db folgt damit dem Best Practices Prinzip „Good setup don't need cleanup" (siehe Abschnitt 7.2.1, „Good setup don't need cleanup").

In der Beispiel-Implementierung werden einige Vorteile von checkerberry db deutlich: Der Software-Entwickler kann sich auf die fachlichen Aspekte der zu testenden Komponente fokussieren, da die Vorbedingungen des Tests bereits in der Setup-Phase durch das Einspielen der initialen Testdaten hergestellt werden. Dadurch bleiben die Tests kurz, übersichtlich und leicht verständlich, sodass eine gute Wartbarkeit gewährleistet ist.

Des Weiteren wird der Aufwand für die Erstellung der Tests minimiert, da das Einspielen der Testdaten keine Programmierung erfordert. Der Entwickler muss lediglich die Testdaten definieren.

2.3. Architektur

Dieses Kapitel beschreibt den Aufbau von checkerberry db und die Beziehungen der Teilkomponenten zueinander.

Checkerberry db besteht aus drei Teilkomponenten (siehe Abbildung 2.3, „Architektur checkerberry db"). Die Kernfunktionalität ist in der Core-Komponente zusammengefasst. Sie bildet die Schnittstelle für die Tests, die mit JUnit3, JUnit4 oder TestNG erstellt werden können. Welches Testframework konkret verwendet wird, wird über den Test-Connector abstrahiert.

Die Anbindung der Core-Komponente an die Datenbank erfolgt über die Bridge-Komponente. Die Bridge-Komponente stellt zusätzlich eine optionale Komponente zur Verfügung, die das Auffinden von Dateien im Klassenpfad ermöglicht.

Während die Core-Komponente unabhängig von der Kundenumgebung ist, kapselt die Bridge-Komponente alle Funktionen, die auf die aktuelle Kundenumgebung zurückgreifen. Somit muss bei jeder Installation von checkerberry db eine eigene Bridge-Komponente implementiert werden (siehe Abschnitt 2.5, „Installation"). Der Implementierungsaufwand ist jedoch gering, da die Bridge-Komponente im Wesentlichen die Verbindung zur Datenbank herstellt, was über eine JDBC-Verbindung erfolgt. Des Weiteren existieren für die gängigsten Konfigurationen bereits Basis-Implementierungen z.B. im Spring/Hibernate-Umfeld.

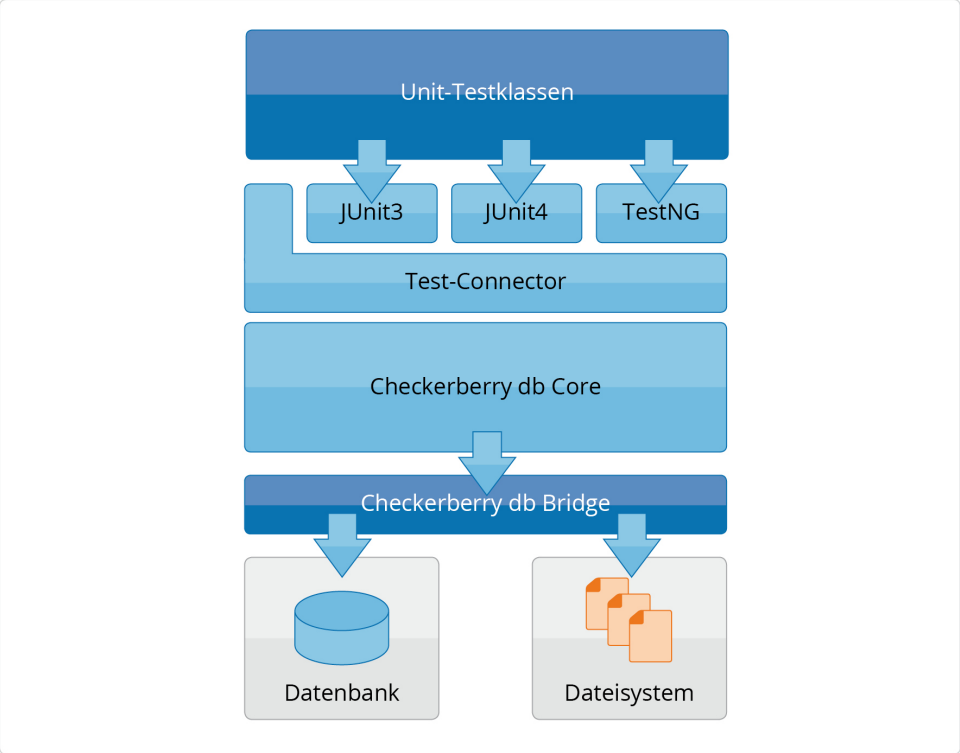


Abbildung 2.3. Architektur checkerberry db

Die Abbildung skizziert die Architektur von checkerberry db. Die dunkel eingefärbten Komponenten sind abhängig von der aktuellen Kundenumgebung und werden somit vom Kunden implementiert. Die erstellten Unit-Testklassen greifen über den Test-Connector auf die Core-Komponente von checkerberry db zu. Der Core umfasst das Laden von initialen Testdaten, die Verwendung aller Features und den Vergleich des Datenbankinhalts mit erwarteten Ergebnismengen. Die Core-Komponente verwendet für den Zugriff auf die Datenbank und das Dateisystem die implementierte Bridge-Komponente.

2.4. Features

Checkerberry db bietet eine Reihe von Features, die das Entwickeln von automatisierten Integrationstests auf mehreren Ebenen vereinfacht. Der Programmieraufwand wird minimiert, da grundlegende Aufgaben direkt von dem checkerberry test center übernommen werden. Dies umfasst zum Beispiel das Laden von initialen Testdaten, was bereits in der Setup-Phase des Tests erfolgt. Dadurch ist der Entwickler in der Lage, sich auf die fachlichen Aspekte in den Tests zu konzentrieren. Die Tests werden dadurch übersichtlich, verständlich und wartbar.

Um die Effizienz bei der Entwicklung von automatisierten Tests signifikant zu steigern, ist die reine Minimierung des Programmieraufwands nicht ausreichend. Ein Großteil des Aufwands bei der Entwicklung von automatisierten Tests entsteht bei der Erstellung der initialen und erwarteten Testdaten und vor allem bei der Fehleranalyse. Aus diesem Grund verfügt checkerberry db ebenfalls über Funktionen, die die Testdatenerstellung und die Fehleranalyse vereinfachen.

2.4.1. Verwendung von XML-Testdaten

In checkerberry db werden die Testdaten in einem einfachen XML-Format definiert. Im Folgenden wird der Aufbau der Testdaten näher erläutert.

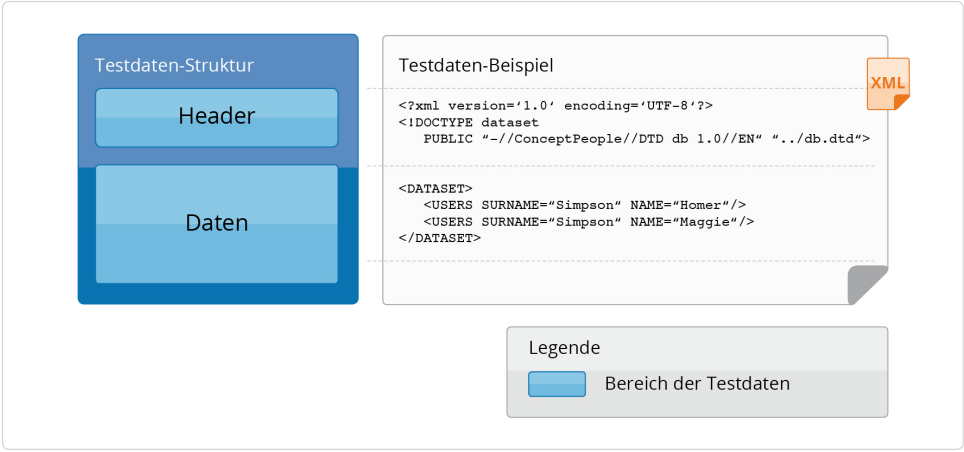


Abbildung 2.4. Testdaten-Struktur

Wie in Abbildung 2.4, „Testdaten-Struktur" dargestellt, bestehen Testdaten aus einem Header- und einem Datenbereich.

2.4.1.1. Header-Bereich der Testdaten

Der Header-Bereich besteht aus dem XML-Tag, der die XML-Version der Datei und das Encoding spezifiziert. Das DOCTYPE-Tag beschreibt, mittels welcher DTD checkerberry db die Korrektheit der Testdaten überprüfen kann.

Die DTD-Informationen werden in einer eigenständigen Datei erwartet, die über einen öffentlichen Bezeichner (*public identifier*) oder über einen expliziten Speicherort angegeben werden kann. Der öffentliche Bezeichner (im Beispiel *"-//ConceptPeople//DTD db 1.0//EN"*) kann den individuellen Bedürfnissen angepasst werden: *ConceptPeople* ist die Angabe der zu veröffentlichenden Person oder Institution und *db* der Name der DTD, welcher mit dem Dateinamen der DTD (ohne Endung) übereinstimmen sollte.

Der öffentliche Identifikator hat eine höhere Priorität als der explizite Speicherort der Datei. Der explizite Speicherort wird lediglich in den Entwicklungsumgebungen verwendet, wenn der öffentliche Identifikator unbekannt ist. In der Eclipse IDE können unter `Windows->Preferences/XML/XML-Catalog` öffentliche Identifikatoren zu DTD-Dateien zugeordnet werden. Dieses Vorgehen bietet sich insbesondere an, wenn mehrere Teilprojekte dieselbe DTD verwenden, da dann die relative Angabe der zu verwendenden DTD-Datei ohne die Überschreitung von Projektgrenzen z.B. *../../base-project/src/main/resources/de/conceptpeople/db.dtd* nicht möglich ist.

Checkerberry db verwendet ausschließlich den öffentlichen Identifikator für die Ermittlung der DTD. Die Zuordnung von öffentlichem Identifikator zu der entsprechenden Datei erfolgt während der Konfiguration von checkerberry db (siehe Abschnitt 2.4.1.4, „Konfiguration der Testdaten-DTD"). Da checkerberry db die DTD immer im Klassenpfad sucht, können verschiedene Teilprojekte dieselbe DTD verwenden. Die DTD kann dann in einem gemeinsamen Elternprojekt definiert werden, sodass alle Teilprojekte auf dieselbe DTD zugreifen können. Bei der Angabe eines relativen Dateinamens würde dies nicht funktionieren, was eine Duplizierung der DTD nach sich ziehen würde.

2.4.1.2. Datenbereich der Testdaten

Der Datenbereich der initialen Testdaten beinhaltet die Informationen, die in die Datenbank eingespielt werden sollen. Bei den erwarteten Testdaten beinhaltet der Datenbereich die Informationen, die mit dem Datenbankinhalt verglichen werden sollen. In beiden Fällen enthält der Datenbereich somit Informationen, die sich auf den Datenbankinhalt beziehen.

Wie aus Abbildung 2.4, „Testdaten-Struktur“ hervorgeht, wird der Datenbankinhalt innerhalb des *<dataset>*-Tags spezifiziert. Die Syntax der darunterliegenden Tags ergibt sich aus der Datenbankstruktur, die über die DTD festgelegt wird. In dem Beispiel wird die Tabelle *USERS* mit Daten gefüllt, und zwar in den Spalten *SURNAME* und *NAME*. Für jeden Datensatz wird ein eigenes *USERS*-Tag verwendet.

Bei dem Einspielen von mehreren Tabellen in die Datenbank muss die Reihenfolge der einzuspielenden Tabellen berücksichtigt werden, um Constraint-Verletzungen in der Datenbank zu vermeiden. Diese Situation tritt dann auf, wenn ein Datensatz eingespielt wird, der einen anderen Datensatz referenziert. Ist der referenzierte Datensatz noch nicht in der Datenbank vorhanden, kommt es zu einem Fehler beim Einspielen.

Um die Eingabe der Testdaten optimal zu unterstützen, wird die korrekte Reihenfolge in der zugehörigen DTD festgelegt. Eventuelle Fehler können bereits bei der Validierung der XML-Datei gegen die DTD (zum Beispiel mit den Eclipse Validatoren) aufgedeckt werden.

2.4.1.3. Definieren der Testdatenstruktur als DTD

Die Testdaten-DTD beschreibt die Struktur der Testdaten und ergibt sich somit aus der Struktur der Datenbank. Die DTD definiert eine Reihenfolge der enthaltenen Tabellen, definiert die Spalten jeder Tabelle und speichert die Information, ob eine Spalte erforderlich ist (`not null`). Die Reihenfolge der Tabellen in der DTD legt die Insert-Reihenfolge beim Einspielen der Testdaten fest. Aus diesem Grund müssen die Abhängigkeiten der Tabellen untereinander berücksichtigt werden, damit bei dem Einspielen der Testdaten keine Constraint-Verletzungen in der Datenbank auftreten.

```
<!-- Definition der enthaltenen Tabellen unter Berücksichtigung -->
<!-- der Foreign-Key-Beziehungen. -->
<!ELEMENT dataset (
  USERS*,
  ADDRESS*,
  COUNTRY*)>

<!-- Die Tabelle USERS hat keine Kind-Elemente (gilt generell für -->
<!-- alle Tabellen). -->
<!ELEMENT USERS EMPTY>

<!-- Definition der Spalten der Tabelle USERS. -->
<!ATTLIST USERS
  <!-- Not nullable Spalten werden durch #REQUIRED definiert. -->
  NAME CDATA #REQUIRED
  SURNAME CDATA #REQUIRED
  <!-- Nullable Spalten werden durch #IMPLIED definiert. -->
  BIRTHDATE CDATA #IMPLIED>
…
```

Beispiel 2.4. Beispiel Testdaten-DTD

Das dargestellte Code-Beispiel enthält eine beispielhafte DTD für eine Testdaten-XML. Am Anfang der DTD wird definiert, welche Elemente das Wurzelelement `dataset` beinhalten kann. In dem konkreten Beispiel kann das `dataset`-Element eine beliebige Anzahl von `USERS`-, `ADDRESS`- und `COUNTRY`-Elementen beinhalten. Da sich die DTD auf eine Datenbankstruktur bezieht, handelt es sich bei den Kind-Elementen (`USERS`, `ADDRESS` und `COUNTRY`) um Namen der zugehörigen Datenbanktabellen.

Die Reihenfolge der Elemente wird durch die DTD fest vorgegeben. Für eine XML-Datei, die der angegebenen DTD genügt, bedeutet dies, dass als erstes alle `USERS`-Elemente, danach alle `ADDRESS`-Elemente und zum Schluss alle `COUNTRY`-Elemente angegeben werden müssen. Auf diese Art und Weise wird auch die Reihenfolge festgelegt, in der die Testdaten in die Datenbank eingespielt werden. Bei der Definition der Reihenfolge müssen die Abhängigkeiten der Tabellen in der Datenbank berücksichtigt werden. Anderenfalls kann das Einspielen der Testdaten zu Constraint-Verletzungen führen, da ggf. ein eingefügter Datensatz auf einen anderen Datensatz verweist, der noch nicht eingespielt wurde.

Nach der Definition des `dataset`-Elements erfolgt die Definition der Kind-Elemente. In dem konkreten Beispiel wird das `USERS`-Element definiert. Zunächst wird festgelegt, dass das `USERS`-Element keine Kind-Elemente besitzt. Diese Angabe ist obligatorisch für alle Tabellen-Definitionen, da die Informationen der Tabellen nicht über XML-Kind-Elemente sondern über XML-Attribute angegeben werden. Für das `USERS`-Element werden in der DTD die Attribute `NAME`, `SURNAME` und `BIRTHDATE` definiert. Es handelt sich dabei um die Spaltennamen der `USERS`-Tabelle. Durch die Eigenschaft `#REQUIRED` werden Pflichtfelder (`not null`) markiert. Optionale Spalten werden über `#IMPLIED` markiert.

Die DTD wird von checkerberry db verwendet, um die zu vergleichenden Tabellen und Spalten zu ermitteln. Obwohl die Verwendung einer DTD unter DbUnit nicht vorgeschrieben ist, verwendet checkerberry db immer eine DTD. Wenn keine DTD vorhanden ist, fehlt DbUnit die Information, welche Spalten in einer Tabelle enthalten sind. Aus diesem Grund verwendet DbUnit die erste Zeile einer Tabelle in den Testdaten zur Ermittlung der vorhandenen Spalten. Dieses Vorgehen kann jedoch zu folgenden Problemen führen.

In DbUnit-Testdaten werden `null`-Werte dadurch „dargestellt“, dass das entsprechende Spalten-Tag in den XML-Daten fehlt. Ohne DTD kann DbUnit in der ersten Zeile einer Tabelle somit nicht unterscheiden, ob eine Spalte fehlt oder ob der entsprechende Wert `null` ist. Diese „Logik“ kann zu verwirrenden Ergebnissen bei der Überprüfung der Testdaten führen. Aus diesem Grund ist eine sinnvolle und nachvollziehbare Verwendung von DbUnit ohne DTD kaum möglich.

Die DTD wird komplett automatisch erstellt und aktualisiert, sodass eine neue DTD nur noch manuell an die richtige Stelle des Ressource-Baumes des Testprojektes verschoben werden muss.

Die DTD-Erstellung aus großen Datenbanken ist sehr langsam, sodass man geänderte DTD-Dateien möglichst schnell in den Ressource-Baum übernehmen sollte. Insbesondere ist es nicht ratsam, die DTD bei jedem Testlauf neu aus der Datenbank generieren zu lassen.

2.4.1.4. Konfiguration der Testdaten-DTD

In der Konfiguration wird die Verbindung des öffentlichen Identifikators zu der tatsächlichen DTD-Datei hergestellt.

```
// Setzen der Zuordnung von öffentlichem Identifikator und der zugehörigen
// DTD-Datei. Die DTD-Datei wird dabei im Klassenpfad gesucht.
configuration.setDatabaseDtd("-//ConceptPeople//DTD sample-db 1.0//EN",
  "de/conceptpeople/sample/sample-db.dtd");
```

Beispiel 2.5. Konfiguration der Testdaten-DTD

Das obige Beispiel zeigt die Konfiguration einer DTD in checkerberry db. Durch den Aufruf der Methode `setDatabaseDtd` wird checkerberry db die Zuordnung zwischen öffentlichem Identifikator (`-//ConceptPeople//DTD sample-db 1.0//EN`) und der DTD-Datei (`de/conceptpeople/sample/sample-db.dtd`) mitgeteilt. Die DTD-Datei wird dabei stets im Klassenpfad gesucht.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC "-//ConceptPeople//DTD sample-db 1.0//EN" "…">
```

Beispiel 2.6. Referenzierung der DTD in den Testdaten.

Diese Konfiguration der DTD ist für die Verwendung von checkerberry db zwingend erforderlich.

2.4.1.5. Anlegen einer neuen Testdaten-DTD

Bei der Installation von checkerberry db wurde der Name der zu verwendenden Testdaten-DTD konfiguriert (siehe auch Abschnitt 2.4.1.4, „Konfiguration der Testdaten-DTD“). Die DTD-Datei existiert zu diesem Zeitpunkt in der Regel noch nicht. Die DTD kann jedoch schnell bei der Erstellung des ersten Testfalls aus der bestehenden Datenbank generiert werden.

```
DbTestHandler testHandler = getEnvironment().getTestHandler();
testHandler.createDtd("c:/tmp/sample-db.dtd");
```

Beispiel 2.7. Erzeugung einer DTD

Mit den Zeilen weißt man checkerberry db an, eine neue DTD zu erzeugen und unter dem gegebenen Dateinamen (hier *c:/tmp/sample-db.dtd*) abzulegen. Zu diesem Zweck wird die Datenbankstruktur aus der Datenbank ermittelt und die gegenseitigen Abhängigkeiten der Tabellen berechnet. Mit Hilfe dieser Informationen erzeugt checkerberry db die neue DTD.

Nach dem Erzeugen muss die DTD lediglich noch in den konfigurierten Pfad kopiert werden. Die beiden Zeilen zu Ihrer Erzeugung müssen aus dem Test wieder entfernt werden.

2.4.1.6. Automatische Aktualisierung der Testdaten-DTD

Bei einer Änderung der Datenbankstruktur muss auch die DTD angepasst werden. Ärgerlich kann es sein, wenn diese Anpassung vergessen wird und aus diesem Grund zahlreiche Tests in der Continuous Integration Umgebung fehlschlagen. Gerade bei großen Test-Suites mit einer langen Laufzeit kann dieser Informationsverlust schmerzhaft sein. Und natürlich passiert das immer zu den ungünstigsten Zeitpunkten.

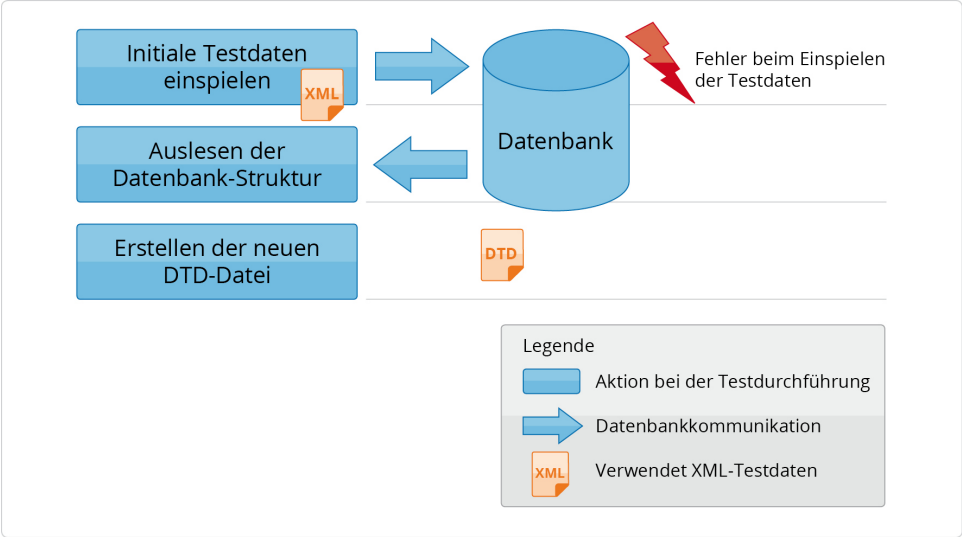


Abbildung 2.5. Aktualisierung der DTD

Abbildung 2.5, „Aktualisierung der DTD" skizziert das Verhalten von checkerberry db für den Fall, dass initiale Testdaten bei der Durchführung eines Tests nicht eingespielt werden können. In der Setup-Phase des durchzuführenden Tests versucht checkerberry db die initialen Testdaten in die Datenbank einzuspielen. Dieser Versuch schlägt fehl. Checkerberry db geht in dieser Situation davon aus, dass die DTD nicht mehr die tatsächliche Datenbankstruktur beschreibt. Checkerberry db ermittelt daraufhin die tatsächliche Datenbankstruktur aus der Datenbank und erstellt anhand dieser Informationen eine neue DTD. Diese wird im Klassenpfad unter demselben Pfad abgelegt wie die alte DTD und lediglich mit dem Namenspräfix „*new-*" versehen. Alle nachfolgenden Tests verwenden die neu generierte DTD.

Das beschriebene Vorgehen hat den Vorteil, dass kein kompletter nächtlicher Testlauf durch Nachlässigkeit verloren geht. Es wäre sinnlos, alle Tests mit der vorherigen DTD auszuführen, wenn davon ausgegangen wird, dass diese DTD fehlerhaft ist.

Der fehlgeschlagene Test, der die Aktualisierung der DTD hervorgerufen hat, wird nicht erneut ausgeführt. Er dient als Erinnerung, dass eine neue DTD erforderlich ist.

Die Erzeugung der DTD aus der Datenbank kann je nach Größe der Datenbank einige Minuten in Anspruch nehmen. Das Ziel sollte somit sein, dass die verwendete DTD-Datei immer den aktuellen Stand der Datenbank widerspiegelt, damit die Tests schnell ausgeführt werden. Insbesondere ist es nicht sinnvoll, die DTD vor jedem Testlauf oder vor jedem Test aus der Datenbank zu generieren. Zum einen verlängern sich auf diese Art und Weise die Ausführungszeiten. Zum anderen benötigen die Entwickler für die Erstellung der Testdaten eine aktuelle DTD, um die Korrektheit der Testdaten zu validieren. Darüber hinaus bieten viele IDEs durch die DTD eine Autovervollständigung bei der Eingabe der Testdaten an.

2.4.1.7. Tool-Unterstützung durch DTD-Verwendung

Die meisten Entwicklungsumgebungen unterstützen den Entwickler bei dem Editieren von XML-Dateien, wenn eine DTD definiert ist. Dazu gehört neben der Auto-Vervollständigung vor allem auch die syntaktische Validierung der Daten. Dies vereinfacht die Erstellung der Testdaten.

Abbildung 2.6. Auto-Vervollständigung XML-Tag

Abbildung 2.6, „Auto-Vervollständigung XML-Tag“zeigt das Beispiel für eine Auto-Vervollständigung in der Eclipse IDE. Die IDE schlägt dem Benutzer die XML-Tags vor, die in die Testdaten eingefügt werden können. Die Beispiel-Datenbank enthält nur die Tabelle `USERS`. Zusätzlich können jedoch auch `INCLUDE`-Einträge eingegeben werden.

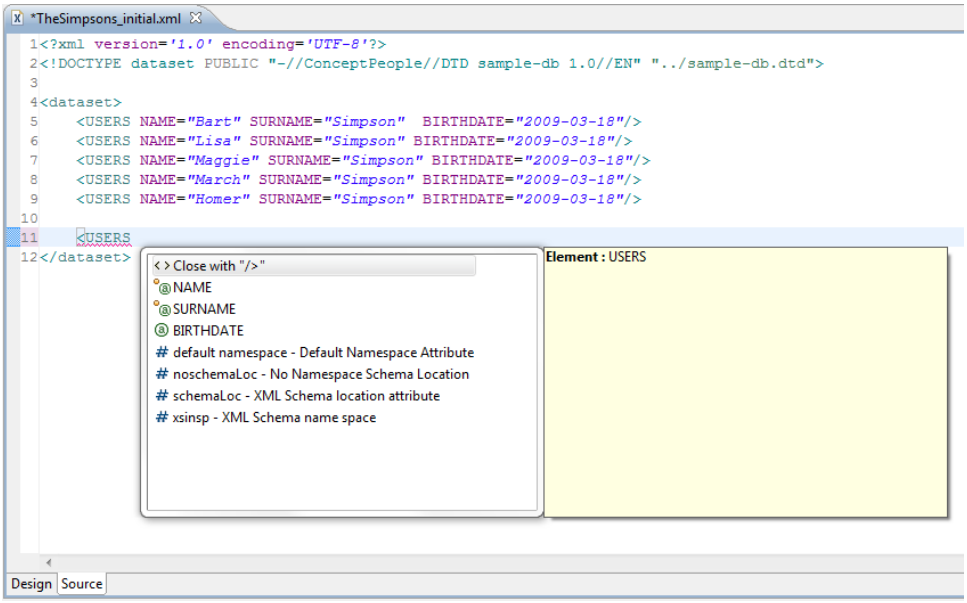


Abbildung 2.7. Auto-Vervollständigung XML-Attribut

Abbildung 2.7, „Auto-Vervollständigung XML-Attribut“zeigt die Auto-Vervollständigung, nachdem ein XML-Tag ausgewählt wurde. In diesem Fall bietet die IDE dem Benutzer die XML-Attribute `NAME`, `SURNAME` und `BIRTHDATE` zur Auswahl an. Es handelt sich dabei um die Tabellenspalten der zugehörigen `USERS`-Tabelle. Die XML-Attribute `NAME` und `SURNAME` sind in dem Kontextmenü mit einem kleinen Punkt markiert. Dieser Punkt wird zur Markierung von Pflichtfeldern verwendet. Die Angaben `NAME` und `SURNAME` müssen somit für jeden `USERS`-Eintrag angegeben werden.

In dem ersten Beispiel wurden auch die XML-Tags `INCLUDE` und `EMPTY_TABLE` durch die Auto-Vervollständigung zur Auswahl angeboten, obwohl es sich nicht um Tabellen aus der Datenbank handelt. Die Auto-Vervollständigung orientiert sich an der DTD und die enthält die Beschreibung des `INCLUDE`- und `EMPTY_TABLE`-Tags. Die zusätzliche Beschreibung dieser beiden Tags wird durch checkerberry db eingefügt, wenn die DTD aus den Datenbankinformationen generiert wird.

2.4.2. Zuordnung von Testdaten und tatsächliche Daten

Bei dem Vergleich der erwarteten Testdaten mit dem aktuellen Datenbankinhalt stellt checkerberry db fest, welche Tabellenzeilen hinzugefügt, gelöscht oder geändert wurden. Zu diesem Zweck stellt checkerberry db eine eindeutige Zuordnung zwischen den Tabellenzeilen in der Datenbank und den Zeilen der Testdaten her. Die Zuordnung erfolgt über ausgewählte Spalten, die in checkerberry db als Lookup-Keys bezeichnet werden. Die Lookup-Keys werden für jede Datenbanktabelle definiert.

Lookup-Keys sind konzeptionell identisch mit Primary Keys. In beiden Fällen ist das Ziel die Identifikation von Datensätzen. Da die Lookup-Keys einer Tabelle von ihren Primary Keys abweichen können, wurde der neue Begriff Lookup-Keys für eine bessere Unterscheidbarkeit in checkerberry db eingeführt.

Im Gegensatz zu den Primary Keys definieren die Lookup-Keys in der Regel fachliche Schlüssel, da diese in den Testdaten intuitiver zu verwenden sind. Die Verwendung von fachlichen Schlüsseln hat den weiteren Vorteil, dass die Werte bereits zum Zeitpunkt der Testerstellung bekannt sind. Primary Keys werden hingegen häufig in der Datenbank generiert, sodass deren Werte zum Zeitpunkt der Testdatenerstellung noch nicht feststehen.

Lookup-Keys werden durch eine Liste von Spaltennamen definiert. Eine Tabellenzeile wird einer Zeile aus den Testdaten zugeordnet, wenn die Werte der Lookup-Keys dieser beiden Zeilen übereinstimmen. Die Lookup-Keys müssen somit jede Zeile der Testdaten und der zugehörigen Tabelle eindeutig identifizieren.

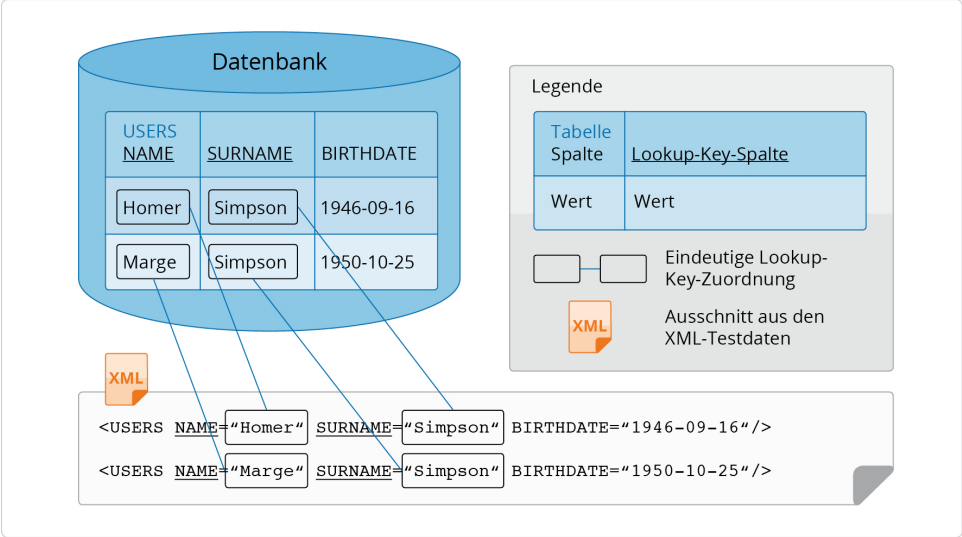


Abbildung 2.8. Lookup-Keys

Abbildung 2.8, „Lookup-Keys" stellt eine Zuordnung von Datenbankinhalt und Testdaten anhand von Lookup-Keys dar. Die Datenbank enthält zwei Einträge in der Tabelle *USERS*. Als Lookup-Key werden die Spalten *NAME* und *SURNAME* verwendet. In den Testdaten sind ebenfalls zwei Datensätze für die Tabelle *USERS* definiert. Über die Werte der Lookup-Keys kann eine eindeutige Zuordnung zwischen den Daten in der Datenbank und den Testdaten hergestellt werden.

Die Lookup-Keys können innerhalb einer Test-Methode einfach geändert werden, wie das folgende Code-Beispiel zeigt.

```
public void testGetUser() throws Exception {
  …
  // Holen des Testhandlers.
  DbTestHandler testHandler = getEnvironment().getTestHandler();
  // Allgemeine Datenbankbeschreibung holen.
  DatabaseDescription dbDescription = testHandler.getDatabaseDescription();
  // Tabellenbeschreibung für Tabelle USERS holen.
  DatabaseTableDescription tableDescription
      = dbDescription.getTableDescription("USERS");
  // Lookup-Keys ändern: Neuer Lookup-Key ist die Spalte SURNAME.
  tableDescription.setLookupKeyColumns("SURNAME");
  …
}
```

Beispiel 2.8. Anpassen der Lookup-Keys

Über die checkerberry db-Umgebung ist der Zugriff auf die Datenbank- und Tabellenbeschreibungen möglich. In dem Beispiel wird der Lookup-Key für die Tabelle *USERS* auf die Spalte *SURNAME* gesetzt. In den anschließenden Vergleichen verwendet checkerberry db diesen neuen Lookup-Key. Da jeder Test eine eigene Kopie der Datenbankbeschreibung verwendet, hat diese Umstellung keine Auswirkungen auf nachfolgende Testfälle.

Die Anpassung im obigen Beispiel wirkt auf den ersten Blick fehlerhaft, da sie voraussetzt, dass die Nachnamen in der Tabelle *USERS* eindeutig sind. Es muss dabei allerdings berücksichtigt werden, dass die Eindeutigkeit lediglich für den aktuellen Test erforderlich ist. Die Eindeutigkeit ist somit auf die Testdaten des entsprechenden Tests beschränkt. Dieser Sachverhalt ist sehr hilfreich, da für spezielle Tests die Lookup-Keys individuell angepasst werden können.

2.4.3. Überprüfen der Ergebnisse durch Validatoren

Innerhalb von checkerberry db werden erwartete Testdaten mit den Werten aus der Datenbank verglichen. Zu diesem Zweck wird zunächst eine Zuordnung einer Zeile aus den Testdaten mit der zugehörigen Zeile aus der Datenbank vorgenommen. Die Werte der einzelnen Spalten werden dann verglichen. Der Vergleich der Spaltenwerte erfolgt dabei über Validatoren.

Checkerberry db verwendet die folgenden Standard-Validatoren in der angegebenen Reihenfolge:

1. TimeValidator
2. DateValidator
3. NumberValidator
4. StringValidator

Jeder Validator überprüft anhand des Werts aus den erwarteten Testdaten, ob er für die Validierung in Frage kommt. Der erwartete Wert „`2010-01-01 12:00:00.0`“ würde beispielsweise durch den TimeValidator validiert werden, während der Wert „`1.0`“ durch den NumberValidator validiert werden würde. Die genaue Funktionsweise der Standard-Validatoren wird in Abschnitt 2.4.3.1, „Verwendung der Standard-Validatoren“ ausführlicher beschrieben.

Bei der Überprüfung eines Wertes werden die Validatoren der Reihe nach durchlaufen. Im Folgenden wird dies am Beispiel des erwarteten Wertes „`2010-45-90`“ erläutert.

Zunächst prüft der TimeValidator, ob er für die Überprüfung zuständig ist. Da der erwartete Wert nicht dem Muster des TimeValidators entspricht, wird die Anfrage an den nächsten Validator in der Liste weitergereicht. Der DateValidator ist aufgrund des Musters für die Überprüfung des Wertes zuständig. Der DateValidator konvertiert daraufhin den erwarteten Wert und den entsprechenden Wert aus der Datenbank in ein *`java.util.Date`*-Objekt. Diese Konvertierung scheitert, da „`2010-45-90`“ kein gültiges Datum ist. Da bereits die Konvertierung fehlgeschlagen ist, scheint der Validator trotz des übereinstimmenden Musters doch nicht für die Überprüfung zuständig gewesen zu sein. Aus diesem Grund wird die Anfrage an den nächsten Validator weitergereicht. Der NumberValidator fühlt sich aufgrund des Musters nicht für die Validierung zuständig. Daher wird die Anfrage an den letzten Validator, den StringValidator, weitergereicht. Der StringValidator ist für alle Werte zuständig und führt die Überprüfung in jedem Fall durch. In diesem Fall wird die Zeichenkette „`2010-45-90`“ mit dem entsprechenden Wert aus der Datenbank als Zeichenkette verglichen. Wenn die Werte nicht übereinstimmen, wird der Test mit einer entsprechenden Fehlermeldung beendet.

Eine Besonderheit bei der Validierung in obigem Beispiel besteht in der Behandlung von Konvertierungsfehlern. Ein Validator hat bei der Überprüfung eines Wertes drei mögliche Ergebnisse:

1. die Werte sind gemäß der speziellen Prüfung identisch
2. die Werte sind gemäß der speziellen Prüfung unterschiedlich
3. der Validator ist für die Überprüfung doch nicht zuständig

In dem dritten Fall wird die Anfrage an den folgenden Validator weitergeleitet, während die Überprüfung in den ersten beiden Fällen beendet wird.

2.4.3.1. Verwendung der Standard-Validatoren

Checkerberry db enthält die vier Standard-Validatoren TimeValidator, DateValidator, NumberValidator und StringValidator, die im Folgenden beschrieben werden.

TimeValidator (de.conceptpeople.checkerberry.common.validation.OperatorContainingTimeValidator)

Der TimeValidator wird verwendet, um Datumsangaben inkl. Uhrzeit zu prüfen. Zu diesem Zweck werden die zu vergleichenden Werte in *java.util.Date*-Objekte konvertiert und als Date verglichen. Dies führt z.B. dazu, dass die Werte „2010-01-01 12:00:00.001" und „2010-01-01 12:00:00.1" identisch sind, was bei einem Vergleich als reine Zeichenketten nicht der Fall wäre.

Der TimeValidator wird verwendet, wenn der erwartete Wert dem Muster [op] yyyy-MM-dd HH:mm:ss.S[SS] genügt, wobei op ein Operator ist. Die folgende Tabelle zeigt die Bedeutungen der Operatoren.

Name	Bedeutung	Beispiel
lt	Kleiner als (less than)	lt 2010-01-01 00:00:00.0 überprüft, ob der Wert in der Datenbank vor dem 01.01.2010 liegt.
le	Kleiner als oder gleich (less than or equal)	le 2010-01-01 00:00:00.0 überprüft, ob der Wert in der Datenbank vor dem 01.01.2010 liegt oder ob er genau diesem Zeitpunkt entspricht.
gt	Größer als (greater than)	gt 2010-01-01 00:00:00.0 überprüft, ob der Wert in der Datenbank hinter dem genannten Zeitpunkt liegt.
ge	Größer als oder gleich (greater than or equal)	ge 2010-01-01 00:00:00.0 überprüft, ob der Wert in der Datenbank hinter dem genannten Zeitpunkt liegt oder dem Zeitpunkt entspricht.
eq	Gleich (equal)	eq 2010-01-01 00:00:00.0 überprüft, ob der Wert in der Datenbank genau dem genannten Zeitpunkt entspricht. Dies ist der Standard-Operator, wenn kein Operator explizit angegeben wurde.
ne	Ungleich (not equal)	ne 2010-01-01 00:00:00.0 überprüft, ob der Wert in der Datenbank nicht dem genannten Zeitpunkt entspricht.

DateValidator (de.conceptpeople.checkerberry.common.validation.OperatorContainingDateValidator)

Der DateValidator wird verwendet, um Datumsangaben ohne Uhrzeit zu prüfen. Zu diesem Zweck werden die zu vergleichenden Werte in *java.util.Date*-Objekte konvertiert und als Date verglichen. Dies führt z.B. dazu, dass die Werte „2010-01-01" und „2010-01-01 00:00:00.0" identisch sind, was bei einem Vergleich als reine Zeichenketten nicht der Fall wäre.

Der DateValidator wird verwendet, wenn der erwartete Wert dem Muster [op] yyyy-MM-dd genügt, wobei op ein Operator ist. Die folgende Tabelle zeigt die Bedeutungen der Operatoren.

Name	Bedeutung	Beispiel
lt	Kleiner als (less than)	lt 2010-01-01 überprüft, ob der Wert in der Datenbank vor dem 01.01.2010 liegt.

Name	Bedeutung	Beispiel
le	Kleiner als oder gleich (less than or equal)	`le 2010-01-01` überprüft, ob der Wert in der Datenbank vor dem 01.01.2010 liegt oder ob er genau diesem Datum entspricht.
gt	Größer als (greater than)	`gt 2010-01-01` überprüft, ob der Wert in der Datenbank hinter dem genannten Datum liegt.
ge	Größer als oder gleich (greater than or equal)	`ge 2010-01-01` überprüft, ob der Wert in der Datenbank hinter dem genannten Datum liegt oder dem Datum entspricht.
eq	Gleich (equal)	`eq 2010-01-01` überprüft, ob der Wert in der Datenbank genau dem genannten Datum entspricht. Dies ist der Standard-Operator, wenn kein Operator explizit angegeben wurde.
ne	Ungleich (not equal)	`ne 2010-01-01` überprüft, ob der Wert in der Datenbank nicht dem genannten Datum entspricht.

NumberValidator
(de.conceptpeople.checkerberry.common.validation.OperatorContainingNumberValidator)

Der NumberValidator wird verwendet, um Zahlen zu prüfen. Zu diesem Zweck werden die zu vergleichenden Werte in *`java.math.BigDecimal`*-Objekte konvertiert und als `BigDecimal` verglichen. Dies führt z.B. dazu, dass die Werte „`1.0`“ und „`1`“ identisch sind, was bei einem Vergleich als reine Zeichenketten nicht der Fall wäre.

Der NumberValidator wird verwendet, wenn der erwartete Wert dem Muster `[op] n.n` genügt, wobei `op` ein Operator ist. Die folgende Tabelle zeigt die Bedeutungen der Operatoren.

Name	Bedeutung	Beispiel
lt	Kleiner als (less than)	`lt 1.6` überprüft, ob der Wert in der Datenbank kleiner als 1.6 ist.
le	Kleiner als oder gleich (less than or equal)	`le 1.6` überprüft, ob der Wert in der Datenbank kleiner als oder gleich 1.6 ist.
gt	Größer als (greater than)	`gt 1.6` überprüft, ob der Wert in der Datenbank größer als 1.6 ist.
ge	Größer als oder gleich (greater than or equal)	`ge 1.6` überprüft, ob der Wert in der Datenbank größer als oder gleich 1.6 ist.
eq	Gleich (equal)	`eq 1.6` überprüft, ob der Wert in der Datenbank gleich 1.6 ist. Dies ist der Standard-Operator, wenn kein Operator explizit angegeben wurde.

Name	Bedeutung	Beispiel
ne	Ungleich (not equal)	`eq 1.6` überprüft, ob der Wert in der Datenbank ungleich 1.6 ist.

StringValidator (de.conceptpeople.checkerberry.common.validation.DefaultStringValidator)

Der StringValidator wird verwendet, um Werte als Zeichenkette zu vergleichen. Er erwartet kein besonderes Muster, sondern führt immer eine Validierung durch. Der StringValidator überprüft somit alle Werte, die nicht bereits zuvor durch einen anderen Validator überprüft wurden. Des Weiteren bedeutet dies, dass der StringValidator nur die beiden Ergebnisse „erfolgreiche Validierung" oder „fehlerhafte Validierung" verwendet.

DateBetweenValidator
(de.conceptpeople.checkerberry.common.validation.OperatorContainingDateArrayValidator)

Der DateBetweenValidator gehört als optionaler Validator nicht zu den Standard-Validatoren. Aus diesem Grund muss er vor seiner Verwendung explizit registriert werden. Die Registrierung von Validatoren ist in Abschnitt 2.4.3.5, „Registrierung von Validatoren"beschrieben.

Der DateBetweenValidator wird verwendet, um Datumsangaben ohne Uhrzeit innerhalb eines Intervalls zu prüfen. Zu diesem Zweck werden die zu vergleichenden Werte in *`java.util.Date`*-Objekte konvertiert und als `Date` verglichen.

Der DateBetweenValidator wird verwendet, wenn der erwartete Wert dem Muster `btw nnnn-nn-nn, nnnn-nn-nn` genügt. Der Validator prüft dann, ob der Wert in der Datenbank zwischen den beiden angegebenen Datumsangaben liegt. Die Grenzen des Intervalls können über folgende Operatoren konfiguriert werden:

- `[btw]`: Der erwartete Wert darf auch dem Start- oder Enddatum entsprechen. Dieser Operator ist identisch mit `btw` ohne die Angabe der eckigen Klammern.
- [btw[: Der erwartete Wert darf auch dem Startdatum aber nicht dem Enddatum entsprechen.
-]btw]: Der erwartete Wert darf auch dem Enddatum aber nicht dem Startdatum entsprechen.
-]btw[: Der erwartete Wert darf weder dem Start- noch dem Enddatum entsprechen.

2.4.3.2. Aktivierung / Deaktivierung von Validatoren

Validatoren können innerhalb einer Testmethode über den `ValidatorContext` aktiviert und deaktiviert werden. Der `ValidatorContext` wird über den Aufruf `DbTestHandler.getValidatorContext()` zurückgeliefert. Es ist zu beachten, dass für jede Testmethode ein neuer Kontext verwendet wird, sodass Änderungen an dem Validator-Kontext keine Auswirkungen auf nachfolgende Tests haben.

```
public interface ValidatorContext {
  /**
   * Setzt den Aktivitätsstatus des Validators mit der angegebenen Id.
   *
   * @param validatorId
   *            Id des Validators, dessen Status gesetzt werden soll.
   * @param active
   *           <code>true</code>, wenn der Validator auf aktiv gesetzt werden
   *            soll.<br/>
   *            <code>false</code>, wenn er auf inaktiv gesetzt werden soll.
   */
  void setValidatorActive(ValidatorId validatorId, boolean active);

  /**
   * Setzt den Aktivitätsstatus für alle Validatoren.
   *
   * @param active
   *            <code>true</code>, wenn die Validatoren auf aktiv gesetzt
   *            werden sollen.<br/>
   *            <code>false</code>, wenn sie inaktiv gesetzt werden
   *            sollen.
   */
  void setAllValidatorsActive(boolean active);
}
```

Beispiel 2.9. Aktivierung von Validatoren

2.4.3.3. Anpassen der Ausführungsreihenfolge von Validatoren

Die Ausführungsreihenfolge legt fest, in welcher Reihenfolge die Validatoren bei der Überprüfung von Werten verwendet werden. Da bei der Überprüfung nur Validatoren berücksichtigt werden, die in der Ausführungsreihenfolge angegeben wurden, können Validatoren implizit deaktiviert werden, indem sie in der Ausführungsreihenfolge nicht angegeben werden. Die Ausführungsreihenfolge kann generell über das `DbConfigurationCallback` geändert werden. Diese Änderungen wirken sich auf alle Tests aus. Das folgende Code-Beispiel beschreibt die Methode der `DbConfiguration`.

```
public interface DbConfiguration {
  /**
   * Setzt die Ausführungsreihenfolge der Validatoren. Die
   * Ausführungsreihenfolge kann im
   * {@link de.conceptpeople.checkerberry.db.bridge.context.ValidatorContext}
   * für jede Testmethode geändert werden.
   *
   * @param validatorIdsInExecutionOrder
   *            Validator-Ids in der gewünschten Ausführungsreihenfolge.
   * @see ValidatorContext#setValidatorIdsInExecutionOrder(ValidatorId...)
   */
  void setValidatorIdsInExecutionOrder(
      ValidatorId... validatorIdsInExecutionOrder);
}
```

Beispiel 2.10. Ausführungsreihenfolge von Validatoren (global)

Über den `ValidatorContext` kann die Ausführungsreihenfolge auch für einzelne Testmethoden überschrieben werden. Der `ValidatorContext` wird über den Aufruf `DbTestHandler.getValidatorContext()` zurückgeliefert. Das folgende Code-Beispiel enthält die entsprechende Methode aus dem `ValidatorContext`.

```
public interface ValidatorContext {
  /**
   * Setzt die Ids der Validatoren in der gewünschten Ausführungsreihenfolge
   * z.B. setValidatorIdsInExecutionOrder(executeAsFirstId, executeAsSecondId,
   * executeAsThirdId, ...).
   *
   * @param validatorIdsInExecutionOrder
   *            Ids der Validatoren in der gewünschten Ausführungsreihenfolge.
   */
  void setValidatorIdsInExecutionOrder(
      ValidatorId... validatorIdsInExecutionOrder);
}
```

Beispiel 2.11. Ausführungsreihenfolge von Validatoren (lokal)

Die Ausführungsreihenfolge der Validatoren wird über die Methode `setValidatorIdsInExecutionOrder` festgelegt. Die Methode erwartet eine Liste von Ids registrierter Validatoren. Wenn die Liste die Id eines nicht registrierten Validators enthält, so führt dies zu einer Exception, wenn der Validator bei der Überprüfung eines Werts verwendet werden soll. Checkerberry db geht immer davon aus, dass für jede Validator-Id ein entsprechender Validator registriert ist. Soll ein Validator bei einer Prüfung nicht berücksichtigt werden, kann er jedoch deaktiviert werden. Deaktivierte Validatoren werden bei der Überprüfung von Werten ignoriert.

Die Anpassung der Ausführungsreihenfolge über den `ValidatorContext` wirkt sich auf alle Tabellen und Spalten aus. Es gibt jedoch auch die Möglichkeit, die Ausführungsreihenfolge für einzelne Tabellenspalten zu konfigurieren.

```
public interface DatabaseTableDescription {
  /**
   * Setzt die Ids der Validatoren in Ausführungsreihenfolge für die
   * angegebene Spalte. Bei der Validierung von Werten zu dieser Spalte,
   * werden ausschließlich die in dieser Liste definierten Validatoren in der
   * vorgegebenen Reihenfolge verwendet.
   *
   * @param columnName
   *            Name der Spalte, dessen Validatoren definiert werden sollen.
   * @param validatorIdsInExecutionOrder
   *            Ids der zu verwendenden Validatoren in der gewünschten
   *            Ausführungsreihenfolge.
   */
  void setValidatorIdsInExecutionOrder(String columnName,
      ValidatorId... validatorIdsInExecutionOrder);
}
```

Beispiel 2.12. Ausführungsreihenfolge von Validatoren pro Tabellenspalte

Innerhalb der Tabellenbeschreibungen kann die Ausführungsreihenfolge der Validatoren für einzelne Spalten definiert werden. Dies ist sinnvoll, wenn sehr spezielle Validatoren existieren, die nur für wenige Werte sinnvoll einsetzbar sind. Der Zugiff auf die Tabellenbeschreibung erfolgt über `DbTestHandler.getDatabaseDescription().getTableDescription(String tableName)`.

Bei der Standardbelegung der Ausführungsreihenfolge ist sichergestellt, dass für jeden zu überprüfenden Wert ein Validator vorhanden ist. Durch die Deaktivierung von Validatoren und die Anpassung der Ausführungsreihenfolge kann dieser Sachverhalt nicht mehr sichergestellt werden. Tritt bei der Überprüfung eines Wertes die Situation ein, dass kein zuständiger Validator gefunden wird, wird eine Exception geworfen.

2.4.3.4. Eigene Validatoren erstellen

Checkerberry db unterstützt die Definition von benutzerspezifischen Validatoren. Die Validatoren müssen das Interface `de.conceptpeople.checkerberry.common.validation.Validator` implementieren, das im Folgenden dargestellt ist.

```
public interface Validator {

  /**
   * Liefert die Id des Validators.
   *
   * @return Id des Validators.
   */
  ValidatorId getValidatorId();

  /**
   * Prüft, ob der erwartete Wert von dem Validator geprüft werden soll.
   *
   * @param expectedValue
   *            erwarteter Wert.
   * @return <code>true</code>, wenn der erwartete Wert mit dem impliziten
   *         Pattern des Validators übereinstimmt.<br/>
   *         <code>false</code>, sonst.
   */
  boolean matches(String expectedValue);

  /**
   * Prüft, ob der erwartete Wert mit dem tatsächlichen Wert übereinstimmt.
   * Der erwartete Wert kann Operationen wie z.B. "=" oder "eq" beinhalten.
   * Die konkrete Implementierung des Validators muss diese Operatoren
   * berücksichtigen.
   *
   * @param currentValueAsString
   *            aktueller Wert.
   * @param expectedValueWithOperatorAsString
   *            erwartete Wert, der ggf. Operatoren enthält z.B.
   *            "lt 2010-01-01".
   *
   * @return Status, ob und wenn ja, wie die Validierung durchgeführt wurde.
   */
  ValidationStatus validate(String currentValueAsString,
    String expectedValueWithOperatorAsString);
}
```

Beispiel 2.13. Validator-Interface

Jeder Validator wird über seine Id angesprochen. Die Methode `getValidatorId` liefert diese Id zurück. Durch den Aufruf der Methode `matches` entscheidet checkerberry db, ob der Validator für die Überprüfung des aktuellen Wertes aus den erwarteten Testdaten verwendet werden soll. Liefert die Methode `matches` den Wert `true` zurück, wird danach die Methode `validate` aufgerufen, die die Überprüfung des erwarteten mit dem tatsächlichen Wert durchführt. Die Methode liefert den Status der Validierung. Die möglichen Werte sind:

- `ValidationStatus.VALIDATED_WITH_SUCCESS`, wenn die Validierung erfolgreich war
- `ValidationStatus.VALIDATED_WITH_FAILURE`, wenn die Validierung fehlerhaft war
- `ValidationStatus.VALIDATION_SKIPPED`, wenn die Validierung nicht durchgeführt werden konnte oder sollte.

Damit der Validator verwendet werden kann, muss er, wie im nächsten Abschnitt beschrieben, an checkerberry db registriert werden.

2.4.3.5. Registrierung von Validatoren

Die Registrierung von Validatoren erfolgt über die `DbConfiguration`.

```
public interface DbConfiguration {

  /**
   * Registriert einen neuen Validator. Neue Validatoren werden an den Anfang
   * der Ausführungsreihenfolge gestellt. Dies ist sinnvoll, da checkerberry db
   * intern ebenfalls Validatoren verwendet und externe Validatoren bei der
   * Registrierung vor den internen Validatoren berücksichtigt werden.
   *
   * @param validator
   *            zu registrierender Validator.
   */
  void register(Validator validator);
}
```

Beispiel 2.14. Registrierung von Validatoren

Bei der Registrierung wird der neue Validator an die erste Stelle der Ausführungsreihenfolge gesetzt. Dies ist in der Regel sinnvoll, damit benutzerspezifische Validatoren vor den Standard-Validatoren ausgeführt werden. Die Anpassung der Ausführungsreihenfolge wurde bereits in Abschnitt 2.4.3.3, „Anpassen der Ausführungsreihenfolge von Validatoren“ beschrieben.

Bei der Registrierung eines Validators mit einer bereits registrierten Id, wird der bestehende Validator überschrieben. Die Ausführungsreihenfolge wird durch das Überschreiben eines Validators nicht beeinflusst.

2.4.4. Definition von modularen Testdaten

In den meisten Datenbanken werden neben fachlichen Informationen auch technische Steuerinformationen oder sonstige Stammdaten gehalten. Diese Daten unterliegen nur sehr geringen oder gar keinen Änderungen. Dennoch beeinflussen sie das Verhalten der fachlichen Funktionen. Für die Entwicklung von automatisierten Tests mit checkerberry db bedeutet dies, dass diese Informationen Teil der Testdaten sein müssen.

Checkerberry db verfügt über einen Include-Mechanismus, um Testdaten auszulagern und über eine Referenz in andere Testdatendateien einzubinden. Zu diesem Zweck kann in den Testdaten das Tag *INCLUDE* verwendet werden. Der Name des Tags ist konfigurierbar (siehe Abschnitt 2.4.4.1, „Bezeichner der INCLUDE Tabelle anpassen“). Es hat die folgenden Attribute:

- *FILE:* Der Name der einzubindenden Testdatendatei.
- *LOCATION:* Der Ort wo die einzubindende Testdatendatei erwartet wird. Mögliche Ausprägungen sind:
  - *CLASSPATH:* Sucht die Datei im Klassenpfad
  - *ABSOLUTE:* Die Angabe in *FILE* muss ein absoluter Pfad sein.
  - *RELATIVE:* In diesem Fall muss in *FILE* der relative Pfad zur einzubindenden Testdatendatei angegeben werden. Bei verschachtelt eingebundenen Testdaten wird der Pfad relativ zu der einbindenden Testdatendatei und nicht zu der ursprünglichen Testdatendatei angegeben. *RELATIVE* ist der Default, wenn *LOCATION* nicht angegeben wird.

```
<dataset>
  // Include einer Datei mit relativer Pfadangabe (relativ zu dieser Datei).
  <INCLUDE FILE="../TheSimpsons.xml"/>
  // Absolute Referenz auf eine Include-Datei
  <INCLUDE FILE="o:/groups/SharedProperties.xml" LOCATION="ABSOLUTE"/>
  // Angabe einer Include-Datei, die im Classpath gesucht wird.
  <INCLUDE FILE="de/conceptpeople/Properties.xml" LOCATION="CLASSPATH"/>

  <USERS NAME="Montgomery" SURNAME="Burns"/>
  <USERS NAME="Waylon" SURNAME="Smithers"/>
</dataset>
```

Beispiel 2.15. Einbinden von Testdaten über INCLUDE

Im Code-Beispiel werden die Dateien `TheSimpsons.xml`, `SharedProperties.xml` und `Properties.xml` in die angegebenen Testdaten eingebunden. Natürlich ist es möglich, Testdaten verschachtelt einzubinden. Im obigen Beispiel wäre es somit denkbar, dass in der Datei `TheSimpsons.xml` weitere Dateien wie z.B. `TheSimpsonsKids.xml` und `TheSimpsonsGrownUps.xml` eingebunden werden.

Bevor checkerberry db die Daten in die Datenbank einspielt, werden die Daten aus den Include-Dateien eingelesen und gemäß der DTD sortiert. Danach werden die Daten in der ermittelten Reihenfolge eingespielt. Die folgenden Grafiken verdeutlichen diesen Sachverhalt.

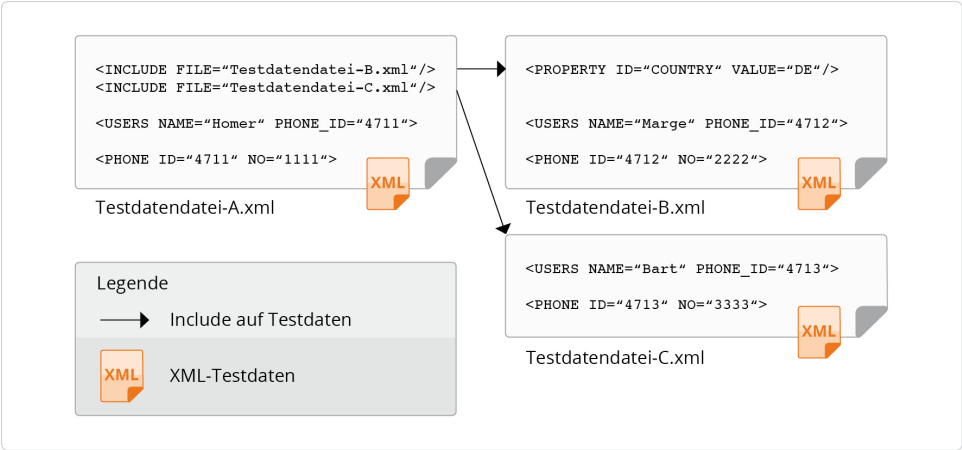


Abbildung 2.9. Include-Beispiel: Testdaten

Die Abbildung zeigt den Datenteil von drei Testdatendateien. Die Datei „Testdatendatei-A.xml" inkludiert die Dateien „Testdatendatei-B.xml" und „Testdatendatei-C.xml". Alle Testdatendateien beinhalten zusätzlich Werte für die Tabellen `USERS` und `PHONE`. Die „Testdatendatei-B.xml" enthält zusätzlich einen Eintrag für die Tabelle `PROPERTY`.

Bei dem Einlesen einer Testdatendatei löst checkerberry db die inkludierten Referenzen auf und fügt die entsprechenden Daten ein. Die folgende Abbildung zeigt, welche Daten die „Testdatendatei-A.xml" nach dem Einlesen enthält.

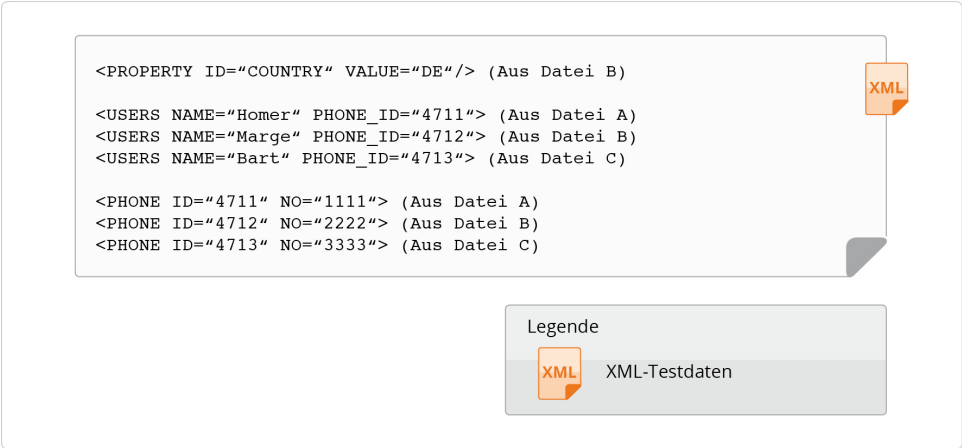


Abbildung 2.10. Include-Beispiel: Ergebnis

Die Datei „Testdatendatei-A.xml" enthält nach dem Einlesen alle Informationen aus den drei Dateien „Testdatendatei-A.xml", „Testdatendatei-B.xml" und „Testdatendatei-C.xml". Die Eingaben wurden dabei gemäß der DTD sortiert, sodass die korrekte Reihenfolge der Tabellen erhalten bleibt.

```
<!-- Reihenfolge der Tabellen -->
<!ELEMENT dataset (
  INCLUDE*,
  EMPTY_TABLE*,
  PROPERTY*,
  USERS*,
  PHONE*,
  …)>
…
```

Beispiel 2.16. Include-Beispiel: Reihenfolge der Tabellen in der DTD

Die Struktur der Testdaten muss für alle Dateienidentisch sein, d.h. alle müssen derselben DTD genügen. In dem Beispiel wird die oben dargestellte DTD von den Testdaten verwendet. Die DTD definiert folgende Reihenfolge der Tabellen: *INCLUDE*, *EMPTY_TABLE*, *PROPERTY*, *USERS* und *PHONE*. Es ist zu beachten, dass die beiden Tabellen *INCLUDE* und *EMPTY_TABLE* intern von checkerberry db verwendet werden.

Das Include-Feature lässt sich sehr flexibel einsetzen. Oftmals gibt es im konkreten Fall fachliche Standard-Konstellationen, die für eine Reihe von Tests verwendet werden sollen. Auch in diesen Situationen kann es sich anbieten, die Daten jeweils in eine eigene Datei auszulagern z.B. *CommonProperties.xml*.

Des Weiteren können verschiedene Konfigurationen in eigenen Dateien abgelegt werden, um diese dann je nach Bedarf einbinden zu können z.B. *PropertiesWebShopX.xml* und *PropertiesWebShopY.xml*.

2.4.4.1. Bezeichner der INCLUDE Tabelle anpassen

Es kann vorkommen, dass checkerberry db in einer Umgebung eingesetzt werden soll, in der bereits eine Datenbanktabelle mit dem Namen „INCLUDE" vorhanden ist. Damit der Name dieser Tabelle nicht mit dem Namen des INCLUDE Tags kollidiert, kann dieser Bezeichner konfiguriert werden. Mit der Methode *setIncludeTableName(String includeTableName)* wird der Default-Wert „INCLUDE" überschrieben.

Beispiel 2.17, „Konfigurationseinstellungen für den Bezeichner der INCLUDE Tabelle"enthält ein Beispiel für diese Konfigurationsmöglichkeit.

```
public class ConfigurationCallback implements DbConfigurationCallback {

  public void configure(DbConfiguration configuration) {
    // Den Bezeichner des Tags von "INCLUDE" auf "INCLUDE_FILE" setzen.
    configuration.setIncludeTableName("INCLUDE_FILE");
  }
}
```

Beispiel 2.17. Konfigurationseinstellungen für den Bezeichner der INCLUDE Tabelle

2.4.5. Tabellenspalten aus Vergleichen ausschließen

Bei der Überprüfung von erwarteten Testdaten stellt checkerberry db eine Zuordnung der zu vergleichenden Zeilen aus den Testdaten und der Datenbank her. Danach werden alle Spalten der Zeilen miteinander verglichen. Wenn es zu Abweichungen kommt, wird eine Fehlermeldung erzeugt und der Test schlägt fehl.

In vielen Fällen enthalten die Tabellen jedoch Spalten, die von dem Vergleich ausgeschlossen werden können. Dies kann für technische Informationen relevant sein, die ggf. dynamisch in die entsprechenden Datenbankzeilen eingefügt werden wie z.B. Änderungszeitstempel oder generierte IDs. Diese Spalten können einfach aus dem Vergleich ausgeschlossen werden, sodass Sie bei dem Vergleich von zwei Zeilen nicht berücksichtigt werden.

```
public void testGetUser() throws Exception {
  …
  // Holen des Testhandlers.
  DbTestHandler testHandler = getEnvironment().getTestHandler();
  // Allgemeine Datenbankbeschreibung holen.
  DatabaseDescription dbDescription = testHandler.getDatabaseDescription();
  // Tabellenbeschreibung für Tabelle USERS holen.
  DatabaseTableDescription tableDescription
      = dbDescription.getTableDescription("USERS");
  // Spalte BIRTHDATE aus dem Vergleich ausschließen.
  tableDescription.addExcludedColumns("BIRTHDATE");
  …
}
```

Beispiel 2.18. Anpassen der auszuschließenden Spalten

Das Code-Beispiel zeigt wie die Spalte `BIRTHDATE` der Tabelle `USERS` vom Vergleich ausgeschlossen wird. Da jeder Test eine eigene Kopie der Datenbankbeschreibung verwendet, bezieht sich diese Änderung nur auf die Methode `testGetUser`. Die Spalten können jedoch auch innerhalb des `DatabaseDescriptionCallback` ausgeschlossen werden. Dies wirkt sich dann auf alle Tests aus.

2.4.6. Ignorieren von Datenbanktabellen

Wenn eine Datenbank Tabellen beinhaltet, die von checkerberry db nicht verändert oder analysiert werden sollen, kann es sinnvoll sein, diese Tabellen global auszuschließen. Dadurch ist es nicht mehr möglich, aus Versehen Daten aus diesen Tabellen zu löschen oder zu überschreiben. Die ausgeschlossenen Tabellen werden von checkerberry db vollständig ignoriert. Dies gilt insbesondere für die Reports, die Datenbank-Dumps und den Vergleich mit erwarteten Testdaten.

```
public class ConfigurationCallback implements DbConfigurationCallback {

  @Override
  public void configure(DbConfiguration configuration) {
    // schließe Tabellen PIZZA und TOPPING aus
    configuration.addTablesToIgnoreList("PIZZA, "TOPPING");
  }
}
```

Beispiel 2.19. Ausschluss von Tabellen, die durch checkerberry db ignoriert werden sollen

Das obige Beispiel zeigt, wie Tabellen global ausgeschlossen werden können. Es ist auch möglich die Wildcards * (für eine beliebige Anzahl Zeichen) und ? (für genau ein Zeichen) zu verwenden.

2.4.7. Verwendung von Parametern in Testdaten

Bei der Definition der erwarteten Testdaten gibt es häufig das Problem, dass einige Werte zum Zeitpunkt der Testerstellung nicht feststehen. Dies umfasst zum Beispiel generierte IDs oder auch technische Zeitstempel, die in die Datenbank eingefügt werden. Es gibt zwei Möglichkeiten, mit dieser Situation umzugehen.

Eine Lösung besteht wie in Abschnitt 2.4.5, „Tabellenspalten aus Vergleichen ausschließen“ beschrieben darin, die Spalten aus dem Vergleich auszuschließen. Dieser Weg ist dann sinnvoll, wenn die entsprechenden Daten nicht im Fokus des Tests stehen.

Wenn die Prüfung von dynamischen Werten für den Test relevant ist, ist der Ausschluss der Spalte keine sinnvolle Maßnahme. Für diese Situationen stellt checkerberry db die Möglichkeit zur Verfügung, Parameter zu definieren.

```
<dataset>
  <!-- Die Spalte ID enthält den Parameter userId. -->
  <USERS ID="${userId}" NAME="Montgomery" SURNAME="Burns"/>
</dataset>
```

Beispiel 2.20. Parameter in Testdaten

Das obige Beispiel zeigt, wie Parameter in den erwarteten Testdaten definiert werden können. In diesem Fall wird der Parameter *userId* als ID für die entsprechende Zeile der Tabelle *USERS* definiert. Die Definition von Parametern erfolgt durch die Zeichen *${*, gefolgt von dem Parameternamen und einer abschließenden geschweiften Klammer *}*.

```
public void testSaveUser() throws Exception {
  …
  // Speichern eines neuen User-Objekts.
  userDao.save(user);
  …
  // Holen des Testhandlers.
  DbTestHandler testHandler = getEnvironment().getTestHandler();
  // Holen des Parameter-Kontextes.
  ParameterContext parameterContext = testHandler.getParameterContext();
  // Wenn das User-Objekt in der Datenbank gespeichert wird, wird in
  // der Datenbank eine neue ID vergeben, die dann in dem User-Objekt
  // gesetzt wird. Dieser Wert wird dem Parameter userId zugewiesen.
  parameterContext.addParameter("userId", user.getId());
  …
}
```

Beispiel 2.21. Setzen von Parametern

In den `ParameterContext` können beliebig viele Parameter eingefügt werden. Das obige Beispiel zeigt eine typische Situation im Hibernate-Umfeld: Ein *User*-Objekt wird in der Datenbank gespeichert. Erst nach der Speicherung ist die ID des Objekts verfügbar und kann erst dann für den Vergleich genutzt werden. An dieser Stelle ist die Entscheidung des Software-Entwicklers gefragt: Muss die Spalte *ID* beim Vergleich berücksichtigt werden oder kann ein alternativer Lookup-Key gewählt werden? Die Antwort auf die Frage bestimmt, ob mit Parametern oder auszuschließenden Spalten gearbeitet werden sollte.

Die erwarteten Testdaten werden innerhalb der Methode *assertEqualsExpected* eingelesen. In den Testdaten wird dabei nach enthaltenen Parametern gesucht, die dann durch die entsprechenden Werte aus dem Parameter-Kontext ersetzt werden. Die Parameter müssen aus diesem Grund in der Testmethode vor dem Aufruf der Methode *assertEqualsExpected* hinzugefügt werden.

Parameter stellen eine einfache Möglichkeit zur Dynamisierung der Testdaten dar. Da bei einer zu intensiven Nutzung jedoch Übersichtlichkeit und damit Wartbarkeit der Tests leiden, bietet checkerberry db mit den Funktionen noch einen mächtigeren Ansatz, dies zu bewerkstelligen. Nähere Informationen finden sich in Abschnitt 2.4.8, „Überprüfen von Datenbankrelationen mit Autoparametern“ und Abschnitt 2.4.9, „Dynamische Testdaten durch Funktionen“.

2.4.8. Überprüfen von Datenbankrelationen mit Autoparametern

In Abschnitt 2.4.7, „Verwendung von Parametern in Testdaten“ wurde beschrieben, wie dynamische Werte mit Hilfe von Parametern beim Test berücksichtigt werden können. Spannend wird es allerdings, wenn Tabellen mit 1:n- oder n:m-Relationen überprüft werden sollen. Die Verwendung von Parametern kann in diesem Fall zu aufwändig werden, da für jede dynamische ID ein eigener Parameter eingefügt und im Test ermittelt werden muss. Mit dem Konzept der Autoparameter stellt checkerberry db eine einfachere Möglichkeit für diese Situationen zur Verfügung.

Die Definition von Autoparametern erfolgt in den Testdaten durch die Zeichen *#{*, gefolgt von dem Parameternamen und einer abschließenden geschweiften Klammer *}*.

```
<dataset>
  // Für die Pizza-ID wird der Autoparameter pizzaId verwendet.
  <PIZZA ID="#{pizzaId}" NAME="Salamipizza"/>
  // Für die Topping-ID wird der Autoparameter toppingId verwendet.
  <TOPPING ID="#{toppingId}" NAME="Salami"/>
  // In der Zuordnungstabelle werden die gleichen Autoparameter
  // verwendet.
  <PIZZA_TOPPING PIZZA_ID="#{pizzaId}" TOPPINGS_ID="#{toppingId}"/>
</dataset>
```

Beispiel 2.22. Beispiel Autoparameter: Erwartete Testdaten

Das obige Code-Beispiel enthält erwartete Testdaten, in denen zwei Autoparameter angegeben sind. Nach der Durchführung eines Tests wird erwartet, dass eine neue Salamipizza in der Datenbank angelegt wurde. Der Pizzabelag wird der jeweiligen Pizza über eine n:m-Beziehung zugewiesen. Zu diesem Zweck werden die beiden Autoparameter *pizzaId* und *toppingId* definiert.

Die folgende Abbildung beschreibt wie die Werte der Autoparameter ermittelt werden.

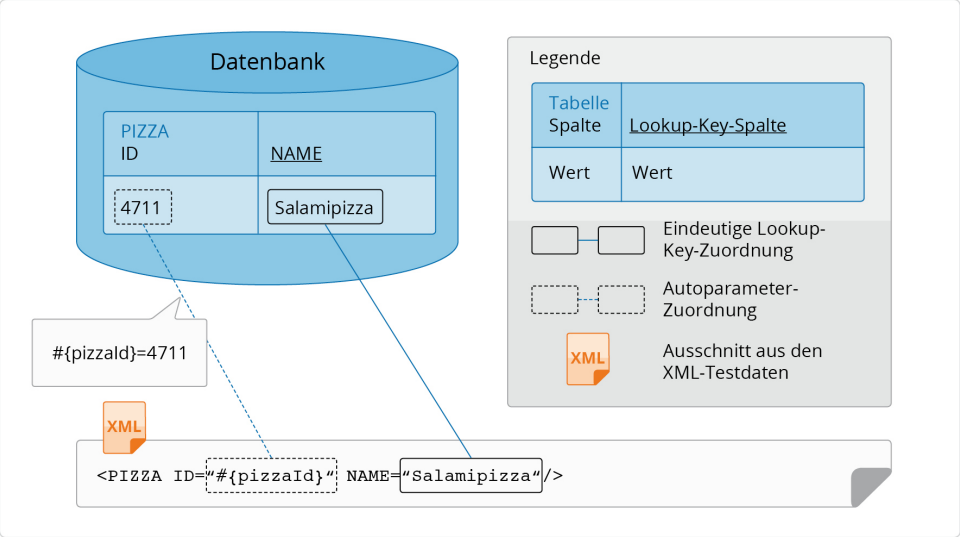


Abbildung 2.11. Beispiel Autoparameter: Ermittlung der pizzaId

Die Abbildung enthält den relevanten Teil der Datenbank und den relevanten Teil der erwarteten Testdaten. Die Spalte *NAME* ist als Lookup-Key der Tabelle *PIZZA* definiert. Über den Wert dieser Spalte ermittelt checkerberry db die Zuordnung zwischen Datenbank und Testdaten. Da der Wert für den Autoparameter *pizzaId* in checkerberry db noch unbekannt ist, wird er auf den Wert der Spalte *ID* der zugeordneten Datenbankzeile gesetzt. Der neue Wert des Autoparameters ist somit „4711“.

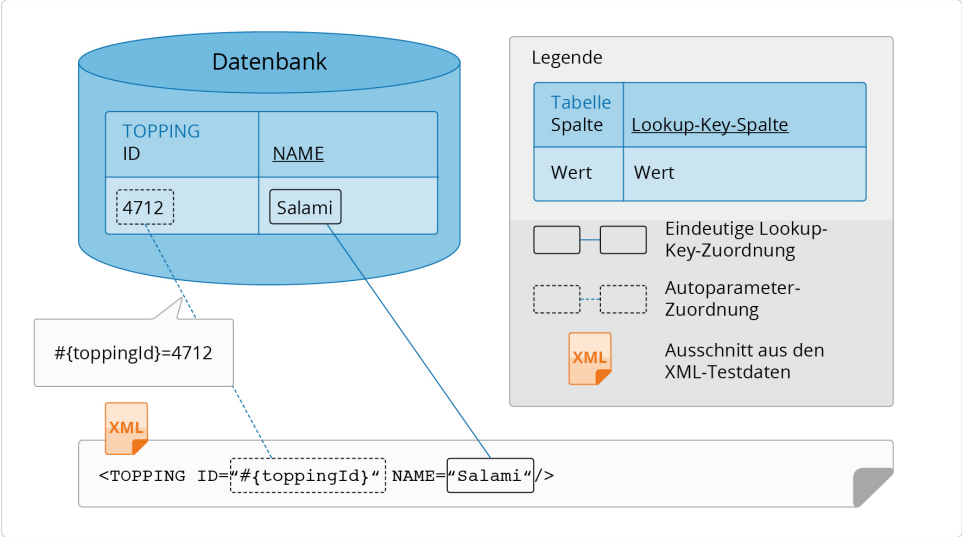


Abbildung 2.12. Beispiel Autoparameter: Ermittlung der toppingId

Wie in Abbildung 2.12, „Beispiel Autoparameter: Ermittlung der toppingId" dargestellt ist, erfolgt die Ermittlung des Autoparameters `toppingId` auf die gleiche Art und Weise wie für den Autoparameter `pizzaId`. Der neue Wert des Autoparameters ist somit „4712".

Beim Einlesen der erwarteten Testdaten für die Tabelle `PIZZA_TOPPING` wird festgestellt, dass für die enthaltenen Autoparameter `pizzaId` und `toppingId` bereits Werte ermittelt wurden. Diese Werte werden in den erwarteten Testdaten eingetragen.

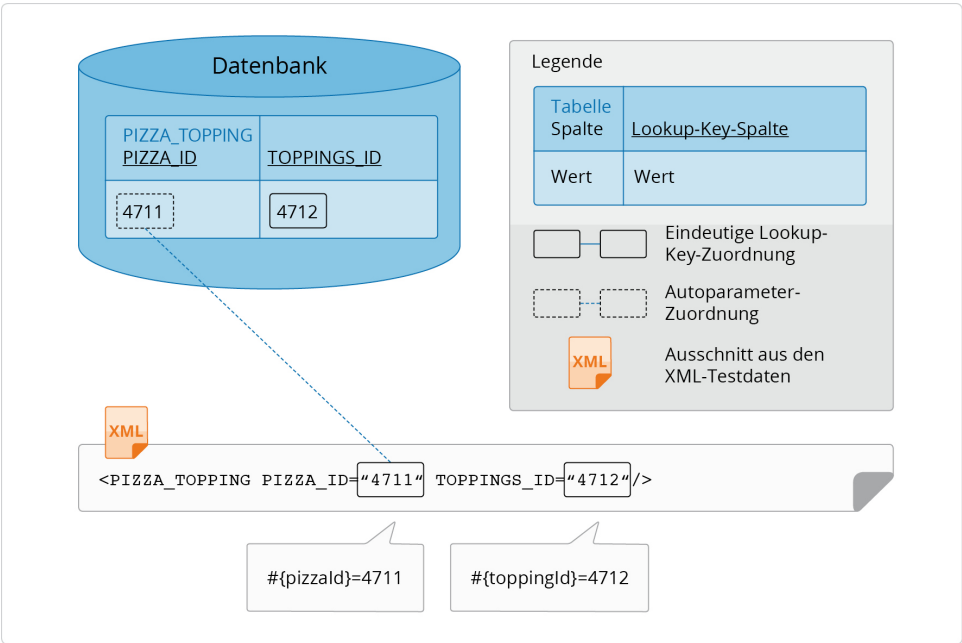


Abbildung 2.13. Beispiel-Autoparameter: Setzen von Werten

Wie in der obigen Abbildung dargestellt, wird der Wert für den Autoparameter `pizzaId` auf „4711" und der Wert für den Autoparameter `toppingId` auf „4712" gesetzt. Da die zugehörigen Spalten `PIZZA_ID` und `TOPPINGS_ID` Teil des Lookup-Keys sind, kann der zugehörige Datensatz in der Datenbank erst nach dem Setzen der Autoparameter ermittelt werden. In dem obigen Beispiel wird ein entsprechender Datensatz in der Datenbank gefunden. Anderenfalls würde der Test mit einer Fehlermeldung abbrechen.

```
public void testSaveUser() throws Exception {
  …
  // Testhandler holen.
  DbTestHandler testHandler = getEnvironment().getTestHandler();
  // Datenbank gegen erwartete Testdaten prüfen. Hier werden implizit die
  // Autoparameter ermittelt.
  testHandler.assertEqualsExpected();

  // Holen des Autoparameter-Kontexts.
  AutoParameterContext context = testHandler.getAutoParameterContext();
  // Auslesen der pizzaId. Die hat in dem Beispiel den Wert 4711.
  String id = context.getAutoParameterValue("pizzaId");
  …
}
```

Beispiel 2.23. Beispiel Autoparameter: Auslesen von Autoparametern im Test

Das obige Code-Beispiel zeigt, wie ein Autoparameter in der Testmethode gelesen werden kann. Die Autoparameter sind nach der Ausführung der Methode *assertEqualsExpected* in dem AutoParameterContext enthalten und können über die Methode *getAutoParameterValue* ausgelesen werden.

Checkerberry db stellt sicher, dass alle Vorkommen eines Autoparameters in einer Testdatendatei den gleichen Wert haben. Wenn der Wert eines Autoparameters noch nicht ermittelt wurde, wird der Wert, wie in Abbildung 2.12, „Beispiel Autoparameter: Ermittlung der toppingId"dargestellt, ermittelt. Für alle weiteren Vorkommen dieses Autoparameters wird der ermittelte Wert in die Testdaten eingetragen. Wenn es zu einer Abweichung zwischen der Datenbank und den erwarteten Testdaten kommt, wird eine Fehlermeldung erzeugt und der Test schlägt fehl.

2.4.8.1. Auflösen von Autoparametern in Lookup-Keys

Bei Autoparametern handelt es sich um automatisch belegte Parameter, die bei der Überprüfung der Datenbankinhalte ermittelt werden. Um den Wert eines Autoparameters ermitteln zu können, muss die entsprechende Zeile in den erwarteten Testdaten zu dem zugehörigen Datensatz in der Datenbank zugeordnet werden. Diese Zuordnung erfolgt über die Lookup-Keys. Was passiert also, wenn Autoparameter in Lookup-Key-Spalten definiert sind?

In dem vorherigen Beispiel wurden die beiden Autoparameter *pizzaId* und *toppingId* in den Lookup-Key-Spalten der Tabelle *PIZZA_TOPPING* definiert. Dennoch konnten die Werte der Autoparameter ermittelt werden, da beide Autoparameter auch in einer anderen Tabelle definiert waren, in denen sie nicht in Lookup-Key-Spalten vorhanden waren. Dies ist eine notwendige Voraussetzung: Jeder Autoparameter muss mindestens in einer Spalte definiert sein, die nicht Teil eines Lookup-Keys ist. Die Reihenfolge der Zeilen in den erwarteten Testdaten ist dabei nicht relevant. Es ist somit möglich, dass das erste Vorkommen eines Autoparameters in einer Lookup-Key-Spalte steht. Die Überprüfung dieser Zeile wird in diesem Fall verschoben, bis der Wert des Autoparameters ermittelt werden konnte.

Das folgende Beispiel zeigt den Fall, dass ein Autoparameter nicht ermittelt werden konnte, da er nur in einer Lookup-Key-Spalte definiert ist.

```
<dataset>
  // Der Autoparameter userName ist in der Lookup-Key-Spalte NAME
  // definiert.
  <USERS NAME="#{userName}" SURNAME="Simpson"/>
</dataset>
```

Beispiel 2.24. Beispiel Autoparameter: Definition in Lookup-Key-Spalte

In den erwarteten Testdaten ist genau ein Eintrag vorhanden. In der Spalte *NAME* der Tabelle *USERS* ist der Autoparameter *userName* eingetragen, wobei es sich bei dieser Spalte um den Lookup-Key handelt.

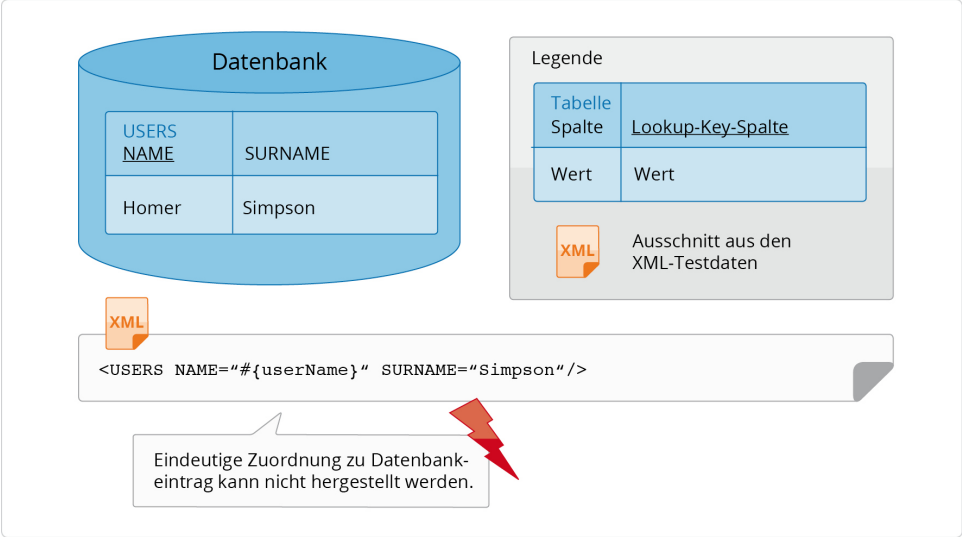


Abbildung 2.14. Beispiel Autoparameter: Fehler bei der Ermittlung

Die obige Abbildung zeigt das Problem. Beim Einlesen der erwarteten Testdaten versucht checkerberry db den Wert des Autoparameters zu ermitteln. Da der Lookup-Key in den erwarteten Testdaten jedoch nicht vorhanden ist, kann keine Zuordnung zu den Daten in der Datenbank hergestellt werden. In dieser Situation erzeugt checkerberry db eine Fehlermeldung und der Test schlägt fehl.

2.4.8.2. Manuelle Prüfung einzelner Autoparameter

Eine besondere Verwendung der Autoparameter ergibt sich, wenn nur ein Vorkommen eines Autoparameters in den Testdaten verwendet wird. Das folgende Code-Beispiel enthält eine derartige Situation.

```
<dataset>
  // Der Autoparameter userId ist nur einmal definiert. Die Spalte NAME
  // ist die Lookup-Key-Spalte.
  <USERS ID="#{userId}" NAME="Homer"/>
</dataset>
```

Beispiel 2.25. Beispiel Autoparameter: Einmalige Definition

Die erwarteten Testdaten enthalten einen Eintrag für die Tabelle *USERS*. Die Spalte *NAME* ist die Lookup-Key-Spalte und die Spalte *ID* enthält den Autoparameter *userId*.

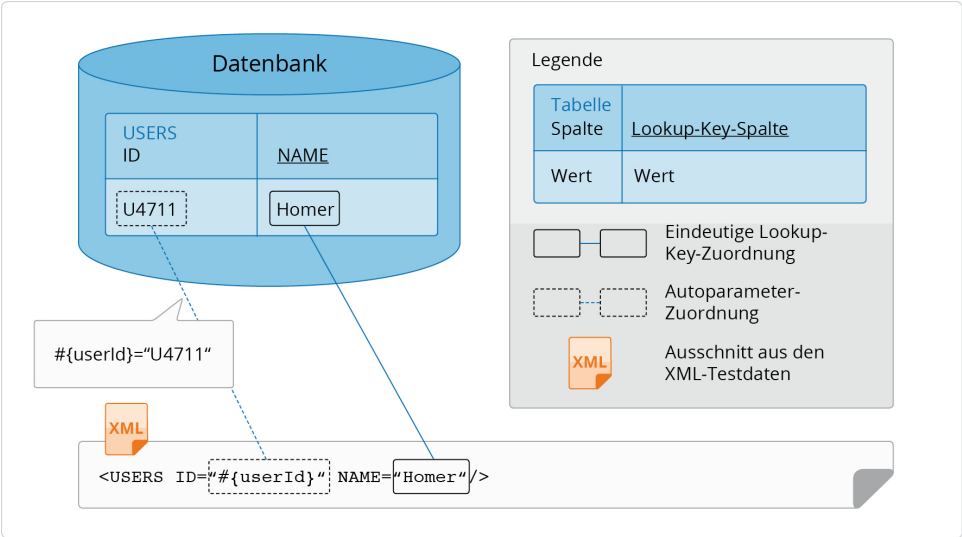


Abbildung 2.15. Beispiel Autoparameter: Spezialfall einfache Definition

Abbildung 2.15, „Beispiel Autoparameter: Spezialfall einfache Definition“stellt die Ermittlung des Werts für den Autoparameter *userId* dar. Über den Lookup-Key *NAME* mit dem Wert „Homer“ wird die Zuordnung zwischen den erwarteten Testdaten und der Datenbank hergestellt. Der Wert von *userId* wird auf „U4711“ gesetzt. Da nur ein Vorkommen für den Autoparameter *userId* existiert, wird er lediglich auf den Wert aus der Datenbank gesetzt, ohne dass die Korrektheit des Wertes überprüft wird. Dieses Verfahren lässt sich somit verwenden, um die Überprüfung der erwarteten Testdaten gezielt für einzelne Werte zu umgehen. Alternativ kann der Wert des Autoparameters in der Testmethode manuell geprüft werden, wie das folgende Code-Beispiel zeigt.

```
public void testSaveUser() throws Exception {
  …
  // Testhandler holen.
  DbTestHandler testHandler = getEnvironment().getTestHandler();
  // Datenbank gegen erwartete Testdaten prüfen. Hier werden implizit die
  // Autoparameter ermittelt.
  testHandler.assertEqualsExpected();

  // Holen des Autoparameter-Kontexts.
  AutoParameterContext context = testHandler.getAutoParameterContext();
  // Auslesen der userId.
  String id = context.getAutoParameterValue("userId");
  // Manuelle Prüfung auf „U“ als Startzeichen.
  assertTrue(id.startsWith("U"));
}
  …
```

Beispiel 2.26. Beispiel Autoparameter: Manuelle Prüfung

In dem Code-Beispiel wird der Wert des Autoparameters wie bereits zuvor beschrieben aus dem `AutoParameterContext` gelesen. Danach wird geprüft, ob der Wert mit einem „U“ beginnt. Dieses Beispiel zeigt, dass man durch das vorgestellte Vorgehen einzelne Werte aus der allgemeinen Überprüfung der Testdaten herausnehmen kann, um beispielsweise speziellere Prüfungen direkt in der Testmethode vorzunehmen. Generell lassen sich spezifische Prüfungen jedoch sinnvoller über eigene Validatoren abbilden, die in Abschnitt 2.4.3, „Überprüfen der Ergebnisse durch Validatoren“beschrieben sind.

2.4.9. Dynamische Testdaten durch Funktionen

Innerhalb von checkerberry db ist es möglich, dynamische Informationen über Parameter und Autoparameter in den erwarteten Testdaten einzufügen. In der Praxis stößt man bisweilen jedoch auf Situationen, die auch ein dynamisches Anpassen der initialen Testdaten erfordern. Beispielsweise mag es in einem komplexen System nicht immer ohne größeren Aufwand möglich sein, die aktuelle Uhrzeit von außen zu beeinflussen. Stellt ein solches System dann zeitabhängige Abfragen an seine Datenbank, wie z.B. „Welche User haben sich innerhalb der letzten Woche eingeloggt“, so erfordert das Testen dieser Abfragen die Verwendung von dynamisch zu errechnenden Zeitangaben. Aus diesem Grund bietet checkerberry db neben den Parametern noch einen generelleren Ansatz zur Dynamisierung von Testdaten: in Testdaten eingebettete Funktionen.

```
<dataset>
  // Homer hat sich vor 2 Tagen das letzte Mal eingeloggt.
  <USERS ID="1" NAME="Homer" LAST_LOGIN="->now(-2 days,+1 hour)"/>
</dataset>
```

Beispiel 2.27. Dynamische initiale Testdaten mit Hilfe von Funktionen

Der Ausdruck *->now(-2 days,+1 hour)* wird hierbei als Aufruf der Funktion *now* mit den Argumenten *-2 days* und *+1 hour* interpretiert und bei Einspielen der initialen Testdaten abhängig vom aktuellen Zeitpunkt beispielsweise zu dem Ergebnis *2011-02-07 16:50:53.047* ausgewertet, wenn der aktuelle Zeitpunkt *2011-02-09 15:50:53.047* ist.

2.4.9.1. Zusammenhang von Parametern und Funktionen

Parameter werden in dem `ParameterContext` abgelegt und behalten ihre Gültigkeit für die Dauer der Ausführung einer Testmethode. Dieser Mechanismus dient der Unabhängigkeit der Testmethoden, indem er verhindert, dass Parameter zum Übergeben von Werten zwischen den Methoden verwendet werden.

Im Gegensatz dazu sind Funktionen in der Konfiguration definiert und können daher in allen Tests und deren Methoden verwendet werden. Sie haben jedoch Zugriff auf die jeweils zur aktuellen Testmethode gehörenden Kontexte (Parameter, Autoparameter, Validatoren) und können daher sowohl Parameter manipulieren als auch ihr Verhalten abhängig von Parameterwerten ändern. Als Beispiel sei hier die vordefinierte Funktion *param* genannt, die den Wert eines Parameters zurückgibt. Im Grunde genommen ist die Parameter-Schreibweise *${name}* lediglich eine Kurzform für *->param(name)*.

Funktionen und Parameter können sowohl in den initialen als auch in den erwarteten Testdaten verwendet werden. Sie lassen sich beliebig schachteln und mit normalen Feldinhalten mischen:

```
<dataset>
  // Ein Mitglied der Familie Simpson und seine Lieblingspizza
  // Für name=Homer wird der Name „Homer Simpson“ in dem Parameter
  // „Homer.fullname“ gespeichert. In der darauf folgenden Zeile wird
  // der Parameter „Homer.fullname“ wieder ausgelesen.
  <USERS ID="1" NAME="->save(${name} Simpson,${name}.fullname)" />
  <PIZZA ID="1" NAME="${->param(name).fullname}’s favourite Pizza" />
</dataset>
```

Beispiel 2.28. Schachteln von Funktionsaufrufen und Parametern

Auf die Funktion „*save*“ wird weiter unten noch detailliert eingegangen. Hier sei nur erläutert, dass die Ergebnisse von Funktionsaufrufen anstelle von diesen in den sie umgebenden Text eingefügt werden und sich auf diese Art auch Funktionsargumente oder Parameternamen dynamisch generieren lassen.

Die einzige Einschränkung bei der Schachtelung bezieht sich auf den Funktionsnamen: Er darf weder Funktionsaufrufe noch Parameter enthalten. *->${function}()* würde daher versuchen eine Funktion namens „*${function}*“ aufzurufen anstatt den Parameter aufzulösen und das Ergebnis als den Namen einer Funktion zu interpretieren.

2.4.9.2. Maskieren von Funktionsaufrufen

Manchmal ist es nötig, das Auswerten von Funktionsaufrufen zu unterdrücken. Beispielsweise möchte man den Text „*->Begrüssung(Frau,Martens)*“ in ein Feld der Datenbank schreiben ohne dabei durch die Fehlermeldung `CB-DB-1026` darauf hingewiesen zu werden, dass die Funktion „*Begrüssung*“ nicht definiert wurde. Hierzu bietet checkerberry db die Möglichkeit der Maskierung von Funktionsaufrufen oder Parametern mit Hilfe des Maskierungszeichens „^“. In unserem Beispiel würde also „*^->Begrüssung(Frau,Martens)*“ das gewünschte Ergebnis liefern. Bei der Maskierung gibt es jedoch folgendes zu beachten:

1. In einfachen Fällen wie dem obigen Beispiel reicht es aus, den Anfang des Parameters/Funktionsaufrufes zu maskieren. Möchte man jedoch maskierte und nicht-maskierte Parameter und/oder Funktionsaufrufe mischen (Beispielsweise um das Zeichen „)“ als Parameter an eine Funktion zu übergeben), so ist es häufig auch nötig die Enden zu maskieren: „*->param(some_(weird)_param)*“ würde beispielsweise die Funktion param mit dem Parameter „*some_(weird*“ aufrufen. Ist checkerberry db nicht in der Lage einen Ausdruck auszuwerten, so erzeugt sie die Fehlermeldung *CB-DB-1025*, um auf diesen Umstand hinzuweisen.
2. Maskierungen innerhalb eines Funktionsnamens maskieren nicht den Funktionsaufruf, sondern nur das Zeichen. Dies ermöglicht Funktionsnamen mit beliebigen Sonderzeichen.

2.4.9.3. Vordefinierte Funktionen

Um die Dynamisierung der Testdaten möglichst einfach zu gestalten, bringt checkerberry db von Haus aus eine Reihe von Funktionen mit. Diese werden hier kurz vorgestellt und erklärt:

->cat(param1,param2,...)
Konkateniert die übergebenen Parameter. Wird intern verwendet um Ergebnisse von Funktionsaufrufen in andere Inhalte einzufügen. „`->cat(a,${name},b)`" ist also äquivalent zu „`a${name}b`". Ist eines der Argumente `null`, so wird es als Leerstring interpretiert. „`a->null()b`" wird daher zu „`ab`" ausgewertet.

->param(name)
Ermittelt aus dem `ParameterContext` den Wert des Parameters `name` und gibt diesen zurück.

->save(value,name1, ..., nameN)
Speichert den Wert `value` in den Parametern `name1` bis `nameN` und gibt ihn zurück.

->null(...)
Ignoriert sämtliche übergebene Parameter und gibt den Wert `null` zurück.

->set(value,name1, ..., nameN)
Speichert den Wert `value` in den Parametern `name1` bis `nameN` und gibt null zurück. Kurzform für `->null(->save(value,name1, …, nameN))`.

->unset(name)
Löscht den Parameter `name` aus dem `ParameterContext`.

->now(param1,param2,...)
Ermittelt den derzeitigen Zeitpunkt, modifiziert ihn entsprechend der Angaben der einzelnen Argumente und liefert das Ergebnis als JDBC Timestamp formatiert zurück. Beispiele:

Tabelle 2.1. now()-Funktion

Ausdruck	Ergebnis
`->now()`	2010-11-25 15:53:29.915
`->now(-2 years)`	2008-11-25 15:53:29.915
`->now(-2 years,+3 days)`	2008-11-28 15:53:29.915
`->now(+21 seconds)`	2010-11-25 15:53:50.915

Als Zeitangaben können folgende Bezeichnungen verwendet werden: `year(s)`, `month(s)`, `week(s)`, `day(s)`, `hour(s)`, `minute(s)`, `second(s)` und `millisecond(s)`.

Anmerkung: Die 4 Ausdrücke müssen nicht innerhalb derselben Millisekunde ausgewertet werden um dieses Ergebnis zu erzielen. Die Funktion `now` verwendet den Parameter `now` um den derzeitigen Zeitpunkt zwischenzuspeichern. Beim ersten Aufruf der `now`-Funktion wird (da der Parameter noch nicht gesetzt wurde) der aktuelle Zeitpunkt bestimmt und in den Parameter geschrieben. Alle weiteren Aufrufe verwenden einfach den gespeicherten Wert. Auf diese Weise sind die Testdaten unabhängig von der Auswertungsgeschwindigkeit der Testdaten und die Auswertung zum Einspielen der initialen Testdaten am Anfang liefert das gleiche Ergebnis wie die zum Vergleichen am Ende des Testes. Auf diese Weise ist es ebenfalls möglich durch manuelles Setzen des Parameters die Version der Testdaten zu dem jeweiligen Zeitpunkt zu erhalten.

Darüberhinaus verfügt die Funktion `now` (`de.conceptpeople.checkerberry.db.core.functions.NowFunction`) über einen Setter, mit dem man einen eigenen `TimeService` definieren kann. Per Default liefert der `TimeService` immer die aktuelle Systemzeit zurück. Es kann jedoch auch sinnvoll sein, eine andere Zeit zu verwenden z.B. Transaktionszeitpunkte oder die aktuelle Zeit in der Datenbank. Durch die Implementierung eines eigenen `TimeService` kann man somit kontrollieren, welche Zeit in der Funktion `now` und `today` verwendet wird. Das folgende Code-Beispiel enthält das Interface des `TimeService`.

```
package de.conceptpeople.checkerberry.db.core.functions.callback;

import java.util.Calendar;

/**
 * Ein TimeService wird beispielsweise von der {@link NowFunction}
 * oder der {@link TodayFunction} benutzt um den aktuellen Zeitpunkt
 * zu bestimmen. Über die Methode setTimeService() ist es möglich,
 * diese Funktionen zum Verwenden einer eigenen Implementation zu
 * bewegen.
 */
public interface DbTimeService {

    /**
     * Liefert den aktuellen Zeitpunkt zurück.
     *
     * @return den Zeitpunkt.
     */
    Calendar getCurrentTime();

}
```

Beispiel 2.29. Interface des TimeService

->today(param1,param2,...)
Ermittelt das derzeitige Datum, modifiziert es entsprechend der Angaben der einzelnen Argumente und liefert das Ergebnis als JDBC Date formatiert zurück. Beispiele:

Tabelle 2.2. today()-Funktion

Ausdruck	Ergebnis
->today()	2010-11-25
->today(-2 years)	2008-11-25
->today(-2 years,+3 days)	2008-11-28

Die Anmerkung zur *now*-Funktion trifft auf die *today*-Funktion ebenfalls zu. Auch sie verwendet den Parameter *now*, um den aktuellen Zeitpunkt zu bestimmen bzw. zu speichern.

2.4.9.4. Eigene Funktionen erstellen

Selbstverständlich ist es möglich, checkerberry db über die Konfiguration um eigene Funktionen zu erweitern. Die Konfiguration erlaubt nicht nur das Registrieren von eigenen Funktionen, sondern auch das Anpassen der Syntax (siehe unten). Die Definition einer Funktion zum Addieren ihrer Parameter zeigt das Beispiel 2.30, „Definieren einer Funktion zur Addition“.

```
/**
 * Addiert alle Argumente.
 */
public class AddFunction implements TestdataFunction {

  @Override
  public String evaluate(ExecutionContext context,
                         List<String> arguments,
                         String functionName) {
    int result = 0;
    for (String argument : arguments) {
      result += Integer.valueOf(argument);
    }
    return Integer.toString(result);
  }
}
```

Beispiel 2.30. Definieren einer Funktion zur Addition

Funktionen implementieren das Interface *TestdataFunction*. Die Methode *evaluate()* wird bei Auswertung der Funktion aufgerufen und bestimmt das Ergebnis. Der aktuelle Ausführungskontext (u.a. Kontexte für Parameter, Autoparameter und Validatoren), eine Liste der Argumente in Form von Strings und der Name, unter dem die Funktion aufgerufen wurde (theoretisch ist es möglich eine Funktion unter mehreren Namen zu registrieren), werden übergeben. Es liegt in der Verantwortung der Implementation zu überprüfen, ob die Parameter in korrekter Anzahl und Form vorliegen. Argumente können null sein und es ist für eine Funktion legitim null zurückzugeben (Beispiel: *->null()*). Ist ein Parameter ausschließlich durch eine Funktion definiert, so führt ein Rückgabewert von *null* zum Belegen des Feldes mit *null*, was beispielsweise gegen evtl. in der Datenbank vorhandene *NOT NULL*-Bedingungen verstoßen würde.

Anschließend ist das Registrieren der Funktion in der Konfiguration folgendermaßen möglich:

```
configuration.registerFunction("add",new AddFunction());
```

Beispiel 2.31. Registrieren der AddFunction

Nach der Registrierung kann die neue Funktion in den Testdaten verwendet werden (*->add(1,3)* würde zu *4* ausgewertet).

2.4.9.5. Definition einer eigenen Funktionssyntax

Weiter oben wurde bereits über die Notwendigkeit des Maskierens einzelner Funktionsaufrufe oder Parameter gesprochen. Die Standard-Syntax von checkerberry db ist auf eine problemlose Verwendbarkeit innerhalb von Java und XML hin ausgelegt. Es ist jedoch auch möglich, die Syntax mit anderen Schlüsselwerten zu belegen.

```
->function(arg1,arg2) ^$( ${parameter}

-> functionCallPrefix
(  functionCallOpeningBracket
,  argumentSeparator
)  functionCallClosingBracket
^  escapeCharacter
${ parameterPrefix
}  parameterSuffix
```

Beispiel 2.32. Elemente der Syntax

Beispiel 2.32, „Elemente der Syntax“ listet alle syntaktischen Elemente von checkerberry db auf. Die Konfiguration stellt Methoden für die Anpassung der einzelnen Elemente zur Verfügung.

```
configuration.setEscapeCharacter("?");
configuration.setFunctionCallSyntax("=", "[-", "-]");
configuration.setArgumentSeparator(":");
configuration.setParameterSyntax("(", ")");
```

Beispiel 2.33. Rekonfiguration der Syntax

Die obigen Anweisungen passen die Syntax derart an, dass „`=save[-Homer:name-]`“ künftig als Funktionsaufruf mit 2 Parametern und „`(name)`“ künftig als Parameter interpretiert wird – es sei denn sie wurden durch ein „`?`“ maskiert.

Auch wenn sich die Syntax auf diese Weise sehr flexibel konfigurieren lässt, so gibt es doch gewisse Anforderungen an sie. Sie sollte beispielsweise eindeutig sein und sich effizient parsen lassen. Um das Konfigurieren dennoch benutzerfreundlich zu gestalten, überprüft checkerberry db die neue Syntax direkt vor der ersten Benutzung, weißt gegebenenfalls durch die Fehlermeldung `CB-DB-1027` auf einen Missstand hin und gibt Informationen, welche 2 syntaktischen Elemente sich nicht miteinander vertragen. Wenn es keinen expliziten Grund gibt, empfehlen wir die Syntax unverändert zu lassen.

2.4.9.6. Auswertungsreihenfolge von Funktionen in Testdaten

Insbesondere beim Arbeiten mit Parametern ist die Reihenfolge, in der die Ausdrücke ausgewertet werden, von entscheidender Bedeutung. Die Regeln der Auswertungsreihenfolge sind im Folgenden kurz zusammengefasst.

- **Innerhalb eines Feldes** werden verschachtelte Funktionsaufrufe und Parameter von innen nach außen und solche auf gleicher Ebene von links nach rechts ausgewertet. Der Ausdruck „`->save(Helga,name) ${name}son`“ wird daher zu „Helga Helgason“ ausgewertet, während „`${name} ->save(Helga,name)son`“ zu der Fehlermeldung `CB-DB-1018` führt, was auf die Benutzung eines unbekannten Parameters hinweist.
- **Felder** innerhalb eines Datensatzes werden in der Reihenfolge ihrer Definition in der DTD ausgewertet.
- **Datensätze** einer Tabelle werden in den Testdaten von oben nach unten ausgewertet.
- Tabellen werden in der Reihenfolge ihrer Definition in der DTD ausgewertet. Da die DTD diese Reihenfolge auch für die Testdaten erzwingt, ist dies äquivalent zu einer Auswertung von oben nach unten.
- Funktionen und Parameter in den **initialen Testdaten** werden zum Zeitpunkt des Einspielens bzw. bei Aufruf der Methode `assertInitialDataUnchanged()` oder `assertEqualsInitial()` ausgewertet. In den **erwarteten Testdaten** passiert dies bei Aufruf der Methode `assertEqualsExpected()`.
- Da, wie eingangs erwähnt, der Parameter-Kontext vor Ausführen jeder Testmethode neu erstellt wird, ist die Ausführungsreihenfolge der einzelnen Methoden unerheblich.

2.4.10. Einschränken des Datenbankzugriffs

Die meisten Produktionssysteme enthalten sensible Kundendaten, die allein aus datenschutzrechtlichen Gründen vor dem Zugriff durch Entwickler gesperrt sind. In der Praxis besteht somit in der Regel nicht die Gefahr, dass Produktionsdaten durch checkerberry db gelöscht werden. Dennoch gibt es in vielen Unternehmen ausgezeichnete und ungeschützte Integrations- und Testsysteme, die wertvolle Daten enthalten. Dies betrifft vor allem Daten, die aufwändig und mühsam konstruiert wurden, sodass ein Datenverlust einen erheblichen Aufwand verursacht. Das versehentliche Überschreiben kann durch eine Zugriffskontrolle verhindert werden.

Die Zugriffskontrolle verwendet eine Black-, eine White- und eine Read-Only-List. Die Verwendung einer JDBC-URL wird erlaubt, wenn sie nicht in der Blacklist definiert ist. Für eine JDBC-URL, die in der Blacklist eingetragen ist, wird die Verbindung nicht gestattet, es sei denn, es existiert ein entsprechender Eintrag

in der White- oder Read-Only-List, der ebenfalls auf diese URL zutrifft. Ist eine URL in der Read-Only-List eingetragen, wird unabhängig von den anderen beiden Listen stets ein lesender Zugriff gestattet. Bei der Definition einer URL in der Black-, White- oder Read-Only-List können die bekannten Platzhalter „*“ (0 – n beliebige Zeichen) und „?“ (ein beliebiges Zeichen) verwendet werden. Die folgenden Abbildungen zeigen einige Beispielkonfigurationen, in denen zunächst nur die Black- und die Whitelist Einträge enthalten. Die Read-Only-List ist leer und beeinflusst den Zugriff daher nicht.

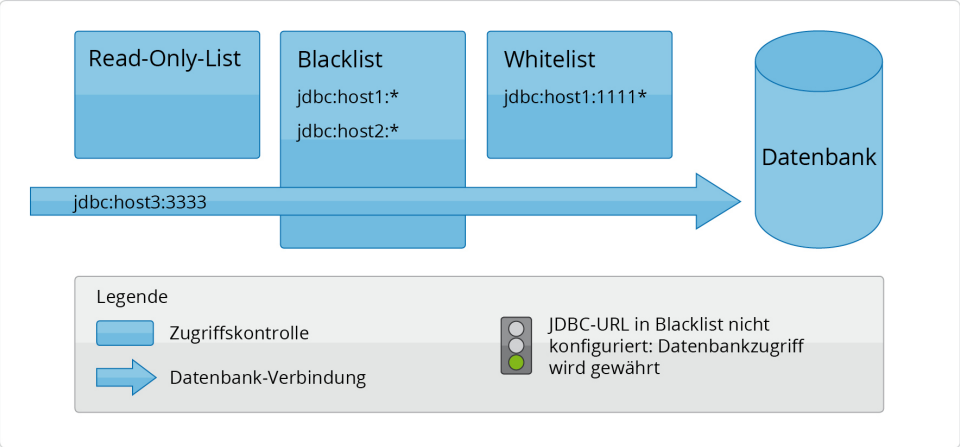


Abbildung 2.16. Zugriffskontrolle

In Abbildung 2.16, „Zugriffskontrolle“ sind sowohl Einträge in der Blacklist als auch in der Whitelist vorhanden. Über die Blacklist werden alle Zugriffe auf die JDBC-URLs von „host1“ und „host2“ unterbunden. In der Whitelist sind die JDBC-URLs für „host1:1111“ explizit freigegeben. In dem Beispiel erfolgt ein Zugriff auf eine JDBC-URL auf „host3“. Da dieser Host in der Blacklist nicht definiert ist, wird er Zugriff durch checkerberry db gewährt.

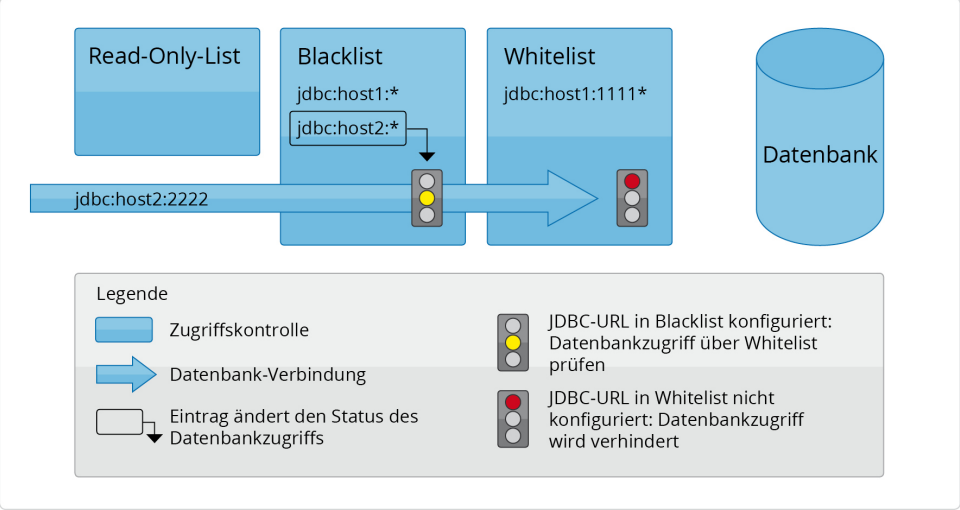


Abbildung 2.17. Blockierter Zugriff

Abbildung 2.17, „Blockierter Zugriff“zeigt ein weiteres Beispiel für die Zugriffskontrolle. Die Konfiguration der Black- und Whitelist entspricht der Konfiguration aus dem vorherigen Beispiel. In diesem Beispiel erfolgt der Zugriff auf eine JDBC-URL von „host2“. In der Blacklist wird ein Eintrag gefunden, der auf die aktuelle JDBC-URL zugeordnet werden kann. Die Freigabe des Zugriffs ist nur möglich, wenn die JDBC-URL auch in der Whitelist konfiguriert ist. Da dies nicht der Fall ist, wird der Zugriff abgewiesen.

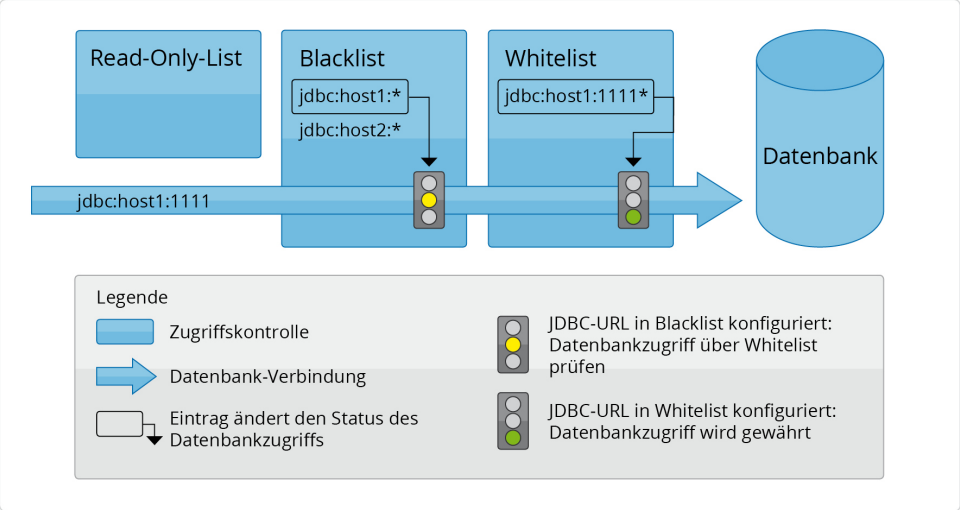


Abbildung 2.18. Whitelist-Freigabe

In Abbildung 2.18, „Whitelist-Freigabe“ ist ein weiteres Beispiel für die Zugriffskontrolle dargestellt. Die Konfiguration der Black- und Whitelist entspricht den Konfigurationen aus den vorherigen Beispielen. In diesem Beispiel erfolgt der Zugriff auf eine JDBC-URL von „host1“. In der Blacklist wird ein Eintrag gefunden, der auf die aktuelle JDBC-URL zugeordnet werden kann. Die Freigabe des Zugriffs ist nur möglich, wenn die JDBC-URL auch in der Whitelist konfiguriert ist. In diesem Beispiel ist in der Whitelist ein Eintrag vorhanden, der der JDBC-URL zugeordnet werden kann. Der Zugriff wird gewährt.

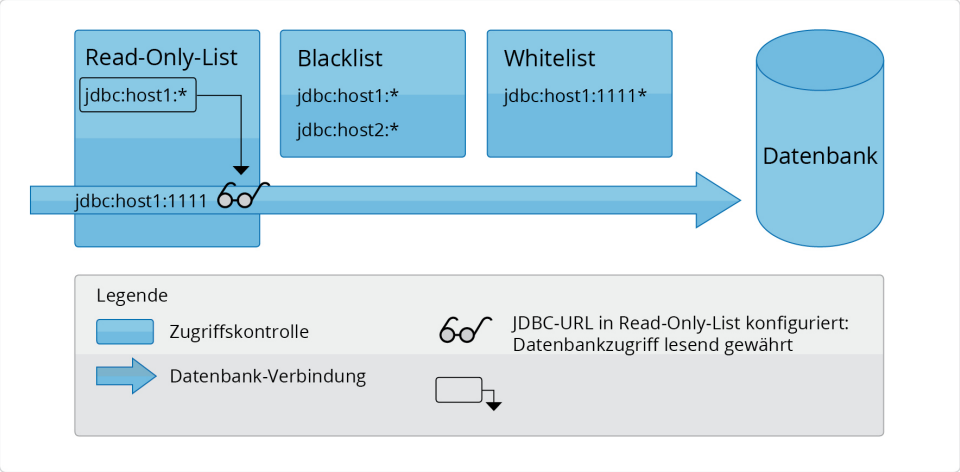


Abbildung 2.19. Read-Only-Freigabe

Auch das in Abbildung 2.19, „Read-Only-Freigabe“ dargestellte Beispiel für eine Zugriffskontrolle besitzt dieselbe Konfiguration der Black- und Whitelist wie die vorherigen Beispiele. Allerdings enthält diesmal die Read-Only-List einen Eintrag, der auf die aktuelle JDBC-URL passt. Der Zugriff wird daher nur lesend durch checkerberry db gewährt. Eine Überprüfung der Black- und Whitelist findet nicht mehr statt.

Die Konfiguration der Black-, White- und Read-Only-List ist in Kapitel Abschnitt 2.5.2.3, „Konfiguration von checkerberry db mit dem DbConfigurationCallback“ beschrieben.

2.4.10.1. Konfiguration der Zugriffskontrolle

Die Konfiguration der Zugriffskontrolle erfolgt über das `DbConfigurationCallback`.

```
public class ConfigurationCallback implements DbConfigurationCallback {
  public void configure(DbConfiguration configuration) {
    // Alle JDBC-URLs verbieten
    configuration.addToAccessControlBlacklist("*");
    // Alle JDBC-URLs auf localhost Vollzuriff erlauben
    configuration.addToAccessControlWhitelist("jdbc:*://localhost:*");
    // JDBC-URLs auf host und Port 1512 Lesezuriff erlauben
    configuration.addToAccessControlReadOnlyList("jdbc:*://host:1512");
}
```

Beispiel 2.34. Zugriffskontrolle

Das obige Beispiel zeigt eine mögliche Konfiguration der Zugriffskontrolle. Zunächst werden alle JDBC-URLs über die Wildcard „*“ in die Blacklist aufgenommen. Durch den Eintrag von „jdbc:*://localhost:*“ in der Whitelist, werden alle lokalen JDBC-Verbindungen erlaubt. Da in der Read-Only-List kein Eintrag enthalten ist, der die lokalen JDBC-Verbindung auf einen Lesezugriff einschränkt, ist für diese der Vollzugriff gestattet. Weil zudem "jdbc:*://host:1512" der Read-Only-List hinzugefügt wurde kann über den Port 1512 lesend auf host zugegriffen werden.

2.4.11. Datenbank durch Annotation leeren

Beim Einspielen der initialen Testdaten leert checkerberry db alle Tabellen, die in der DTD definiert sind. Danach werden die initialen Testdaten eingespielt. Möchte man einen Test mit leerer Datenbank starten, kann man somit eine Datei mit leeren Initialdaten angeben. Alternativ kann man die Annotation `ClearTables` nutzen. Die Verwendung der Annotation hat den Vorteil, dass die Information explizit am Test angegeben wird. Des Weiteren entfällt die explizite Erstellung einer leeren initialen Testdatei.

Bei der Kombination von initialen Testdaten und `ClearTables`-Annotationen gilt folgendes Verhalten:

1. Es ist keine `ClearTables`-Annotation vorhanden:
   - Sind initiale Testdaten für die Methode definiert, werden diese eingespielt.
   - Sind initiale Testdaten für die Klasse definiert, werden diese eingespielt.
   - Sind keine initialen Testdaten definiert, wird der Datenbankinhalt nicht verändert.
2. Es ist eine `ClearTables`-Annotation an der Methode annotiert:
   - Sind initiale Testdaten für die Methode definiert, wird die checkerberry Fehlermeldung `CB-DB-1043` erzeugt. Da nicht klar ist, ob die Methode initiale Testdaten einspielen oder die Datenbank leeren soll, kann dieser Konflikt nicht automatisch aufgelöst werden.
   - Sind initiale Testdaten für die Klasse definiert, wird die Datenbank dennoch geleert, da das für die Methode definierte Verhalten das für die Klasse definierte Verhalten überschreibt.
   - Sind keine initialen Testdaten definiert, wird der Datenbankinhalt geleert.
3. Es ist eine `ClearTables`-Annotation an der Klasse annotiert:
   - Sind initiale Testdaten für die Methode definiert, werden diese eingespielt, da das für die Methode definierte Verhalten das für die Klasse definierte Verhalten überschreibt.
   - Sind initiale Testdaten für die Klasse definiert, wird Fehlermeldung `CB-DB-1042` erzeugt. Da nicht klar ist, ob die Klasse initiale Testdaten einspielen oder die Datenbank leeren soll, kann dieser Konflikt nicht automatisch aufgelöst werden.
   - Sind keine initialen Testdaten definiert, wird der Datenbankinhalt geleert.

Die folgenden Code-Beispiele demonstrieren die Verwendung der `ClearTables`-Annotation.

```
public class ClearTablesAnnotatedMethodTest {

  @Test
  @ClearTables
  public void testAnnotated() {
    // Sind für diese Methode Initialdaten definiert,
    // schlägt der Test bereits im SetUp fehl.
    // Wenn nicht, wird der Datenbankinhalt geleert.
  }
}
```

Beispiel 2.35. ClearTables Annotation an Methode

```
@ClearTables
public class ClearTablesAnnotatedClassTest {

  @Test
  public void testAnnotated() {
    // Sind für diese Methode Initialdaten definiert,
    // werden diese eingespielt.
    // Wenn nicht und für die Klasse sind Initialdaten definiert,
    // dann schlägt der Test bereits im SetUp fehl.
    // Sind weder auf Klassen- noch auf Methodenebene Initialdaten
    // definiert, wird der Datenbankinhalt geleert.
  }
}
```

Beispiel 2.36. ClearTables-Annotation an Klasse

Zur Performance-Optimierung können Tabellen von checkerberry db gecacht werden (siehe Abschnitt 2.4.14, „Performance-Optimierung durch Caching von Tabellen"). Per Default wird die `ClearTables`-Annotation so interpretiert, dass die Inhalte aller Tabellen gelöscht werden. Dies betrifft insbesondere auch gecachte Tabellen. Möchte man den Cache durch die `ClearTables`-Annotation nicht löschen, kann die Annotation mit dem Attribut *`@ClearTables(TableTypes.NON_CACHEABLE)`* verwendet werden.

2.4.12. Verwenden von leeren Tabellen

Möchte man bei dem Einspielen von initialen Testdaten eine Tabelle leer lassen, so wird das implizit dadurch erreicht, dass man keine Datensätze für diese Tabelle definiert. Über die Angabe des Tags *`EMPTY_TABLE`*, kann jedoch auch explizit sichergestellt werden, dass keine Datensätze für diese Tabelle eingespielt werden. Der Name der Tabelle wird in diesem Fall über das Attribut *`TABLENAME`* angegeben.

```
<dataset>
  <!-- Die Tabelle PIZZA soll leer sein. -->
  <EMPTY_TABLE TABLENAME="PIZZA" />
  <!-- Die Tabelle USERS soll Daten enthalten. -->
  <USERS NAME="Marge" SURNAME="Simpson" />
  <USERS NAME="Homer" SURNAME="Simpson" />
  <USERS NAME="Bart" SURNAME="Simpson" />
  <USERS NAME="Lisa" SURNAME="Simpson" />
  <USERS NAME="Maggie" SURNAME="Simpson" />
</dataset>
```

Beispiel 2.37. Explizite Angabe einer leeren Tabelle

Die explizite Angabe einer initial leeren Tabelle garantiert, dass keine Datensätze für diese Tabelle eingespielt werden. Dies könnte z.B. passieren, wenn initialen Testdaten für diese Tabelle in einer inkludierten Datei vorhanden sind. Entsteht ein solcher Widerspruch in den Testdaten, wird die checkerberry db Fehlermeldung *`CB-DB-1044`* erzeugt.

Das *`EMPTY_TABLE`* Tag kann nicht nur in den initialen, sondern auch in den erwarteten Testdaten genutzt werden. Die folgende Tabelle stellt den Unterschied zwischen impliziter Annahme einer leeren Tabelle durch Weglassen der Definition und expliziter Angabe eines *`EMPTY_TABLE`* Tags dar.

	Initiale Testdaten	Erwartete Testdaten
Es sind Datensätze für Tabelle XXX definiert.	Die Datensätze werden in die Datenbank eingespielt.	Es wird überprüft, ob die Tabelle genau diese Datensätze enthält.
Es sind keine Datensätze für die Tabelle XXX definiert.	Die ggf. vorhandenen Datensätze der Tabelle werden in der Datenbank gelöscht.	Diese Tabelle wird während der Überprüfung nicht berücksichtig, d.h. auch wenn für diese Tabelle Datensätze in der Datenbank vorhanden sind, verläuft die Überprüfung positiv.
Für die Tabelle XXX ist das EMPTY_TABLE Tag definiert.	Die ggf. vorhandenen Datensätze der Tabelle werden in der Datenbank gelöscht.	Es wird überprüft, ob diese Tabelle in der Datenbank leer ist.
Es sind Datensätze für Tabelle XXX definiert und für die Tabelle XXX ist das EMPTY_TABLE Tag definiert.	Die checkerberry db Fehlermeldung *CB-DB-1044* wird erzeugt.	Die checkerberry db Fehlermeldung *CB-DB-1044* wird erzeugt.

2.4.13. Erstellen von Datenbank-Dumps im XML-Testdatenformat

Checkerberry db bietet die Möglichkeit den gesamten Datenbankinhalt oder den Inhalt ausgewählter Tabellen im Testdatenformat zu speichern.

```
public void testAnything() throws Exception {

  // Holen des Testhandlers.
  DbTestHandler testHandler = getEnvironment().getTestHandler();
  // Gesamte Datenbank in XML speichern.
  testHandler.dumpDatabase("c:/temp/db-dump.xml");
  // Tabelle USERS in XML speichern.
  testHandler.dumpTable("c:/temp/users-dump.xml", "USERS");
  // Tabellen ADDRESS und USERS in XML speichern.
  testHandler.dumpTables("c:/temp/partial-dump.xml", "ADDRESS", "USERS");
}
```

Beispiel 2.38. Erstellen von Datenbank-Dumps

Das obige Beispiel zeigt die verschiedenen Möglichkeiten Datenbankinhalte aus der Testmethode heraus zu speichern. Über die Methode `dumpDatabase` wird der gesamte Datenbankinhalt in einer XML-Datei gespeichert, während die Methode `dumpTable` bzw. `dumpTables` nur eine bzw. ausgewählte Tabelleninhalte in einer XML-Datei speichern. Die XML-Dateien werden gemäß der konfigurierten DTD erstellt, sodass sie die gleiche Struktur wie die Testdaten aufweisen.

Die folgenden Abbildungen verdeutlichen das Verhalten der verschiedenen Dump-Methoden. In allen Abbildungen ist eine Datenbank dargestellt, die für die drei Tabellen `PIZZA`, `TOPPING` und `USER` jeweils einen Eintrag enthalten.

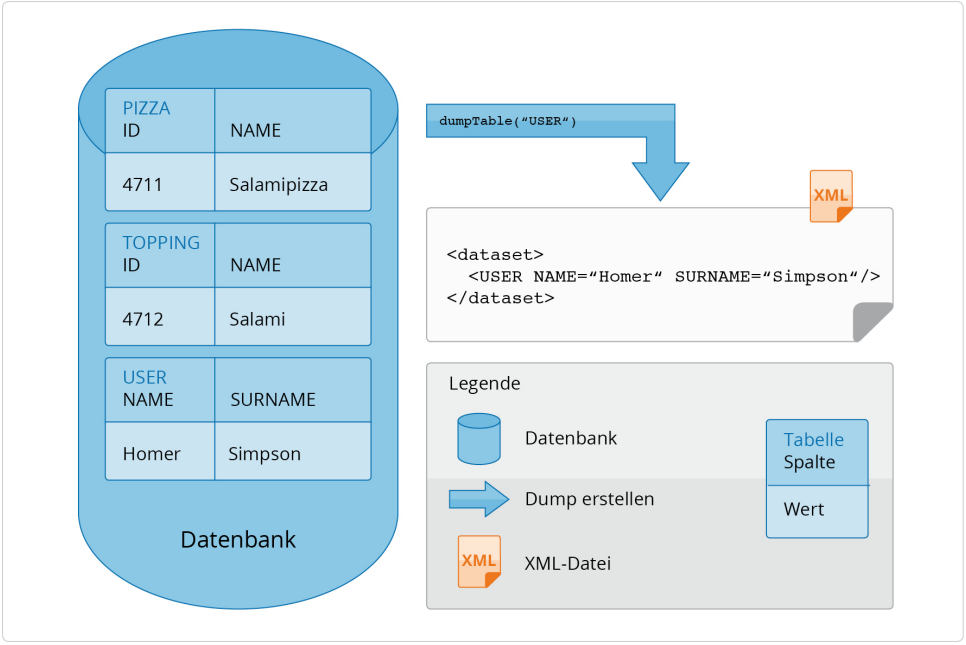


Abbildung 2.20. Datenbank-Dump für eine Tabelle

In Abbildung 2.20, „Datenbank-Dump für eine Tabelle" wird die Methode `dumpTable` für die Tabelle `USER` aufgerufen. Das Ergebnis ist eine XML-Datei, die nur den Inhalt der Tabelle `USERS` beinhaltet.

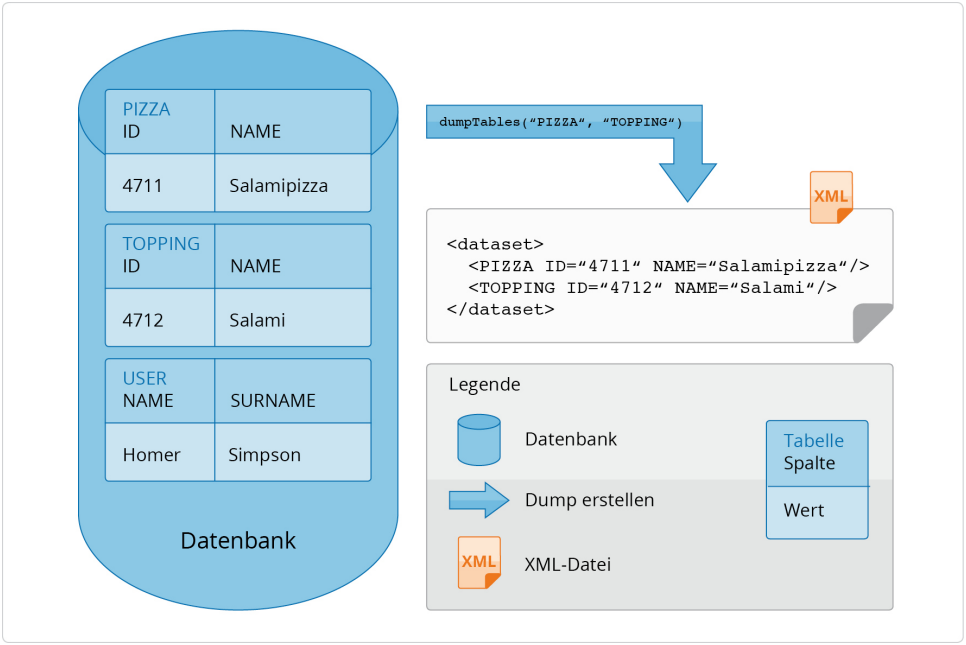


Abbildung 2.21. Datenbank-Dump für mehrere Tabellen

In Abbildung 2.21, „Datenbank-Dump für mehrere Tabellen" wird die Methode `dumpTables` für die Tabellen `PIZZA` und `TOPPING` aufgerufen. Das Ergebnis ist eine XML-Datei, die nur die Inhalte der Tabellen `PIZZA` und `TOPPING` beinhaltet.

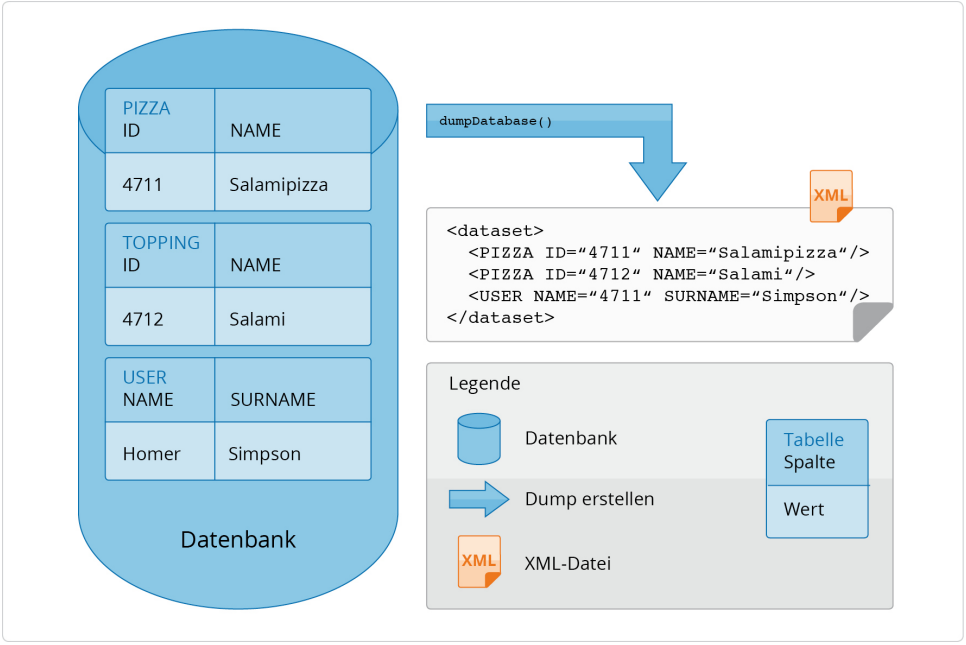


Abbildung 2.22. Datenbank-Dump für die gesamte Datenbank

In Abbildung 2.22, „Datenbank-Dump für die gesamte Datenbank" wird die Methode `dumpDatabase` aufgerufen. Das Ergebnis ist eine XML-Datei, die alle Tabelleninhalte der Datenbank beinhaltet. Die XML-Datei enthält somit die Inhalte der Tabellen `USER`, `PIZZA` und `TOPPING`.

Möchte man den Datenbank-Dump übersichtlicher halten, bietet es sich an, die cacheable Tabellen aus dem Datenbank-Dump auszuschließen. Als cacheable werden ohnehin meistens Tabellen definiert, die Daten enthalten, die während der Testdurchführung nicht geändert werden. Um den Datenbank-Dump auf die nicht cacheable Tabellen zu beschränken, kann die Methode *`dumpDatabase(String targetFile, boolean excludeCacheableTables)`* verwendet werden, wobei der Boolean *`excludeCacheableTables`* auf *`true`* gesetzt werden muss.

2.4.13.1. Feingranulares, kaskadiertes Erstellen von Datenbank-Dumps

Wenn die Daten zu umfangreich sind oder man bestimmte Daten für ein Testszenario auswählen möchte, kann man Daten anhand eines Regelwerks auswählen und in eine XML-Datei exportieren. Die Konsistenz der Daten wird dabei anhand von Fremdschlüsselinformationen sichergestellt.

```
public void testAnything() throws Exception {

  // Holen des Testhandlers.
  DbTestHandler testHandler = getEnvironment().getTestHandler();

  // Selektiert die Zeilen aus USERS mit ID=1
  Recipe recipe = Recipe.with(Selection.table("USERS")
                      .where(TableConstraint.key("ID").eq().to(1))
                      .build())
                  .build();

  // Selektiert alle Zeilen aus USERS mit ID>1 und ID<10
  Recipe recipe = Recipe.with(Selection.table("USERS")
                      .where(TableConstraint.key("ID").type(Type.GT).to(1))
                      .where(TableConstraint.key("ID").type(Type.LT).to(10))
                      .build())
                  .build();

  // Selektiert zusätzlich alle Zeilen aus LOCATIONS mit CityCode='HH'
  Recipe recipe = Recipe.with(Selection.table("USERS")
                      .where(TableConstraint.key("ID").eq().to(1))
                      .build())
                  .and(Selection.table("LOCATIONS")
                      .where(TableConstraint.key("CityCode").eq().to("HH"))
                      .build())
                  .build();

  // Daten aus Quelltabelle und alle referenzierten Daten in XML speichern.
  testHandler.dumpTables("c:/temp/db-dump.xml", recipe);
}
```

Beispiel 2.39. Feingranulares Erstellen von Datenbank-Dumps

Zur Veranschaulichung der Funktionsweise gehen wir von folgenden Beispieldaten aus. In der Datenbank finden sich 3 Tabellen mit folgenden Werten:

Tabelle 2.3. Tabelle USERS

ID	NAME	SURNAME	BIRTHDATE	WEIGHT
1	Homer	Simpson	1946-09-16	80
2	Marge	Simpson	1950-09-16	60

Tabelle 2.4. Tabelle DISH

DISH_ID	DESCRIPTION
1	Donuts
2	Cheese

Tabelle 2.5. Tabelle USERDISHREL

ID	DISH_ID
1	1
2	1

Bei diesen Tabellen gibt es jeweils eine ForeignKey-Beziehung von `USERS` und `DISH` auf `USERDISHREL` (`USERS.ID -> USERDISHREL.ID` und `DISH.ID -> USERDISHREL.DISH_ID`). Wenn nun ein Eintrag

aus USERS selektiert wird, benötigt man für konsistente Daten auch alle Zeilen aus USERDISHREL die durch die ForeignKey-Beziehung adressiert werden. Ausgehend von den aus USERDISHREL selektierten Zeilen werden dann noch alle Zeilen aus DISH benötigt, die durch die ForeignKey-Beziehung zwischen DISH und USERDISHREL adressiert werden.

```
// Selektiert die Zeilen aus USERS mit ID=1
Recipe recipe = Recipe.with(Selection.table("USERS")
                    .where(TableConstraint.key("ID").eq().to(1))
                    .build())
                .build();
```

```
// Inhalt des Datenbank-Dump:
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC ...>
<dataset>
  <USERS ID="1" NAME="Homer" SURNAME="Simpson" BIRTHDATE="1946-09-16"
    WEIGHT="80.0"/>
  <USERDISHREL ID="1" DISH_ID="1"/>
  <DISH DISH_ID="1" DESCRIPTION="Donuts"/>
</dataset>
```

Beispiel 2.40. Datenbankdump ausgehend von der USERS-Tabelle

Wird zusätzlich zum Eintrag aus USERS eine weitere Selektion auf DISH definiert, dann benötigt man auch alle Zeilen aus USERDISHREL und USERS, die nach dem selben Schema durch die ForeignKey-Beziehungen adressiert werden.

```
// Selektiert die Zeilen aus USERS mit ID=1
Recipe recipe = Recipe.with(Selection.table("USERS")
                    .where(TableConstraint.key("ID").eq().to(1))
                    .build())
                .and(Selection.table("DISH")
                    .where(TableConstraint.key("DISH_ID").eq().to(1))
                    .build())
                .build();
```

```
// Inhalt des Datenbank-Dump:
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC ...>
<dataset>
  <USERS ID="1" NAME="Homer" SURNAME="Simpson" BIRTHDATE="1946-09-16"
    WEIGHT="80.0"/>
  <USERS ID="2" NAME="Marge" SURNAME="Simpson" BIRTHDATE="1950-09-16"
    WEIGHT="60.0"/>
  <USERDISHREL ID="1" DISH_ID="1"/>
  <USERDISHREL ID="2" DISH_ID="1"/>
  <DISH DISH_ID="1" DESCRIPTION="Donuts"/>
</dataset>
```

Beispiel 2.41. Datenbankdump ausgehend von der USERS- und der DISH-Tabelle

Wenn in der Datenbank Fremdschlüsselinformation (oder Primärschlüsselinformationen) fehlen, kann diese für den Export ergänzt werden.

```
// Definiert einen PrimaryKey mit der Spalte ID für USERDISHREL
Recipe recipe = Recipe.with(Selection.table("USERS")
                    .where(TableConstraint.key("ID").eq().to(1))
                    .build())
                .primaryKey("USERDISHREL", "ID")
                .build();

// Definiert eine ForeignKey-Beziehung von der Tabelle USERDISHREL
//   und der Spalte DISH_ID
//   auf Tabelle DISH und Spalte DISH_ID
Recipe recipe = Recipe.with(Selection.table("USERS")
                    .where(TableConstraint.key("ID").eq().to(1))
                    .build())
                .foreignKey(ForeignKey.from("USERDISHREL", "DISH_ID")
                    .to("DISH", "DISH_ID"))
                .build();
```

Beispiel 2.42. Zusätzliche Definition von Primär- und ForeignKeys

Wenn ein komplexes Domainmodell vorliegt, dass über sehr viele Fremdschlüsselbeziehungen verfügt, kann es vorkommen, dass beim Export der Daten das voreingestellte Limit für die Anzahl der Durchläufe überschritten wird. In diesem Fall kann man entweder die Auswahl eingrenzen oder das Limit erhöhen.

Die Anzahl der Durchläufe korreliert in einfachen Modellen mit der Anzahl der durch Fremdschlüssel referenzierten Tabellen. Für den einfachen Fall ist ein Wert der doppelt so hoch ist wie die Anzahl der im Export erwarteten Tabellen ein guter Startpunkt.

```
public class ConfigurationCallback implements DbConfigurationCallback {
    public void configure(DbConfiguration configuration) {
        configuration.setDumpRecipeMaxLoops(100);
    }
}
```

Beispiel 2.43. Erhöhen der maximalen Anzahl der Durchläufe

2.4.13.2. Analysieren von Problemen mit Hilfe von Datenbank-Dumps

In der Software-Entwicklung tritt häufig das Problem auf, dass Fehler nicht in der lokalen Entwicklungsumgebung, sondern auf Test- oder Auslieferungssystemen festgestellt werden. Die Analyse auf diesen Systemen ist in der Regel aufwändig, da mit Ausnahme der Log-Ausgaben nur wenige Analyse-Werkzeuge zur Verfügung stehen.

Die Reproduktion des Fehlerfalls ist oftmals möglich, wenn in der lokalen Entwicklungsumgebung der gleiche Datenbankinhalt wie in dem externen System angelegt wird. Durch geringe Konfigurationsänderungen an checkerberry db kann der Datenbankinhalt von externen Systemen in einer XML-Datei gespeichert werden. Zu diesem Zweck muss lediglich die JDBC-Verbindung von checkerberry db umgehängt werden. Die genaue Änderung ist dabei von der Umgebung, in der checkerberry db eingesetzt wird, abhängig.

Achtung: Beachten Sie, dass checkerberry db auch versucht Testdaten in das externe System einzuspielen, wenn entsprechende initiale Testdaten gefunden werden. Sofern die Rechte auf der Datenbank eine Änderung zulassen, kann der gesamte Datenbankinhalt des externen Systems überschrieben werden! Ändern Sie nur temporär die Konfiguration, erstellen Sie den Dump und setzen dann die Konfiguration wieder zurück! Verwenden Sie für den Zugriff auf sensible Daten außerdem einen Datenbankbenutzer, der ausschließlich Leserechte besitzt!

2.4.13.3. Erstellen von initialen Testdaten mit Hilfe von Datenbank-Dumps

In manchen Konstellationen können die erstellten Dumps als Basis für initiale Testdaten verwendet werden. Dies ist zum Beispiel für das oben beschriebene Szenario sinnvoll: Auf einem externen System wird ein Fehler festgestellt. Die Daten des externen Systems werden zu Analysezwecken übernommen und der Fehler kann reproduziert werden. Nachdem die Fehlerursache ermittelt wurde, kann der Dump so angepasst werden, dass er als Basis für einen Test mit checkerberry db fungiert.

Es gibt auch Situationen, in denen die manuelle Erstellung der initialen Testdaten aufwändig und fehleranfällig ist. In diesen Situationen kann es sinnvoll sein, die Daten z.B. durch die zu testende Anwendung selbst zu erstellen. Durch die Erstellung eines Datenbank-Dumps können dann die Daten als Basis für die initialen Testdaten verwendet werden. Die nicht benötigten Datensätze sollten jedoch aus den gedumpten Testdaten entfernt werden, um die Übersichtlichkeit der Testdaten zu wahren.

2.4.13.4. Erstellen von erwarteten Testdaten mit Hilfe von Datenbank-Dumps

Der Prozess zur Erstellung von automatisierten Tests wird durch die Erzeugung von Datenbank-Dumps vereinfacht. Der Entwickler erstellt die initialen Testdaten und programmiert den Test. Bevor die erwarteten Testdaten im Test überprüft werden, wird ein Dump des Datenbankinhalts erstellt. Die Aufgabe des Entwicklers besteht nun darin, den erstellten Dump auf Korrektheit zu überprüfen, um ihn dann als erwartete Testdaten zu definieren.

Dieses Vorgehen kann die Testerstellung massiv beschleunigen, da die Definition von erwarteten Testdaten sehr aufwändig sein kann. Die Kontrolle der Ergebnisdaten auf Korrektheit ist dagegen meist sehr viel schneller zu erledigen.

2.4.13.5. Encoding der Datenbank-Dumps ändern

Durch die Methode `setDumpFileEncoding` in `DbConfiguration` kann das Encoding der erstellten Datenbank-Dumps konfiguriert werden. Das Standard-Encoding ist UTF-8.

Bei der Erstellung von Datenbank-Dumps kodiert checkerberry db alle Sonderzeichen in der XML-Datei. Dies gilt insbesondere für Umlaute. Aus diesem Grund kann man über die Methode `setDumpFileDeEscapedCharacters` konfigurieren, welche Sonderzeichen nicht kodiert werden sollen. Der Methode werden alle Sonderzeichen als eine Zeichenkette übergeben. Als Standard wird die Zeichenkette „äöüßÄÖÜ“ verwendet, d.h. die Umlaute und das ß werden im XML nicht kodiert.

Beispiel 2.44, „Konfigurationseinstellungen für Datenbank-Dumps“ enthält ein Beispiel für diese Konfigurationsmöglichkeit.

```
public class ConfigurationCallback implements DbConfigurationCallback {

  public void configure(DbConfiguration configuration) {
    // Setzen des Encodings für die Datenbank-Dumps auf UTF-16.
    configuration.setDumpFileEncoding("UTF-16");
    // Die Zeichen ä und Ä werden in den Datenbank-Dumps nicht durch
    // &#228; bzw. &#196; kodiert.
    configuration.setDumpFileDeEscapedCharacters("äÄ");
  }
}
```

Beispiel 2.44. Konfigurationseinstellungen für Datenbank-Dumps

2.4.14. Performance-Optimierung durch Caching von Tabellen

Tabellen können in den Tabellenbeschreibungen folgendermaßen als cacheable markiert werden:

```
databaseDescription.addTableDescription("TOPPING", Cacheable.Yes, "NAME");
```

Beispiel 2.45. Markieren einer Tabelle als cacheable

Das Markieren von Tabellen als cacheable geschieht ausschließlich bei der Erstellung der Datenbankbeschreibung im `DatabaseDescriptionCallback`. Nähere Informationen hierzu finden sich in Abschnitt 2.5.2, „Implementierung der konkreten checkerberry-db-Bridge“.

Bei dem Einspielen von initialen Testdaten prüft checkerberry db für jede Tabelle, ob sie cacheable ist und ob sie bereits in die Datenbank eingespielt wurde. Wenn die Tabelle cacheable ist und noch nicht im Cache vorhanden ist, werden die Daten in die Datenbank eingespielt und der Name der Tabelle im Cache gespeichert. Ist die Tabelle bereits in Cache vorhanden, werden die zugehörigen Daten nicht in die Datenbank eingespielt.

Das Caching-Verfahren bietet dem Benutzer die Möglichkeit, statische Stammdaten nur einmal pro Testlauf in die Datenbank zu laden. Nachfolgende Tests nutzen somit dieselben Testdaten wie die vorherigen Tests.

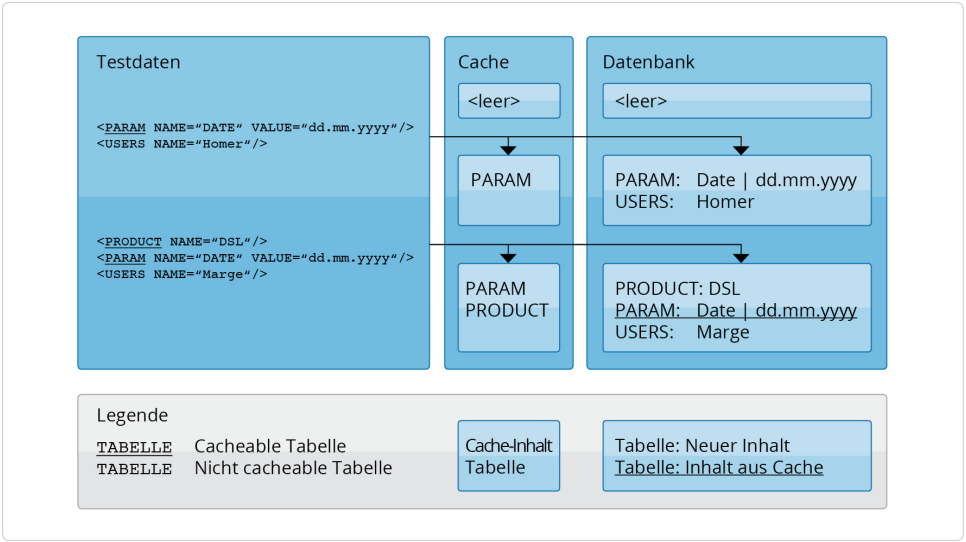


Abbildung 2.23. Cache

Die obige Abbildung stellt die Funktionsweise des Cache-Verfahrens dar. Auf der linken Seite sind die initialen Testdaten aufgeführt, die in die Datenbank eingespielt werden sollen. In der Mitte wird der Zustand des Caches beschrieben. Der Datenbankinhalt wird auf der rechten Seite dargestellt. In dem dargestellten Beispiel sind zunächst der Cache als auch die Datenbank leer. Die ersten Testdaten enthalten einen Eintrag für die cacheable Tabelle `PARAM` und einen Eintrag für die nicht-cacheable Tabelle `USERS`. Die Daten

werden in die Datenbank eingespielt und der Name der cacheable Tabelle `PARAM` wird in den Cache aufgenommen.

Die zweiten Testdaten enthalten die beiden cacheable Tabellen `PRODUCT` und `PARAM` und die nicht-cacheable Tabelle `USERS`. Beim Einspielen der Daten werden zunächst die nicht-cacheable Tabellen geleert. In diesem Beispiel bedeutet das, dass die Inhalte der Tabelle `USERS` gelöscht werden. Dann wird der Datensatz für die Tabelle `PRODUCT` in die Datenbank eingespielt und der Tabellenname wird zu dem Cache hinzugefügt. Die Daten für die Tabelle `PARAM` werden nicht eingespielt, da der Tabellenname bereits im Cache vorhanden ist. Der Datensatz für die Tabelle `USERS` wird wiederum in die Datenbank eingespielt.

Aus Performance-Gründen prüft checkerberry db lediglich die Namen der Tabellen und nicht deren Inhalt. Es kann somit vorkommen, dass die initialen Testdaten für einen Test nicht mit dem eingespielten Datenbankinhalt übereinstimmen, da cacheable Tabellen eines vorherigen Tests nicht überschrieben wurden. Um diese potentielle Fehlerquelle zu umgehen, sollten alle Tests, die den Cache nutzen, die gleichen Werte im Cache erwarten. Aus diesem Grund ist es empfehlenswert, die Werte von cacheable Tabellen in eine eigene Include-Datei auszulagern. Diese Include-Datei sollte dann von jedem Test eingebunden werden, der eine Teilmenge der cacheable Tabellen benötigt (siehe auch Abschnitt 7.2.6, „Lagere Caching-Daten in Includes aus“).

Die cacheable Tabellen sollten so gewählt sein, dass ein Großteil der Tests die gleichen Daten verwenden kann. Das Verfahren bietet sich somit insbesondere für statische Konfigurations- oder Stammdaten an.

2.4.14.1. Leeren des Cache

Die Vorteile des Caching sind schnell nachzuvollziehen, doch wie testet man unterschiedliche Ausprägungen der cacheable Tabellen? In der Regel gibt es auch für die cacheable Tabellen eine Reihe von Komponenten, die getestet werden müssen. Aus diesem Grund stellt checkerberry db eine Möglichkeit zur Verfügung, den Inhalt des Caches zu löschen.

```
// Cache für Tabellen PROPERTIES und COUNTRIES nicht verwenden, d.h.
// entfernen dieser Tabellen aus dem Cache und für diesen Test nicht neu in
// den Cache aufnehmen.
@NoCache("PROPERTIES", "COUNTRIES")
public void testAnything() throws Exception {
  …
}
```

Beispiel 2.46. Leeren des Cache

Das obige Beispiel zeigt, wie der Cache durch Angabe der Annotation `@NoCache` geleert werden kann. Die Annotation kann mit einer Liste von Tabellennamen oder ohne Parameter aufgerufen werden. Wenn kein Parameter angegeben wird, bezieht sich die Annotation auf den gesamten Cache.

Die Annotation bewirkt zum einen, dass die angegebenen Tabellen aus dem Cache entfernt werden. Das führt dazu, dass in der Setup-Phase die cacheable Tabellen durch die initialen Testdaten überschrieben werden. Zum anderen führt die Annotation dazu, dass die neuen Werte der cacheable Tabellen nicht in den Cache aufgenommen werden. Der Cache wird für den Test und die angegebenen Tabellen somit temporär deaktiviert.

Die Annotation ist sowohl auf Methoden- als auch auf Klassenebene anwendbar. Des Weiteren ist es möglich, das Caching zu Testzwecken global zu deaktivieren, ohne dass man eine Annotation an jede Klasse oder Methode schreiben muss (siehe Abschnitt 2.4.14.5, „Globales Deaktivieren des Cache“).

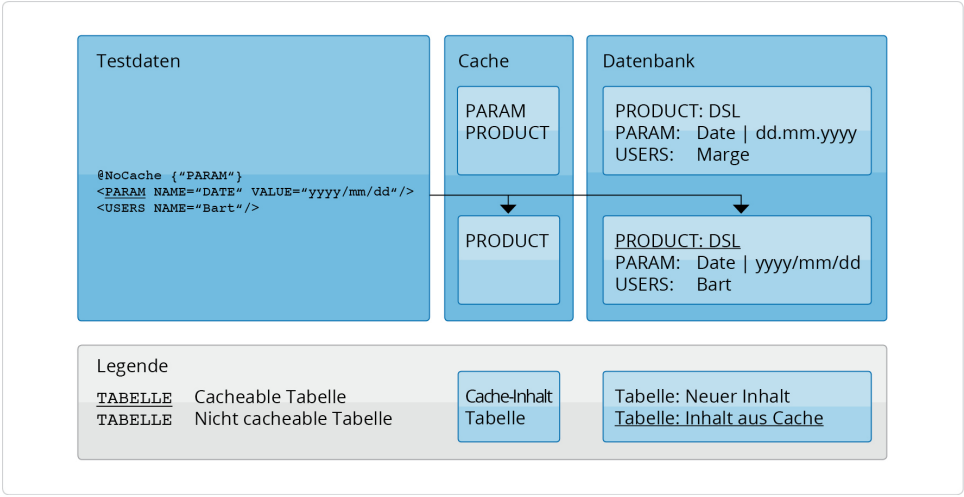


Abbildung 2.24. NoCache-Annotation

Die obige Abbildung stellt beispielhaft die Verwendung der `NoCache`-Annotation dar. Vor dem Einspielen der Testdaten enthält der Cache die Tabellen `PARAM` und `PRODUCT`. Die Datenbank enthält für die Tabellen `PRODUCT`, `PARAM` und `USERS` jeweils einen Eintrag. Die initialen Testdaten gehören zu einer Testmethode oder Testklasse, die über die `NoCache`-Annotation das Caching für die Tabelle `PARAM` unterbindet. Vor dem Einspielen der Testdaten wird der Tabellenname `PARAM` aus dem Cache entfernt. Die Datenbankinhalte der Tabellen `PARAM` und `USERS` werden gelöscht. Danach werden die initialen Testdaten eingespielt. Die Datensätze für die Tabellen `PARAM` und `USERS` werden in die Datenbank eingespielt. Die Tabelle `PARAM` wird dabei jedoch nicht in den Cache aufgenommen, obwohl sie als cacheable definiert ist. Der Inhalt der Tabelle `PRODUCT` bleibt unverändert, da diese Tabelle bereits im Cache vorhanden ist.

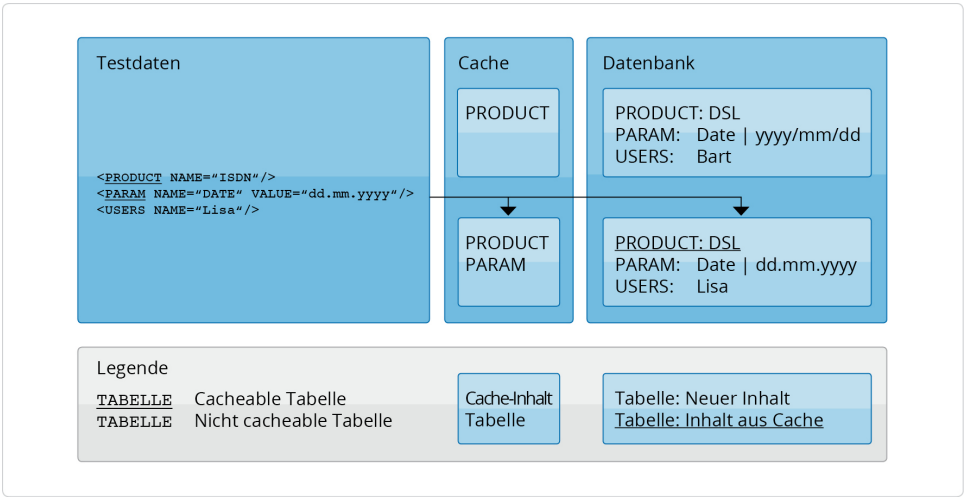


Abbildung 2.25. Wachsender Cache

Das Beispiel in Abbildung 2.25, „Wachsender Cache“ zeigt, wie der Cache nach einer `NoCache`-Annotation wieder aufgebaut wird. Der Anfangszustand des Beispiels entspricht dem Endezustand des vorherigen Beispiels. Die initialen Testdaten enthalten jeweils einen Eintrag für die Tabellen `PRODUCT`, `PARAM` und `USERS`. Die Tabellen `PRODUCT` und `PARAM` sind cacheable. Vor dem Einspielen der Testdaten werden die Inhalte der Tabellen `PARAM` und `USERS` gelöscht. Danach werden die Testdaten eingespielt. Der Datensatz der Tabelle `PRODUCT` wird nicht in die Datenbank geschrieben, da diese Tabelle bereits im Cache vorhanden ist. Der Datensatz für die Tabelle `PARAM` wird hingegen in die Datenbank geschrieben. Des Weiteren wird der Tabellenname in den Cache aufgenommen. Der Datensatz der Tabelle `USERS` wird in die Datenbank eingespielt.

Am Anfang des Caching-Kapitels wurde bereits die potenzielle Fehlerquelle abweichender Cache-Inhalte beschrieben. Die obige Abbildung liefert ein Beispiel für dieses Problem. Die Tabelle `PRODUCT` ist mit dem Wert „DSL" in der Datenbank vorhanden. Da die Tabelle im Cache vorhanden ist, werden die Daten in der Datenbank nicht überschrieben, obwohl in den initialen Testdaten der Wert „ISDN" eingespielt werden soll.

2.4.14.2. Caching von Includes

Checkerberry db speichert in dem Cache ebenfalls Informationen über inkludierte Testdatendateien. Wenn eine Datei über ein Include eingebunden wird, speichert checkerberry db, welche Tabellen in dieser Include-Datei vorhanden sind. Wenn diese Datei in einem nachfolgenden Test erneut eingebunden wird, prüft checkerberry db, ob ggf. alle Tabellen aus der Include-Datei bereits im Cache vorhanden sind. In diesem Fall wird diese Datei gar nicht erst eingelesen.

Wie reagiert checkerberry db, wenn die Include-Dateien wiederum andere Include-Dateien beinhalten? Checkerberry db löst dieses Problem, indem jeder Include-Datei auch alle verschachtelt referenzierten Tabellen zugeordnet werden. Auf diese Art und Weise ist sichergestellt, dass Include-Dateien nur ignoriert werden, wenn auch die verschachtelt hinzugefügten Tabellen bereits im Cache vorhanden sind.

2.4.14.3. Caching und Löschen von Tabellen

Per Default wird die ClearTables-Annotation so interpretiert, dass initial die Inhalte aller Tabellen gelöscht werden sollen (siehe Abschnitt 2.4.11, „Datenbank durch Annotation leeren"). Es ist jedoch auch möglich, nur die nicht zu cachenden Tabellen zu löschen und den Cache unverändert zu lassen, indem man die Annotation mit dem Attribut `@ClearTables(TableTypes.NON_CACHEABLE)` verwendet.

2.4.14.4. Caching von leeren Tabellen

Leere Tabellen werden nicht gecacht. Wird eine gecachte Tabelle in mehreren Tests leer verwendet und ein folgender Test definiert Initialdaten für sie, werden diese Datensätze in die Datenbank eingespielt. Die Tabelle wird dann als gecacht markiert, sodass andere Tests diese Daten nur durch die Angabe von NoCache-Annotationen überschreiben können.

2.4.14.5. Globales Deaktivieren des Cache

Das Caching dient der Performance Optimierung der Testfälle. Bei der Fehlersuche kann es jedoch hinderlich sein, da man die Seiteneffekte des Caching bedenken muss. Über die Methode `setGlobalCacheDisabled` kann das Caching für alle Testklassen deaktiviert werden, die diese Konfiguration nutzen. Dies erleichtert die Analyse, ohne dass man an jede Klasse die Annotation `@NoCache` setzen muss.

```
public void configure(DbConfiguration configuration) {
  // Deaktiviert das Caching global
  // (es muss nicht aktiviert werden, da der Default false ist)
  configuration.setGlobalCacheDisabled(true);
}
```

Beispiel 2.47. Konfigurationseinstellungen zum globalen Deaktivieren des Cache

2.4.15. Analysieren mit Hilfe von Reports

Während in automatisierten Tests in der Regel genau eine Eigenschaft überprüft wird, umfasst die Überprüfung der Datenbank den gesamten Datenbestand. Der erwartete Datenbestand kann somit an mehr als einer Stelle von dem tatsächlichen Datenbestand abweichen. In automatisierten Tests wird das Verhalten so abgebildet, dass lediglich die erste erkannte Abweichung gemeldet wird. In den meisten Fällen liefert die Fehlermeldung ausreichende Informationen, um die Ursache für den fehlgeschlagenen Test zu finden. Teilweise kann diese Fehlermeldung jedoch auch irreführend sein.

Nachhaltige automatisierte Tests zeichnet eine lange Lebensdauer aus. Die Testbasis wird durch die Erstellung neuer Tests in der Entwicklungsphase stetig verbreitert. In jedem Projekt kommt früher oder später

der Moment, in dem die Änderungen eines Entwicklers einen bestehenden Test fehlschlagen lassen. Es ist die Aufgabe des Entwicklers festzustellen, ob seine Änderungen fehlerhaft sind oder ob der bestehende Test nicht mehr korrekt ist. Dabei hilft ein Blick auf das Ganze, anstatt sich nur mit der einen durch den Test gemeldeten Abweichung zu beschäftigen. Zu diesem Zweck bietet checkerberry db die Möglichkeit, verschiedene Reports zu erstellen. Im Diff-Report werden die erwarteten Testdaten mit dem tatsächlichen Datenbestand vergleichen. Der Process-Analysis-Report zeigt, wie sich nach dem Einspielen der initialen Testdaten der Datenbankbestand verändert hat. In dem Expectation-Report werden die initialen mit den erwarteten Testdaten verglichen.

2.4.15.1. Anzeigen aller Testabweichungen

Der Diff-Report vergleicht die erwarteten Testdaten mit dem tatsächlichen Datenbestand und speichert das Ergebnis in einer HTML-Datei. Für jede Datenbanktabelle wird eine Übersicht erzeugt, die unerwartete, fehlende oder abweichende Werte anzeigt.

```
public void testAnything() throws Exception {

  …
  // Testhandler holen.
  DbTestHandler testHandler = getEnvironment().getTestHandler();
  // Diff-Report erstellen.
  testHandler.createDiffReport("c:/temp/diff.html");
}
```

Beispiel 2.48. Erstellung Diff-Report

Das Code-Beispiel zeigt, wie ein Diff-Report aus der Testmethode heraus erstellt wird. Der Test-Handler stellt die Methode `createDiffReport` zur Verfügung, die den Report erstellt und in einer HTML-Datei im Dateisystem speichert.

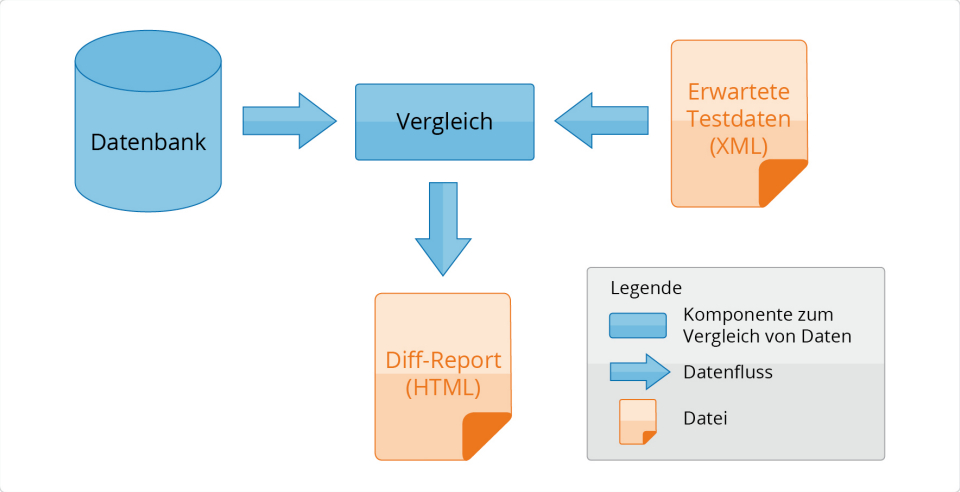


Abbildung 2.26. Erstellung Diff-Report

Abbildung 2.26, „Erstellung Diff-Report“ zeigt schematisch den Ablauf bei der Erstellung des Diff-Reports. Der Datenbankinhalt und die erwarteten Testdaten werden eingelesen und verglichen. Die Ergebnisse des Vergleichs werden im HTML-Format in einer Datei gespeichert. Die Datei kann dann über einen Browser angezeigt werden.

Wie die Ergebnisse des Diff-Reports dargestellt sind, wird in Abschnitt 2.4.15.4, „Aufbau eines Reports am Beispiel des Diff-Reports“ detailliert erläutert.

2.4.15.2. Anzeigen der tatsächlichen Datenbankänderungen

Der Process-Analysis-Report vergleicht die initialen Testdaten mit dem tatsächlichen Datenbestand und speichert das Ergebnis in einer HTML-Datei. Für jede Datenbanktabelle wird eine Übersicht erzeugt, die veränderte, hinzugefügte oder gelöschte Werte anzeigt. Auf diese Art und Weise kann somit analysiert werden, welche Datenbankänderungen durch die im Test aufgerufenen Funktionen verursacht werden.

```
public void testAnything() throws Exception {

  …
  // Testhandler holen.
  DbTestHandler testHandler = getEnvironment().getTestHandler();
  // Process-Analysis-Report erstellen.
  testHandler.createProcessAnalysisReport("c:/temp/processanalysis.html");
}
```

Beispiel 2.49. Erstellung Process-Analysis-Report

Das Code-Beispiel zeigt, wie ein Process-Analysis-Report aus der Testmethode heraus erstellt wird. Der Test-Handler stellt die Methode `createProcessAnalysisReport` zur Verfügung, die den Report erstellt und in einer HTML-Datei im Dateisystem speichert.

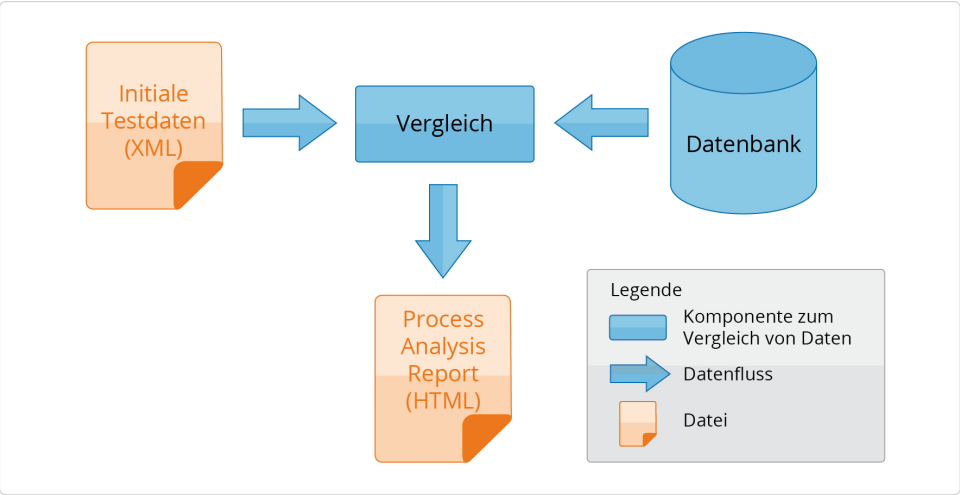


Abbildung 2.27. Erstellung Process-Analysis-Report

Abbildung 2.27, „Erstellung Process-Analysis-Report“ zeigt schematisch den Ablauf bei der Erstellung des Process-Analysis-Reports. Die initialen Testdaten und der Datenbankinhalt werden eingelesen und verglichen. Die Ergebnisse des Vergleichs werden im HTML-Format in einer Datei gespeichert. Die Datei kann dann über einen Browser angezeigt werden.

Die Darstellung der Ergebnisse wird in Abschnitt 2.4.15.4, „Aufbau eines Reports am Beispiel des Diff-Reports“ ausführlich erläutert.

2.4.15.3. Anzeigen der erwarteten Datenbankänderungen

Der Expectation-Report vergleicht die initialen mit den erwarteten Testdaten und speichert das Ergebnis in einer HTML-Datei. Der Report erzeugt eine Übersicht, welche die durch die Testausführung erwarteten Änderungen darstellt.

```
public void testAnything() throws Exception {

  …
  // Testhandler holen.
  DbTestHandler testHandler = getEnvironment().getTestHandler();
  // Expectation-Report erstellen.
  testHandler.createExpectationReport("c:/temp/expectation.html");
}
```

Beispiel 2.50. Erstellung Expectation-Report

Das Code-Beispiel zeigt, wie ein Expectation-Report aus der Testmethode heraus erstellt wird. Der Test-Handler stellt die Methode `createExpectationReport` zur Verfügung, die den Report erstellt und in einer HTML-Datei im Dateisystem speichert.

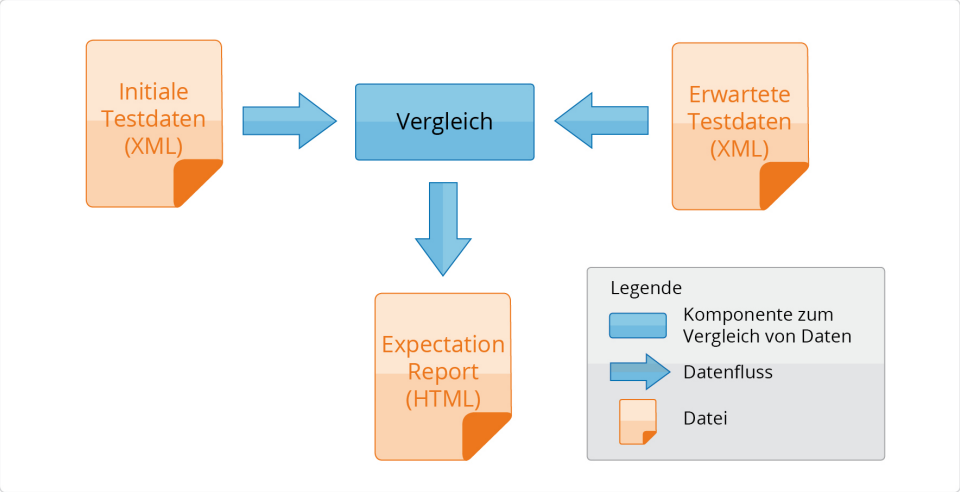


Abbildung 2.28. Erstellung Expectation-Report

Abbildung 2.28, „Erstellung Expectation-Report" zeigt schematisch den Ablauf bei der Erstellung des Expectation-Reports. Die initialen und die erwarteten Testdaten werden eingelesen und verglichen. Die Ergebnisse des Vergleichs werden im HTML-Format in einer Datei gespeichert. Die Datei kann dann über einen Browser angezeigt werden.

Die Darstellung der Ergebnisse wird in Abschnitt 2.4.15.4, „Aufbau eines Reports am Beispiel des Diff-Reports" detailliert erläutert.

2.4.15.4. Aufbau eines Reports am Beispiel des Diff-Reports

Stellvertretend für die verschiedenen Reports wird die Darstellung der Vergleichsergebnisse anhand eines Diff-Reports beschrieben. Die folgende Grafik zeigt einen möglichen Diff-Report.

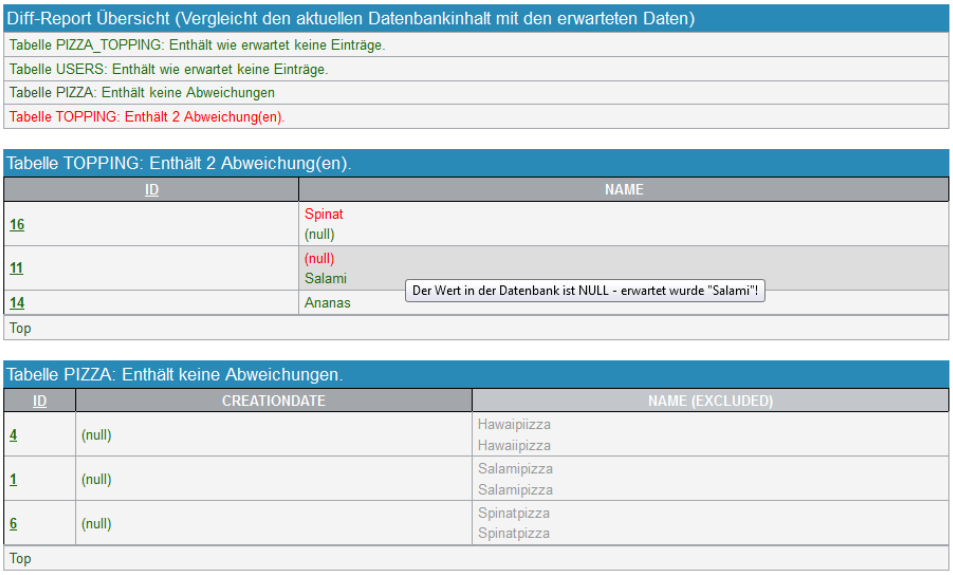


Abbildung 2.29. Diff-Report

Anhand des Diff-Reports erkennt man, dass die Datenbank aus mehreren Tabellen mit den Namen `USERS`, `PIZZA`, `TOPPING`, `PIZZA_TOPPING` besteht. Die Werte der Spalten werden in Grün dargestellt, wenn sie in den erwarteten Testdaten und in dem tatsächlichen Datenbestand übereinstimmen. In dem obigen Beispiel gibt es zwei Abweichungen in der Spalte `NAME` der Tabelle `TOPPING`. In der Zelle ist sowohl der erwartete als auch der tatsächliche Datenbankwert angegeben. Aufgrund der Abweichung ist der tatsächliche Wert Rot und der erwartete Wert Grün dargestellt. Weitere Informationen zu Abweichungen liefern Tooltipps, die angezeigt werden, wenn der Mauszeiger auf die entsprechende Zelle zeigt.

Der Diff-Report ist besonders dann hilfreich, wenn die erwarteten Testdaten viele Abweichungen zum tatsächlichen Datenbestand aufweisen. Es ist sehr unwahrscheinlich, dass der fehlschlagende Test die wirkliche Ursache für den Fehler anzeigt. Stattdessen kann der Test einen Folgefehler anzeigen, den der Entwickler nicht erwartet hat. Durch den Diff-Report wird jedoch schnell klar, wo die tatsächliche Fehlerursache zu finden ist.

Der Diff-Report ist wie folgt gegliedert: In der Kopfzeile findet sich die Diff-Report Übersicht, in der Tabellen ohne Abweichung grün und Tabellen mit Abweichung rot markiert sind. Über einen Link kann man direkt zu der angegebenen Tabelle springen.

Anschließend werden alle nicht ausgeschlossenen Tabellen aufgelistet, die Daten enthalten oder für die Daten erwartet wurden. Die Daten werden auf Zeilen- und Werte-Ebene verglichen und die Unterschiede grafisch hervorgehoben (siehe Abbildung 2.30, „Tabelle mit Abweichungen"). Jede Zeile, die sowohl in der Datenbank als auch in den erwarteten Daten vorhanden ist, wird grau hinterlegt dargestellt. Eine Zeile, die in der Datenbank vorhanden ist, aber nicht erwartet wurde, ist gelb hinterlegt. Wurde eine Zeile erwartet, die nicht in der Datenbank vorhanden ist, wird sie rot hinterlegt dargestellt. Gibt es Abweichungen in einer Zeile, die sowohl in der Datenbank als auch in den erwarteten Daten vorhanden ist, so werden beide Werte innerhalb einer Zelle angezeigt. Dabei ist der tatsächliche Wert Rot und der erwartete Wert Grün eingefärbt.

Tabelle PIZZA: Enthält 6 Abweichung(en).

ID	CREATIONDATE	NAME
4	(null)	Hawaipiizza Hawaiipizza
2	(null)	Rucolapizza
1	(null)	Salamipizza
8	(null)	Honeymoon
3	(null)	Margerita
7	(null)	Quattro Formaggi
5	(null)	Schinkenpizza
6	(null)	Spinatpizza

Diese Zeile wurde erwartet, ist aber nicht in der Datenbank vorhanden!

Top

Abbildung 2.30. Tabelle mit Abweichungen

Um die Zeilen der erwarteten und tatsächlichen Daten einander zuordnen zu können, werden Lookup-Keys benötigt (siehe Abschnitt 2.4.2, „Zuordnung von Testdaten und tatsächliche Daten"). Wenn keine Lookup-Keys für eine Tabelle definiert wurden, wird der Diff-Report für diese Tabellen trotzdem erstellt, indem alle Spalten als Lookup-Key verwendet werden. So kann der Diff-Report auch ohne die Definition von Lookup-Keys genutzt werden.

Die folgende Abbildung zeigt zwei Tabellen im Diff-Report, die ohne Lookup-Keys verglichen wurden.

Tabelle TOPPING: Enthält Einträge und soll verglichen werden, aber für diese Tabelle sind keine Lookup Keys definiert. Enthält 7 Abweichung(en).

ID	NAME
12	Rucola
16	Spinat
15	Schinken
18	Honig
14	Ananas
17	Käse
14	Annanas
11	Salami
13	Tomate

Top

Tabelle PIZZA_TOPPING: Enthält Einträge und soll verglichen werden, aber für diese Tabelle sind keine Lookup Keys definiert. Enthält keine Abweichungen.

PIZZA_ID	TOPPINGS_ID
1	16
2	14
1	11
3	12

Top

Abbildung 2.31. Diff-Report ohne Definition von Lookup-Keys

Weil der Diff-Report bei fehlenden Lookup-Keys alle Spalten als Lookup-Keys verwendet, werden geänderte Spalten zweimal aufgelistet: Als erwartete und als tatsächliche Spalte. Das ist z.B. bei ID 14 „Annanas" / „Ananas" der Fall.

Abbildung 2.32, „Tabelle nicht in der Datenbank vorhanden, obwohl sie erwartet wurde" zeigt den Fall, dass eine ganze Tabelle in der Datenbank fehlt, obwohl sie erwartet wurde.

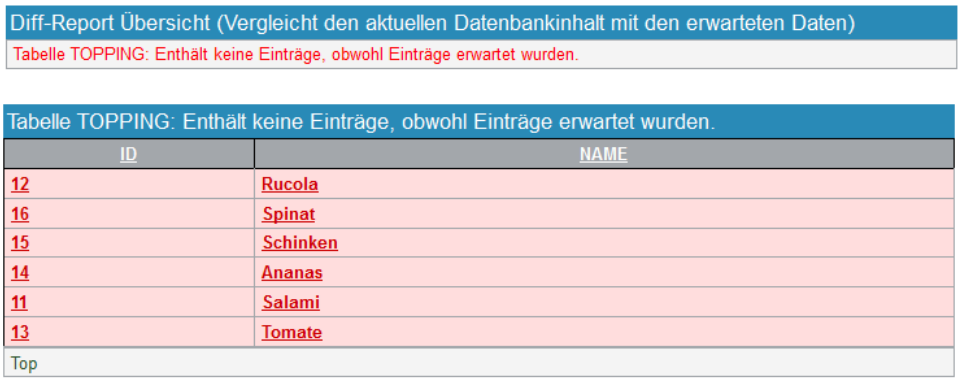


Abbildung 2.32. Tabelle nicht in der Datenbank vorhanden, obwohl sie erwartet wurde

Tabellen, die nicht in den erwarteten Daten definiert wurden, werden nicht gegen den Datenbankinhalt geprüft. Sind für eine derartige Tabelle Einträge in der Datenbank vorhanden, wird die Tabelle in der Übersicht grün markiert (siehe Abbildung 2.33, „Tabelle soll nicht verglichen werden").

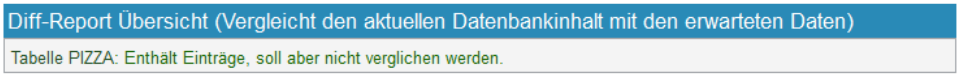


Abbildung 2.33. Tabelle soll nicht verglichen werden

Es ist möglich, in den erwarteten Daten anzugeben, dass eine Tabelle leer sein soll (siehe Abschnitt 2.4.12, „Verwenden von leeren Tabellen"). Ist das der Fall und in der Datenbank sind für diese Tabelle Daten vorhanden, dann wird die Tabelle rot markiert (siehe Abbildung 2.34, „Tabelle wurde leer erwartet").

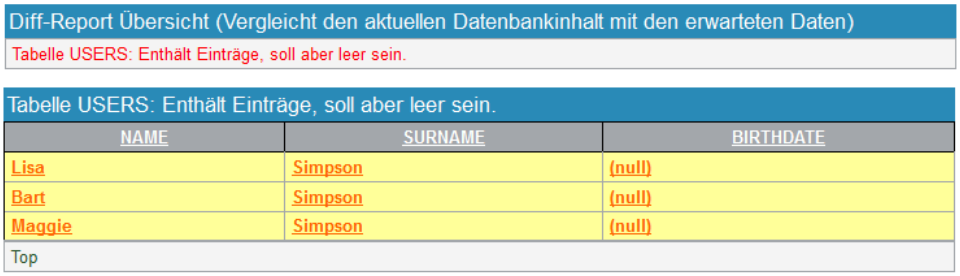


Abbildung 2.34. Tabelle wurde leer erwartet

2.4.16. Einspielen von Testdaten temporär unterdrücken

Bei der Entwicklung von Tests oder bei der Fehleranalyse tritt häufig die Situation auf, dass ein einzelner Test in der lokalen Entwicklungsumgebung mehrmals hintereinander ausgeführt wird. Dies bedeutet insbesondere, dass auch die initialen Testdaten bei jedem Durchlauf neu in die Datenbank eingespielt werden. Das erneute Einspielen der Testdaten kann jedoch überflüssig sein, wenn die zu testende Funktionalität die Testdaten nicht verändert. Aus diesem Grund bietet checkerberry db die Möglichkeit, das Laden der Testdaten zu verhindern bzw. einzuschränken.

```
// Das Einspielen der initialen Testdaten unterbinden: Die Datenbank ist
// noch in dem benötigten Zustand.
@SkipDbSetUp
public void testAnything() throws Exception {
  …
}
// Das Einspielen der cacheable Tabellen unterbinden: Der Cache enthält noch
// die benötigten Informationen.
@SkipImportCacheableTables
public void testAnythingElse() throws Exception {
  …
}
```

Beispiel 2.51. Skip-Annotationen

Das obige Beispiel zeigt die Verwendung der beiden Annotationen zur Manipulation der Testdateneinspielung. In Abhängigkeit der zu testenden Anwendung können sehr große Datenmengen in den initialen Testdaten vorhanden sein. Die Verwendung dieser Annotation kann somit das Laden von großen Datenmengen unterbinden, was die Geschwindigkeit der Testdurchführung erhöht.

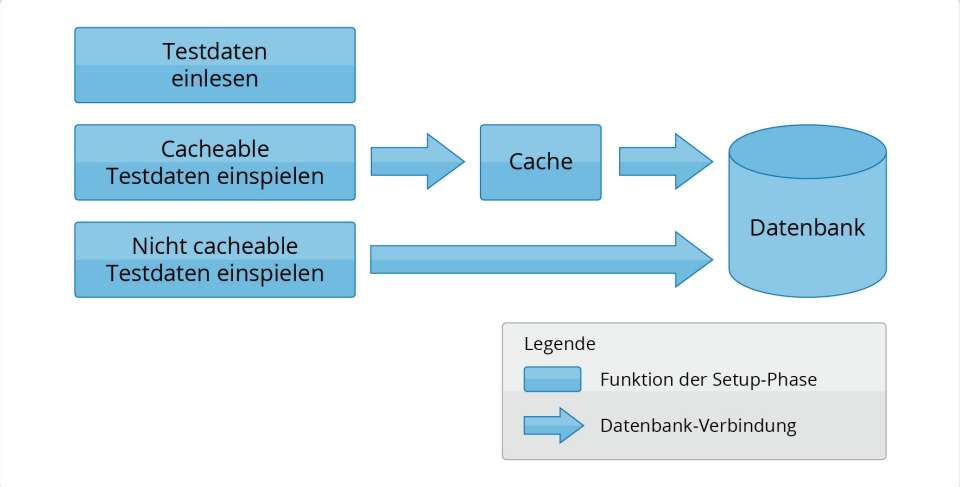


Abbildung 2.35. Setup-Phase ohne Skip-Annotationen

Die obige Abbildung skizziert das Verhalten von checkerberry db in der Setup-Phase. Zunächst werden initiale Testdaten eingelesen. Danach werden die cacheable Tabellen in die Datenbank eingespielt, sofern sie noch nicht im Cache vorhanden sind. Ist eine Tabelle im Cache vorhanden, werden die zugehörigen Daten nicht erneut in die Datenbank eingespielt. Danach werden die restlichen Testdaten in die Datenbank eingespielt.

2.4.16.1. Einspielen der initialen Testdaten unterdrücken

Durch die Markierung einer Testmethode mit der Annotation `SkipDbSetUp` wird das Einspielen der Testdaten vollständig unterdrückt. Die Verwendung ist lediglich sinnvoll, wenn die zu testende Komponente die Datenbank nicht ändert und wenn der Datenbankinhalt vor der Testausführung bereits korrekt ist.

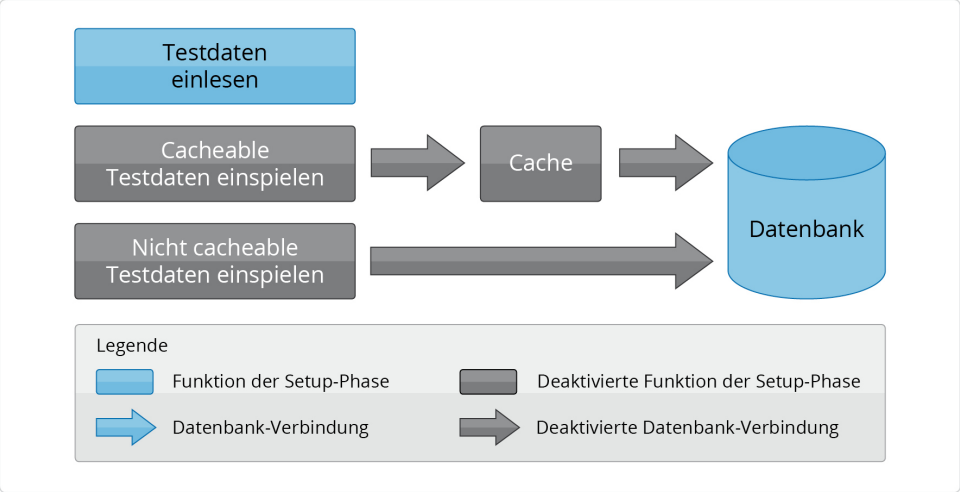


Abbildung 2.36. Setup-Phase mit SkipDbSetup

Die obige Abbildung skizziert das Verhalten von checkerberry db in der Setup-Phase, wenn die Annotation `SkipDbSetup` gesetzt ist. Sowohl das Einspielen der cacheable als auch das Einspielen aller anderen Tabellen wird unterdrückt. Der Datenbankinhalt wird somit nicht aktualisiert, obwohl initiale Testdaten vorhanden sind.

2.4.16.2. Einspielen der gecachten Testdaten unterdrücken

Durch die Markierung einer Testmethode mit der Annotation `SkipImportCacheableTables` wird das Einspielen der Testdaten eingeschränkt. Alle Tabellen der initialen Testdaten, die in den Tabellenbeschreibungen als `cacheable` markiert sind, werden nicht in die Datenbank eingespielt.

Die Verwendung ist lediglich sinnvoll, wenn die zu testende Komponente die Tabellen des Datenbank-Caches nicht ändert und wenn der Datenbank-Cache vor der Testausführung bereits korrekt ist. Details zum Datenbank-Cache sind im Kapitel Abschnitt 2.4.14, „Performance-Optimierung durch Caching von Tabellen“ beschrieben.

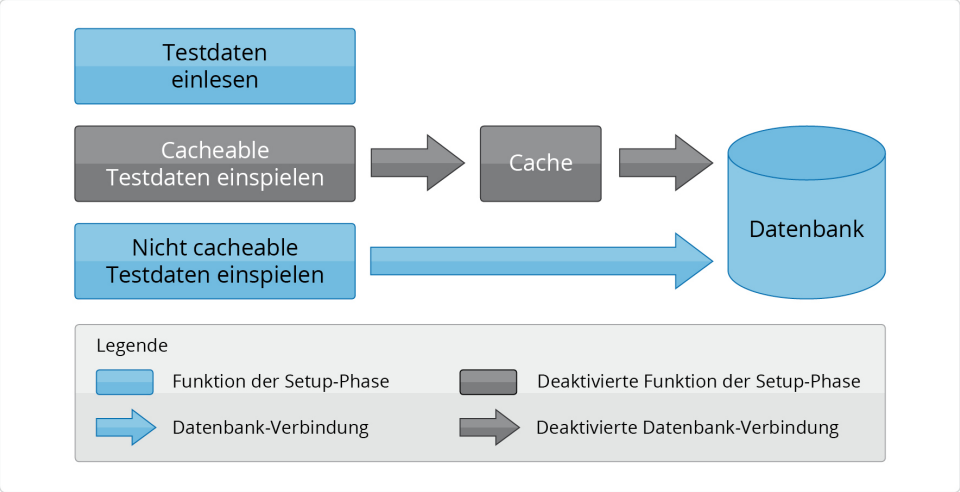


Abbildung 2.37. Setup-Phase mit SkipImportCacheableTables

Die obige Abbildung skizziert das Verhalten von checkerberry db in der Setup-Phase, wenn die Annotation `SkipImportCacheableTables` gesetzt ist. Das Einspielen der cacheable Tabellen wird vollständig unterdrückt. Die Inhalte aller anderen Tabellen werden jedoch in die Datenbank eingespielt.

2.4.17. Aktivieren des Datenbank-Loggings

Aus Performance-Gründen werden die initialen Testdaten durch checkerberry db im Batch-Modus in die Datenbank geschrieben. Beim Einspielen der Testdaten wird für jeden einzufügenden Datensatz ein SQL-INSERT-Statement an die Datenbank gesendet. Durch die Verwendung des Batch-Modus wird nicht jedes SQL-Statement separat an die Datenbank gesendet. Stattdessen werden die SQL-Statements gesammelt und zusammen an die Datenbank übertragen. Dadurch wird die Kommunikation mit der Datenbank minimiert, was insbesondere bei großen Datenmengen zu Performance-Vorteilen führt.

Die Anzahl der SQL-Statements, die zusammen an die Datenbank gesendet wird, bezeichnet man als BatchSize. Die BatchSize ist initial auf den Wert 100 gesetzt und kann über das DbUnit-Property http://www.dbunit.org/properties/batchSize konfiguriert werden [DbUnit-Properties, 2010].

Die Verwendung des Batch-Modus ist dann ein Nachteil, wenn bei dem Einspielen der initialen Testdaten ein Fehler auftritt. Beim Einspielen der Daten durch DbUnit im Batch-Modus kann nicht angegeben werden, welches Statement den Fehler verursacht hat. Aus diesem Grund bietet checkerberry db die Möglichkeit, SQL-Statements zu loggen. Auf diese Art und Weise kann der Entwickler genau nachvollziehen, wann welche Anfragen an die Datenbank gesendet wurden. Dies vereinfacht die Suche nach der Fehlerursache.

2.4.17.1. Konfigurieren des Datenbank-Loggings

Die Konfiguration des Datenbank-Loggings ist optional. In der Regel wird das Logging nur zu Analysezwecken aktiviert, sodass während der Installation keine weiteren Schritte erforderlich sind.

```
public class ConfigurationCallback implements DbConfigurationCallback {
  public void configure(DbConfiguration configuration) {
    // Datenbank-Logging aktivieren
    configuration.setStatementLoggingEnabled(true);
}
```

Beispiel 2.52. Aktivierung des Datenbank-Loggings

Das obige Code-Beispiel demonstriert, wie das Datenbank-Logging über die Methode `setStatementLoggingEnabled` aktiviert wird.

Intern verwendet checkerberry db das Datenbank-Interceptor-Werkzeug P6Spy, um die Zugriffe zur Datenbank zu loggen. Standardmäßig wird eine eingebaute Konfiguration verwendet, die die Log-Ausgabe in die Konsole schreibt. Zur Änderung der P6Spy-Konfigurationseinstellungen, kann eine eigene Datei `spy.properties` in den Klassenpfad einfügt werden. Die Konfiguration der Property-Datei ist in [P6Spy Dokumentation, 2010] beschrieben.

2.4.18. Namenskonvention der XML-Testdaten anpassen

Checkerberry db nutzt den `TestdataNameResolver`, um die Namen der Testdaten-Datei zu ermitteln. In checkerberry db wird als Standard der `DefaultTestdataNameResolver` verwendet, der den Namen der Testdaten-Datei nach dem Muster *`<Klassenname>_<Methodenname>_<suffix>.xml`* zusammenstellt. Wenn Sie die Namen ihrer Testdaten-Dateien nach einem anderen Muster aufbauen möchten, so können Sie das Interface `TestdataNameResolver` erweitern und eine Instanz dieser Klasse über `setTestdataNameResolver(TestdataNameResolver)` in der `DbConfiguration` setzen. Alternativ können Sie auch das Muster für die Erstellung der Namen anpassen.

```
public class ConfigurationCallback implements DbConfigurationCallback {
  public void configure(DbConfiguration configuration) {
    // Pattern der Testdatennamen ändern.
    configuration.setTestdataNameResolverPattern("{0}{1}/{2}.xml");
    // Präfix vor der Methode ändern (Default "_").
    configuration.setTestdataNameResolverMethodPrefix("/Method_");
    // Präfix vor dem Suffix z.B. 'initial' ändern (Default "_").
    configuration.setTestdataNameResolverSuffixPrefix("Testdata_");
    …
  }
}
```

Beispiel 2.53. Konfiguration der Testdatennamen

In obigem Beispiel wird das Muster für die Bestimmung der Testdatennamen geändert. Das Muster wird vom Default („`{0}{1}{2}.xml`") auf „`{0}{1}/{2}.xml`" geändert, wobei `{0}` den vollständigen Namen der Klasse definiert z.B. `de.conceptpeople.MyTest`, `{1}` den Namen der Testmethode definiert z.B. `testThis` und `{2}` das aktuelle Suffix definiert z.B. `initial`. Darüber hinaus werden die Präfixe für den Testmethodennamen und das Suffix geändert. Die initialen Testdaten würden in diesem Fall somit nicht mehr unter *`de/conceptpeople/MyTest_testThis_initial.xml`* sondern unter *`de/conceptpeople/MyTest/Method_testThis/Testdata_initial.xml`* gesucht werden.

2.4.19. Suffixe der XML-Testdaten anpassen

Über die Methoden *`setInitialSuffix`* und *`setExpectedSuffix`* können die Suffixe der initialen und erwarteten Testdaten geändert werden.

Beispiel 2.54, „Konfigurationseinstellungen für die Dateiendungen der Testdaten" enthält ein Beispiel für diese Konfigurationsmöglichkeit.

```
public class ConfigurationCallback implements DbConfigurationCallback {

  public void configure(DbConfiguration configuration) {
    // Suffix der initialen Testdaten von "initial" auf "ini" setzen.
    configuration.setInitialSuffix("ini");
    // Suffix der erwarteten Testdaten von "result" auf "exp" setzen.
    configuration.setExpectedSuffix("exp");
  }
}
```

Beispiel 2.54. Konfigurationseinstellungen für die Dateiendungen der Testdaten

2.4.20. Anpassen von DbUnit-Properties und Features

DbUnit bietet eine Reihe von Konfigurationsmöglichkeiten, die ausführlich in [DbUnit-Properties, 2010] beschrieben sind. Checkerberry db bietet die Möglichkeit Properties und Features von DbUnit direkt über die Methode `setProperty` zu setzen.

DbUnit verwendet Properties um beliebige Werte zu setzen, während Features als Schalter verwendet werden. Für Features sind daher lediglich die Werte `true` oder `false` erlaubt.

```
public class ConfigurationCallback implements DbConfigurationCallback {
  public void configure(DbConfiguration configuration) {

    // Setzen des Properties batchSize auf den Wert 50.
    configuration.setProperty(DatabaseConfig.PROPERTY_BATCH_SIZE,
        Integer.valueOf(50));
    // Setzen des Features dataTypeWarning auf false.
    configuration.setProperty(DatabaseConfig.FEATURE_DATATYPE_WARNING,
        false);

    // Setzen der DataTypeFactory für die entsprechende Datenbank.
    configuration.setProperty(DatabaseConfig.PROPERTY_DATATYPE_FACTORY,
        new PostgresqlDataTypeFactory());
    // Setzen der ForwardOnlyResultSetTableFactory für die Behandlung von
    // großen Datenmengen.
    configuration.setProperty(
        DatabaseConfig.PROPERTY_RESULTSET_TABLE_FACTORY,
        new ForwardOnlyResultSetTableFactory());
    …
  }
}
```

Beispiel 2.55. Setzen von DbUnit-Properties und -Features.

Das Code-Beispiel zeigt exemplarisch das Setzen eines Properties (*DatabaseConfig.PROPERTY_BATCH_SIZE*) und eines Features (*DatabaseConfig.FEATURE_DATATYPE_WARNING*) in DbUnit.

Des Weiteren werden die beiden Properties `DatabaseConfig.PROPERTY_DATATYPE_FACTORY` und `DatabaseConfig.PROPERTY_RESULTSET_TABLE_FACTORY` gesetzt. Die Klasse `org.dbunit.database.DatabaseConfig` enthält Konstanten für alle IDs der DbUnit-Properties und -Features, sodass die explizite Angabe der Property-ID nicht erforderlich ist.

Das Property `DatabaseConfig.PROPERTY_DATATYPE_FACTORY` muss gemäß der verwendeten Datenbank gesetzt werden. Aktuell werden die folgenden Datenbanken unterstützt:

- `org.dbunit.ext.db2.Db2DataTypeFactory`
- `org.dbunit.ext.h2.H2DataTypeFactory`
- `org.dbunit.ext.db2.HsqldbDataTypeFactory`
- `org.dbunit.ext.mssql.MsSqlDataTypeFactory`
- `org.dbunit.ext.mysql.MySqlDataTypeFactory`
- `org.dbunit.ext.oracle.OracleDataTypeFactory`
- `org.dbunit.ext.oracle.Oracle10DataTypeFactory`
- `org.dbunit.ext.postgresql.PostgresqlDataTypeFactory`
- `org.dbunit.ext.netezza.NetezzaDataTypeFactory`

Für das Property `DatabaseConfig.PROPERTY_RESULTSET_TABLE_FACTORY` gibt es zwei Ausprägungen: `CachedResultSetTableFactory` und `ForwardOnlyResultSetTableFactory`, wobei `CachedResultSetTableFactory` voreingestellt ist. Die `CachedResultSetTableFactory` lädt alle Daten in den Speicher, sodass diese Factory für große Datenmengen nicht geeignet ist. Dies betrifft insbesondere die Erstellung von großen Datenbank-Dumps. In diesem Fall sollte man die Factory temporär auf `ForwardOnlyResultSetTableFactory` umstellen. Zusätzlich kann es erforderlich sein, dass der Heap-Size des Tests über die JVM-Argumente `-Xms` und `-Xmx` erhöht werden muss.

2.4.21. Verwenden des BCD-Formats

BCD steht für „binary coded decimal" und bezeichnet ein binäres Datenformat. Das Format wurde aus Gründen der Platzersparnis entworfen und ist daher gerade bei Legacy-Anwendungen häufiger anzutreffen. Im BCD-Format werden Ziffern jeweils in einem Nibble (4 Bit eines Bytes) abgelegt, sodass zwei Ziffern nur

ein Byte belegen. Speichert man z.B. das Datum 31122012 im BCD-Format, benötigt man lediglich vier Byte. Jedes Byte speichert dabei 2 Ziffern. Am Beispiel der 31 bedeutet dies, dass die 3 im höheren Nibble und die 1 im niedrigen Nibble gespeichert wird. Binär sieht das dann wie folgt aus: 0011 0001. Dies wiederum entspricht der Zahl 49. Es ist zu beachten, dass Bytes in Java das Zweierkomplement verwenden. Das Byte 1001 0001 (BCD „91“) entspricht somit der Zahl -111 = -128 (1000 0000) + 17 (0001 0001) und nicht 145 = 128 (1000 0000) + 17 (0001 0001).

In der Regel ist es nicht sehr sinnvoll, binäre Vergleiche im Rahmen von Datenbanktests durchzuführen. Aus diesem Grund unterstützt DbUnit lediglich ein Binärformat: Base64. Es ist somit möglich, Binärdaten in die Datenbank einzuspielen und zu vergleichen. Bei der Verwendung von BCD-Werten müsste somit erst eine Konvertierung in das Base64-Format erfolgen, um die Daten in den Testdaten verwenden zu können. Dieses Vorgehen ist zum einen jedoch aufwändiger als direkt die BCD-Werte anzugeben. Zum anderen können die fachlichen Inhalte von BCD-Werten in der Regel einfach gelesen werden. Dies Lesbarkeit geht mit der Base64-Kodierung verloren. Aus diesem Grund unterstützt checkerberry db direkt das BCD-Format.

Für die Verwendung des BCD-Formats gibt es zwei unterschiedliche Möglichkeiten. Zum einen kann das BCD-Format einfach über eine Funktion verwendet werden. In diesem Fall werden jedoch die BCD-Daten bei einem Datenbankdump in Base64 dargestellt. Durch die Verwendung der Funktion wird somit lediglich die Lesbarkeit der BCD-Werte in den Testdaten verbessert. Zum anderen kann das interne Binärformat vollständig von Base64 auf BCD umgestellt werden. In diesem Fall werden die Binärdaten auch im Datenbankdump immer im BCD-Format dargestellt.

2.4.21.1. Verwenden von BCD über eine Funktion

Checkerberry db enthält eine Funktion zur Konvertierung von BCD-Formaten in Base64. Durch die Verwendung der Funktion in den Testdaten können die Daten im BCD-Format angegeben werden, wobei intern weiterhin Base64-Werte verwendet werden. Der Einsatz dieser Funktion ist sinnvoll, wenn die Verwendung von BCD-Formaten nur vereinzelt erforderlich ist und zusätzlich binäre Daten in Base64-Kodierung verwendet werden sollen.

Das folgende Beispiel zeigt die Verwendung der Funktion in den Testdaten:

```
<dataset>
  <USERS ID="1" NAME="Simpson" BIRTHDATE="->bcdToBase64(X'31121950')" />
</dataset>
```

Beispiel 2.56. Funktion bcdToBase64

Durch die Verwendung der Funktion bleiben die BCD-Werte in den Testdaten lesbar. Im Beispiel wird das Datum 31.12.1950 in den Testdaten verwendet. Es ist jedoch zu beachten, dass Vergleiche nach wie vor in Base64 erfolgen. Gibt es somit eine Abweichung in dem erwarteten und tatsächlichen BCD-Wert, werden in der Fehlermeldung beide Werte in Base64 dargestellt.

Die Java-Klasse `BcdToBase64Function` (in Package `de.conceptpeople.checkerberry.db.core.functions`) ist standardmäßig in checkerberry db nicht aktiviert. Um die Funktion verwenden zu können, muss sie daher zunächst in der Konfiguration registriert werden. Das folgende Code-Beispiel zeigt den erforderlichen Aufruf.

```
public class ConfigurationCallback implements DbConfigurationCallback {
  public void configure(DbConfiguration configuration) {
    configuration.registerFunction("bcdToBase64", new BcdToBase64Function());
  }
}
```

Beispiel 2.57. Funktion bcdToBase64 registrieren

Weitere Informationen zum Thema Funktionen sind in Abschnitt 2.4.9, „Dynamische Testdaten durch Funktionen" dargestellt.

2.4.21.2. Aktivieren von BCD als Standard-Binärformat

Die `bcdToBase64`-Funktion sorgt lediglich dafür, dass in den Testdaten BCD-Werte in lesbarer Form eingetragen werden können. Intern werden die BCD-Werte jedoch in Base64-Format konvertiert. Das führt dazu, dass bei dem Vergleich von Werten und bei der Erstellung von Datenbankdumps immer das Base64-Format verwendet wird. In diesen Fällen muss der Software-Entwickler somit manuell für eine Konvertierung von Base64 nach BCD sorgen. Da dies gerade bei der umfangreichen Verwendung von BCD-Werten aufwändig werden kann, besteht in checkerberry db die Möglichkeit, das interne Binärformat von Base64 auf BCD umzustellen.

Das folgende Beispiel zeigt die Angabe von BCD-Werten in den Testdaten, nach einer Umstellung auf das BCD-Format.

```
<dataset>
  <USERS ID="1" NAME="Simpson" BIRTHDATE="X'31121950'" />
</dataset>
```

Beispiel 2.58. Direkte Angabe von BCD-Werten

Das folgende Code-Beispiel zeigt die Aktivierung des BCD-Formats in der Konfiguration.

```
public class ConfigurationCallback implements DbConfigurationCallback {
  public void configure(DbConfiguration configuration) {
    configuration.setBcdBinaryFormatActive(true);
  }
}
```

Beispiel 2.59. Umstellung des Binärformats auf BCD

2.4.21.3. Aktivierung und Verwendung des binären Validators

Innerhalb von checkerberry db werden binäre Daten in eine String-Darstellung (Base64 oder BCD) konvertiert, sodass der Vergleich von erwarteten zu tatsächlichen Daten durch einen String-Validator erfolgt. Dieses Vorgehen kann dann zu Problemen führen, wenn bei einer Umstellung auf das BCD-Format auch Daten im Base64-Format vorliegen, da der Vergleich von Werten im BCD-Format mit Werten im Base64-Format zu Fehlern führt. Aus diesem Grund gibt es einen zusätzlichen Validator, der eine explizite Unterscheidung der Formate BCD und Base64 ermöglicht. Das folgende Beispiel zeigt die Verwendung des Validators in den erwarteten Testdaten.

```
<dataset>
    <!-- Verwendung des String-Validators -->
    <INCLUDE NAME="Homer" SURNAME="Simpson" BIRTHDATE="X'03071973'" />
    <INCLUDE NAME="Bart" SURNAME="Simpson" BIRTHDATE="QsO2c2UgNjQ=" />
    <!-- Explizite Verwendung des Binär-Validators -->
    <INCLUDE NAME="Marge" SURNAME="Simpson" BIRTHDATE="bcd X'25111974'" />
    <INCLUDE NAME="Lisa" SURNAME="Simpson" BIRTHDATE="base64 MzExMjIwMTE=" />
</dataset>
```

Beispiel 2.60. Verwendung des Binär-Validators in den erwarteten Testdaten

Der Binär-Validator wird für die Validierung verwendet, wenn die erwarteten Daten eines der Präfixe „bcd" oder „base64" enthält. Bei der Verwendung des Präfixes „bcd" werden die tatsächlichen Binärdaten in BCD-Format umgewandelt und mit dem erwarteten Wert verglichen. Dementsprechend werden die Binärdaten in das Base64-Format konvertiert und verglichen, wenn das Präfix „base64" in den erwarteten Testdaten angegeben wurde.

Der Binär-Validator ist standardmäßig in checkerberry db deaktiviert, da er in der Regel nicht benötigt wird. Um den Validator verwenden zu können, muss er über die Konfiguration registriert werden. Das folgende Code-Beispiel stellt die Registrierung des Validators dar.

```
public class ConfigurationCallback implements DbConfigurationCallback {
  public void configure(DbConfiguration configuration) {
    configuration.register(new OperatorContainingBinaryValidator());
  }
}
```

Beispiel 2.61. Registrieren des Binär-Validators

Weitere Informationen zum Thema Validatoren sind in Abschnitt 2.4.3, „Überprüfen der Ergebnisse durch Validatoren“ dargestellt.

2.5. Installation

Dieses Kapitel beschreibt die Einbindung von checkerberry db in die Umgebung des Kunden. Die Installation besteht aus den folgenden Schritten

1. Einbinden der erforderlichen Bibliotheken
2. Implementierung der konkreten checkerberry db-Bridge
3. Erzeugen der checkerberry db-Umgebung.

Die Implementierung der checkerberry db-Bridge ist abhängig von der Kundenumgebung und nimmt bei der Installation den größten Aufwand in Anspruch. Dieser Aufwand muss jedoch nur einmalig betrieben werden. Nach der erfolgreichen Integration von checkerberry db werden die Ergebnisse in der Versionsverwaltung gespeichert und stehen dann jedem Entwickler zur Verfügung.

Die nächsten Kapitel beschreiben die erforderlichen Schritte in der Reihenfolge ihrer Durchführung.

2.5.1. Einbinden der erforderlichen Bibliotheken

Dieses Kapitel beschreibt, wie checkerberry db in ein konkretes Projekt eingebunden werden kann. Abschnitt 2.5.1.1 behandelt die Einbindung über Maven und die Abschnitte 2.5.1.2 und ??? das direkte Einbinden der Bibliotheken.

2.5.1.1. Einbinden der Bibliotheken über Maven

Das Einbinden der Bibliotheken über Maven ist deutlich einfacher als das direkte Einbinden der Bibliotheken. Maven verwaltet die Abhängigkeiten der verschiedenen Bibliotheken und lädt fehlende Bibliotheken bei Bedarf nach.

Zur Einbindung von checkerberry db in ein Maven-Projekt müssen die Abhängigkeiten in der Maven-Konfiguration (*pom.xml*) eingetragen werden. Folgendes Code-Beispiel zeigt die erforderlichen Einträge:

```
<dependencies>
  …
    <dependency>
      <groupId>de.conceptpeople.checkerberry</groupId>
      <artifactId>checkerberry-db</artifactId>
      <version>3.2.x</version>
      <scope>test</scope>
    </dependency>
  …
<dependencies>
```

Beispiel 2.62. Erforderliche Maven-Abhängigkeiten von checkerberry db

Durch die Angabe von checkerberry db werden ebenfalls alle abhängigen Bibliotheken in das Projekt eingebunden. Der Scope *test* besagt, dass die Bibliothek und alle abhängigen Bibliotheken nur für die Testdurchführung benötigt werden.

Bei der Verwendung von Spring, Hibernate und ggf. JPA kann die Abhängigkeit zur konkreten Bridge-Implementierung eingefügt werden. Alternativ kann eine eigene Bridge-Implementierung eingefügt werden. Das folgende Code-Beispiel zeigt wie die Abhängigkeiten in der *pom.xml* eingetragen werden müssen.

```
<dependencies>
…
  <dependency>
    <groupId>de.conceptpeople.checkerberry</groupId>
    <!-- Verwende entweder die Spring-/Hibernate-Bridge… -->
    <artifactId>checkerberry-db-spring3-bridge</artifactId>
    <!-- …oder die Spring/JPA/Hibernate-Bridge. -->
    <artifactId>checkerberry-db-spring-jpa-bridge</artifactId>
    <version>3.2.x</version>
    <scope>test</scope>
  </dependency>
…
<dependencies>
```

Beispiel 2.63. Optionale Maven-Abhängigkeiten auf die Basis-Bridge-Implementierung

Die Maven-Artefakte des checkerberry test center können über das öffentliche checkerberry Maven-Repository geladen werden (siehe Abschnitt 1.4, „Checkerberry Maven-Repository").

Damit ist die Einbindung von checkerberry db beendet. Der nächste Schritt zum Aufsetzen einer lauffähigen checkerberry db-Umgebung ist das Implementieren der checkerberry db-Bridge, welches in Abschnitt 2.5.2, „Implementierung der konkreten checkerberry-db-Bridge" beschrieben wird. Der Rest dieses Kapitels beschäftigt sich mit der direkten Einbindung der Bibliotheken.

2.5.1.2. Checkerberry db-Bibliotheken

Die folgende Tabelle enthält alle checkerberry db-Bibliotheken.

Tabelle 2.6. Bibliotheken von checkerberry db

checkerberry-db-common-3.2.x.jar	Allgemeine Klassen für die verschiedenen Projekte von checkerberry db
checkerberry-db-3.2.x.jar	Kern-Komponenten von checkerberry db.
checkerberry-db-bridge-3.2.x.jar	Schnittstelle der checkerberry-db-Bridge
checkerberry-db-spring3-bridge-3.2.x.jar	Optionale Basis-Implementation der Spring/ Hibernate-Bridge.
checkerberry-db-spring4-bridge-3.2.x.jar	Optionale Basis-Implementation der Spring/ Hibernate-Bridge ab Spring-Version 4.
checkerberry-db-spring-jpa-bridge-3.2.x.jar	Optionale Basis-Implementation der Spring/JPA/ Hibernate-Bridge.
checkerberry-main-3.2.x.pom	Enthält als Eltern-Projekt aller checkerberry-Projekte wichtige Maven-Einstellungen.
checkerberry-common-3.2.x.jar	Allgemeine Klassen für alle checkerberry-Projekte
checkerberry-test-connector-3.2.x.jar	Schnittstelle für die Abstraktion des zu verwendendes Testframeworks.

checkerberry-test-connector-switch-3.2.x.jar	Ermittelt das aktuelle Testframework und bindet die konkrete Instanz des Test-Connectors ein.
checkerberry-junit3-connector-3.2.x.jar	Klassen für die Verwendung von JUnit3. Diese Bibliothek wird automatisch von dem Test-Connector-Switch eingebunden.
checkerberry-junit4-connector-3.2.x.jar	Klassen für die Verwendung von JUnit4. Diese Bibliothek wird automatisch von dem Test-Connector-Switch eingebunden.
checkerberry-testng-connector-3.2.x.jar	Klassen für die Verwendung von TestNG. Diese Bibliothek wird automatisch von dem Test-Connector-Switch eingebunden.
checkerberry-spring-integration-3.2.x.jar	Klassen für die vereinfachte Verwendung des Springframeworks.
checkerberry-spring4-integration-3.2.x.jar	Klassen für die vereinfachte Verwendung des Springframeworks ab Spring-Version 4.

Zur Verwendung von checkerberry db sind folgende Bibliotheken zwingend erforderlich: `checkerberry-common-3.2.x.jar`, `checkerberry-test-connector-3.2.x.jar`, `checkerberry-test-connector-switch-3.2.x.jar`, `checkerberry-db-common-3.2.x.jar`, `checkerberry-db-bridge-3.2.x.jar`, `checkerberry-db-3.2.x.jar`, `checkerberry-junit3-connector-3.2.x.jar`, `checkerberry-junit4-connector-3.2.x.jar` und `checkerberry-testng-connector-3.2.x.jar`.

Bei der Verwendung von Spring und Hibernate kann die konkrete Bridge-Implementierung `checkerberry-db-hibernate-bridge-3.2.x.jar` eingefügt werden. Wird zusätzlich die JPA verwendet, kann die JAR-Datei `checkerberry-db-jpa-bridge-3.2.x.jar` eingefügt werden. Alternativ kann eine eigene Bridge-Implementierung eingefügt werden.

2.5.2. Implementierung der konkreten checkerberry-db-Bridge

Nach dem Einbinden der benötigten Bibliotheken erfolgt die Implementierung der konkreten checkerberry-db-Bridge. Die Bridge bildet die Schnittstelle zwischen dem Kern von checkerberry db und der System-Umgebung des Kunden.

Die checkerberry db-Bridge besteht aus vier Java-Interfaces, von denen zwei implementiert werden müssen und von denen zwei optional sind:

1. *`DatabaseConnector:`* (optional) Kapselt den Zugriff auf die Datenbank (siehe Abschnitt 2.5.2.1)
2. *`DatabaseDescriptionCallback:`* (must) Definiert die Beschreibung für die einzelnen Datenbanktabellen (siehe Abschnitt 2.5.2.2)
3. *`DbConfigurationCallback:`* (optional) Konfiguriert checkerberry db (siehe Abschnitt 2.5.2.3)
4. *`ClasspathResourceLoader:`* (must) Lädt Testdaten aus dem Klassenpfad. (siehe Abschnitt 2.5.2.4)

Durch die Implementierung dieser Interfaces wird die Verbindung zwischen checkerberry db und der Kundenumgebung hergestellt. Die Implementierung des Interfaces `ClasspathResourceLoader` ist dabei optional und ist in der Regel nicht erforderlich. Die Implementierung des Interfaces `DatabaseConnector` ist ebenfalls optional, wobei eine eigene Implmentierung in der Regel sinnvoll ist. Gerade zum Einstieg kann es jedoch hilfreich sein, die Standard-Implementation zu verwenden und über die Konfiguration mit den erforderlichen Datenbankparametern zu versorgen.

Im Folgenden werden exemplarische Implementationen der Interfaces in einem Spring/Hibernate-Umfeld dargestellt.

2.5.2.1. Anbinden der Datenbank durch den DatabaseConnector

Wird bei der Erzeugung der checkerberry db-Bridge kein `DatabaseConnector` explizit angegeben, wird die Standard-Implementierung des `DatabaseConnectors` verwendet. Dieser erzeugt eine JDBC-Verbindung anhand der Datenbankparameter, die über die Konfiguration angegeben werden. In Abschnitt 2.5.2.3, „Konfiguration von checkerberry db mit dem DbConfigurationCallback“ ist ein entsprechendes Beispiel dargestellt.

Der Nachteil der Standard-Implementierung besteht darin, dass die JDBC-Parameter (JDBC-URL, User, Passwort, Datenbanktreiber) in der Konfiguration `DbConfiguration` angegeben werden müssen. Das ist für den Einstieg ein schneller Weg. In einer produktiven Umgebung mit mehreren Entwicklern kann das jedoch hinderlich sein, da jeder Entwickler sein eigenes Datenbankschema verwendet. Da alle die gleiche `DbConfiguration` bzw. das gleiche `DbConfigurationCallback` verwenden, müssten die JDBC-Parameter z.B. aus einer Property-Datei eingelesen werden. Diese Lösung ist möglich und in einigen Umgebungen vielleicht auch sinnvoll. In der Regel werden jedoch andere Frameworks wie z.B. das Spring Framework eingesetzt, die eine einfachere Integration von checkerberry ermöglichen. In einer Spring/Hibernate-Umgebung steht im Projekt *`checkerberry-db-spring3-bridge`* beispielsweise die Basisklasse *`SpringHibernateDatabaseConnector`* für diese Zwecke zur Verfügung.

Im Folgenden sei kurz das `DatabaseConnector`-Interface dargestellt.

```
/**
 * Connector, der den Zugriff zur Datenbank kapselt. Auf diese Art und
 * Weise wird checkerberry db von den unterschiedlichen
 * Technologien zum Datenbankzugriff entkoppelt.
 */
public interface DatabaseConnector {

  /**
   * Liefert den Namen des zu verwendenden Datenbankschemas zurück.
   *
   * @return zu verwendendes Schema.
   */
  String getDatabaseSchema();

  /**
   * Liefert die DataSource zum Zugriff auf die Datenbank zurück.
   *
   * @return DataSource.
   */
  DataSource getDataSource();

  /**
   * Erzeugt ein neues Datenbankschema. Diese Methode ist in der Regel
   * nur wichtig, falls eine Speicherbasierte Datenbank verwendet
   * wird.
   */
  void createDatabaseSchema();

  /**
   * Gibt die URL der Datenbankverbindung zurück, die z.B. für
   * die Zugriffskontrolle benötigt wird.
   *
   * @return URL der Datenbankverbindung
   */
  String getUrl();

}
```

Beispiel 2.64. Interface DatabaseConnector

Das folgende Code-Beispiel zeigt eine exemplarische Implementierung unter Verwendung der erwähnten Basisklasse. Im Konstruktor werden dieser die *`LocalSessionFactoryBean`* und die *`DataSource`*

übergeben. Die Session-Factory wird verwendet, um generelle Informationen zur Datenbank zu ermitteln, während die Data-Source die Verbindung zur Datenbank darstellt.

```
import javax.sql.DataSource;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.orm.hibernate3.LocalSessionFactoryBean;

import de.conceptpeople.checkerberry.db.spring.bridge.hibernate.database.\\
SpringHibernateDatabaseConnector;

  /**
   * Zugang zur Datenbank.
   */
  public class SampleDatabaseConnector extends
    SpringHibernateDatabaseConnector {

  /**
   * Erzeugt einen neuen Datenbank-Connector.
   *
   * @param beanFactory
   *            Spring-Beanfactory zum Auslesen der relevanten Beans.
   */
  public SampleDatabaseConnector(BeanFactory beanFactory) {
    super(
      (LocalSessionFactoryBean) beanFactory.getBean("&sessFactory"),
        (DataSource) beanFactory.getBean("dataSource"));
  }
}
```

Beispiel 2.65. Beispiel-Implementierung des DatabaseConnectors (Spring/Hibernate-Umgebung)

Anmerkung: Zu diesem Beispiel sei erwähnt, dass das „*&*"-Zeichen vor *sessFactory* für die korrekte Referenzierung der SessionFactory notwendig ist und dazu dient, die Factory selbst und nicht eine durch sie erzeugte Session zu erhalten.

Die wesentliche Aufgabe des Database-Connectors besteht in der Bereitstellung einer JDBC-Verbindung zur Kommunikation mit der Datenbank. Die Implementierung dieses Interfaces ist somit sehr schnell und einfach auch in Umgebungen möglich, die nicht Spring und Hibernate verwenden.

In dem dargestellten Beispiel wird die `DataSource` zur Anbindung der Datenbank aus dem Application-Kontext von Spring eingelesen. Dieses Vorgehen hat den Vorteil, dass die `DataSource` bereits voll konfiguriert ist. Der Anwender muss sich somit nicht mehr darum kümmern, welche Datenbank mit welchem Benutzer verwendet wird, da dies bereits über Spring gekapselt ist. Auf diese Art und Weise kann sichergestellt werden, dass sowohl die zu testende Anwendung als auch checkerberry db auf die gleiche Datenbank zugreifen. Aus diesem Grund ist es sinnvoll, den `DatabaseConnector` individuell zu implementieren. Für die Einführung oder Evaluierung des checkerberry test centers kann es jedoch sinnvoller sein, den Standard-DatabaseConnector zu verwenden.

2.5.2.2. Definition der Datenbankstruktur im DatabaseDescriptionCallback

Checkerberry db vergleicht XML-Testdaten mit Daten aus der Datenbank. Aus diesem Grund ist es erforderlich, dass checkerberry db die Struktur (Tabellen und Spalten) der verwendeten Datenbank kennt. Die Definition der erforderlichen Informationen erfolgt über eine Implementation des Interfaces `DatabaseDescriptionCallback`. Folgendes Code-Beispiel enthält das Interface.

```
/**
 * Callback-Interface zur Definition der Datenbankbeschreibung.
 */
public interface DatabaseDescriptionCallback {

  /**
   * Erzeugt die Datenbankbeschreibung, indem für alle Tabellen die Lookup Key
   * Spalten definiert werden.
   *
   * @param databaseDescription
   *            zu initialisierende Datenbankbeschreibung.
   * @see DatabaseDescription#addTableDescription(String, String...)
   */
  void createDatabaseDescription(DatabaseDescription databaseDescription);
}
```

Beispiel 2.66. Interface DatabaseDescriptionCallback

Da die Datenbankstruktur für jede System-Umgebung unterschiedlich ist, gibt es keine Basisklasse für die Implementierung des Interfaces. Das folgende Code-Beispiel zeigt eine exemplarische Implementation.

```
/**
 * Spezielles Callback zur Definition der Datenbank-Struktur.
 */
public class SampleDatabaseDescriptionCallback
    implements DatabaseDescriptionCallback {

  /**
   * {@inheritDoc}
   */
  public void createDatabaseDescription(
      DatabaseDescription databaseDescription) {

    // Tabelle USERS hat die Spalten NAME und SURNAME als Lookup-Keys.
    databaseDescription.addTableDescription("USERS", "NAME", "SURNAME");
    // Tabelle COUNTRIES hat die Spalte CODE als Lookup-Key. Die Tabelle
    // kann im Cache gespeichert warden.
    databaseDescription.addTableDescription("COUNTRIES", Cacheable.Yes,"CODE");
    // Tabelle NOTES hat die Spalte ID als Lookup-Key. Die
    // Spalte UPDATE_DATE wird bei einem Vergleich nicht berücksichtigt.
    databaseDescription.addTableDescription("NOTES", "ID")
        .addExcludedColumns("UPDATE_DATE");
  }
}
```

Beispiel 2.67. Beispiel-Implementierung des DatabaseDescriptionCallbacks

Dem Interface `DatabaseDescriptionCallback` wird die Klasse *`DatabaseDescription`* übergeben, um dort alle Informationen zur Datenbankstruktur zu speichern. Das Interface ist als Callback realisiert, um den Erzeugungszeitpunkt der Datenbankbeschreibung in checkerberry db verwalten zu können. Jede Test-Methode verwendet eine eigene Instanz der Datenbankbeschreibung, sodass das Callback vor der Durchführung jeder Testmethode aufgerufen wird. Innerhalb des Callbacks werden Informationen zu allen Datenbanktabellen angegeben, deren Inhalt durch checkerberry db geprüft werden soll. Die wichtigste Angabe ist die Definition der Lookup-Keys für jede Tabelle. Anhand der Lookup-Keys kann checkerberry db eine Zuordnung zwischen Testdaten und den Daten der Datenbank herstellen.

Neben den Lookup-Keys kann für jede Datenbanktabelle eine Liste von Spaltennamen angegeben werden, die bei dem Vergleich von Testdaten mit den Daten der Datenbank nicht berücksichtigt werden. Es handelt sich dabei in der Regel um dynamische und/oder technische Informationen wie z.B. technische Zeitstempel, die den Änderungszeitpunkt eines Datensatzes anzeigen.

Des Weiteren ist die Markierung einer Datenbanktabelle als `cacheable` möglich. Im Gegensatz zu den Lookup Keys und den „Excluded Columns" kann die Caching-Eigenschaft nur in dem Callback-Interface gesteuert werden. Die Modifikation in einer Testmethode ist für die Caching-Eigenschaft nicht möglich. Da die

Caching-Eigenschaft bereits in der Setup-Phase, also vor der Durchführung einer Testmethode ausgewertet wird, hat die Modifikation in der Testmethode keine Auswirkungen auf die aktuelle Testmethode. Aus diesem Grund wird die Angabe der Caching-Eigenschaft nur in dem Callback-Interface erlaubt.

Es ist im Übrigen nicht erforderlich, sofort bei der Einbindung von checkerberry alle Tabellenbeschreibungen einzutragen. Es ist ausreichend, wenn man die Beschreibungen für die Tabellen definiert, die tatsächlich getestet werden. In der Praxis sieht es so aus, dass das Interface nach und nach implementiert wird.

2.5.2.3. Konfiguration von checkerberry db mit dem DbConfigurationCallback

Über das Interface *DbConfigurationCallback* erfolgt die Konfiguration von checkerberry db. Das folgende Code-Beispiel enthält das Interface.

```
/**
 * Callback zum Setzen der Konfigurationen für checkerberry db und
 * DbUnit. Die Konfiguration wird bei der Erzeugung einer neuen
 * checkerberry db-Umgebung angelegt und ändert sich dann zur Laufzeit nicht
 * mehr.
 */
public interface DbConfigurationCallback {

  /**
   * Callback zum Setzen von Konfigurationseinstellungen.
   *
   * @param configuration
   *            zu bearbeitende Konfiguration.
   */
  void configure(DbConfiguration configuration);

}
```

Beispiel 2.68. Interface DbConfigurationCallback

Die Implementation des Callbacks wird nur einmalig bei der Initialisierung von checkerberry db aufgerufen. Durch die Verwendung des Callbacks kann der Zeitpunkt der Erzeugung der Konfiguration in checkerberry db gesteuert werden.

Die Implementierung des Callback-Interfaces ist abhängig von der aktuellen System-Umgebung, sodass keine Basisklassen verwendet werden können. Die Implementierung ist jedoch sehr einfach, wie im nächsten Code-Beispiel zu erkennen ist.

```
/**
 * Beispiel-Implementierung eines Callbacks zur Manipulation der
 * Konfiguration.
 */
public class SampleConfigurationCallback implements
  DbConfigurationCallback {

  @Override
  public void configure(DbConfiguration configuration) {
    // Hiermit werden DTD-Id und ihr Ort spezifiziert. Die Angabe
    // der DTD ist zwingend erforderlich.
    configuration.setDatabaseDtd(
      "-//ConceptPeople//DTD sample-db 1.0//EN",
      "de/conceptpeople/example/sample-db.dtd");

    // Zugriff für jede JDBC-URL verbieten…
    configuration.addToAccessControlBlacklist("*");

    // …und nur hsqldb:mem erlauben.
    configuration.addToAccessControlWhitelist("*hsqldb:mem*");
  }
}
```

Beispiel 2.69. Beispiel-Implementierung des DbConfigurationCallbacks

In dem Code-Beispiel ist zu erkennen, dass dem Callback die Klasse *DbConfiguration* übergeben wird, sodass die Einstellungen der Konfigurationen angepasst werden können. Die Konfigurationsmöglichkeiten werden im Rahmen der einzelnen Features detailliert dargestellt.

Standard-DatabaseConnector

Bei der Verwendung der Standard-Implementation des `DatabaseConnectors` werden die zu verwendenden Datenbankparameter über die Konfiguration angegeben. Folgendes Beispiel zeigt eine mögliche Konfiguration.

```
public class ConfigurationCallback implements DbConfigurationCallback {

  @Override
  public void configure(DbConfiguration configuration) {
    // Datenbanktreiber festlegen
    configuration.setJdbcDriverClassName("com.mysql.jdbc.Driver");
    // JDBC-URL festlegen
    configuration.setJdbcUrl("jdbc:mysql://localhost:3306/testing");
    // Namen des Datenbankbenutzers setzen
    configuration.setJdbcUserName("homer");
    // Passwort des Datenbankbenutzers setzen
    configuration.setJdbcPassword("duff");
    // Ggf. Datenbankschema angeben
    configuration.setJdbcDatabaseSchema("");
  }
}
```

Beispiel 2.70. Datenbankparameter bei Verwendung des Standard-DatabaseConnectors

Die Verwendung der Standard-Implementation ist sehr schnell und einfach. Da die zu testende Anwendung die gleichen Datenbankparameter wie checkerberry db verwenden muss, besteht jedoch die Gefahr einer redundanten Pflege der Parameter. Aus diesem Grund ist es in der Regel sinnvoller, eine bereits konfigurierte `DataSource` zu verwenden. Für den Einstieg in checkerberry ist die Verwendung der Standard-Implementation jedoch sinnvoll.

2.5.2.4. Laden der Testdaten durch den ClasspathResourceLoader

Bei der Durchführung von Tests mit checkerberry db werden die XML-Testdaten aus dem Klassenpfad geladen. Das Laden der XML-Testdaten aus dem Klassenpfad erfolgt über die Implementierung des Interfaces *ClasspathResourceLoader*. Das Interface ist im folgenden Code-Beispiel dargestellt.

```
/**
 * Resource-Loader, der die zu ladenden Dateien im Classpath sucht.
 */
public interface ClasspathResourceLoader {

  /**
   * Liefert die URL zum angegebenen Dateinamen zurück.
   *
   * @param filename
   *            Dateiname relativ zum Classpath.
   * @return URL der Datei aus dem Classpath oder <code>null</code>, wenn
   *            eine solche Datei nicht existiert.
   * @throws java.io.IOException
   *            wenn ein Fehler beim Zugriff auf die Datei aufgetreten
   *            ist.
   */
  URL getUrl(String filename) throws IOException;
}
```

Beispiel 2.71. Interface ClasspathResourceLoader

Dem Resource-Loader wird ein relativer Dateiname übergeben. Die Datei wird dann im Klassenpfad gesucht und als URL zurückgeliefert.

Wenn keine spezifische Implementierung von `ClasspathResourceLoader` angegeben wurde, wird der `CoreClassLoader` als Default verwendet.

In Spring-Umgebungen existiert in dem Projekt *checkerberry-db-spring3-bridge* mit der Klasse *SpringClasspathResourceLoader* eine weitere Basisimplementierung dieses Interfaces. Die Klasse implementiert das Spring-Interface *org.springframework.context.ResourceLoaderAware*, was zum Setzen eines Spring-Resource-Loaders (*org.springframework.core.io.ResourceLoader*) führt. Über diesen Resource-Loader wird die gesuchte Datei geladen.

Damit der Spring-Resource-Loader durch Spring in die Klasse `SpringClasspathResourceLoader` injiziert wird, muss die Klasse über Spring erzeugt werden. Anderenfalls wird das Interface `ResourceLoaderAware` nicht ausgewertet und der Resource-Loader wird nicht gesetzt. Es ist somit wichtig, dass der `SpringClasspathResourceLoader`, wie im folgenden Code-Beispiel gezeigt, in dem ApplicationContext von Spring definiert wird.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
…
  <!-
    Resourceloader für Testdaten im Classpath: Immer easy, wenn Spring
    im Spiel ist.
  -->
  <bean name="resourceLoader" class="de.conceptpeople.checkerberry.db.\\
    spring.bridge.hibernate.resource.SpringClasspathResourceLoader"/>
…
</beans>
```

Beispiel 2.72. SpringClasspathResourceLoader in ApplicationContext.xml

2.5.3. Erzeugen der checkerberry db-Umgebung

Für die Entwicklung und Durchführung der automatisierten Integrationstests ist die Erzeugung einer checkerberry db-Umgebung erforderlich. Die checkerberry db-Umgebung steuert das Einspielen der Testdaten und stellt dem Entwickler die erforderlichen Schnittstellen zur Verfügung.

Die checkerberry db-Umgebung wird in der Setup-Phase erzeugt und lokal in der Testklasse gespeichert. Um diesen Schritt nicht für jeden Test zu wiederholen, sollte er in einer Oberklasse des Tests erfolgen. Über eine Getter-Methode wird den abgeleiteten Tests die Umgebung zur Verfügung gestellt.

```
public void setUp() {

  // Erstellen des Database-Connectors für die checkerberry db-Bridge
  SampleDatabaseConnector connector = new SampleDatabaseConnector();

  // Callback für die Datenbankbeschreibung erstellen.
  SampleDatabaseDescriptionCallback databaseDescriptionCallback
    = new SampleDatabaseDescriptionCallback();

  // Callback für die Konfiguration von checkerberry db erstellen.
  SampleConfigurationCallback configurationCallback
    = new SampleConfigurationCallback();

  // Checkerberry db-Umgebung mit den Bridge-Komponenten erzeugen. Die
  // Member-Variable environment wird allen erbenden Klassen über den Getter
  // getEnvironment() zur Verfügung gestellt.
  environment = CheckerberryDb.getInstance().getEnvironment(
    connector, databaseDescriptionCallback, configurationCallback);

  // Checkerberry db-Umgebung für den Test initialisieren.
  // Die Parameter sind abhängig von dem verwendeten Testframework
  environment.setUp(...);
}
```

Beispiel 2.73. Erzeugen der checkerberry db-Umgebung

Für die Erzeugung der Umgebung sind, wie in den vorherigen Kapiteln beschrieben, einige anwendungsspezifische Informationen erforderlich. Dies umfasst einen optionalen `DatabaseConnector` für die Anbindung von checkerberry db an die zu verwendende Datenbank. Zusätzlich wird ein `DatabaseDescriptionCallback` für die Datenbankbeschreibung benötigt. Für die Konfiguration von checkerberry db wird ein `DbConfigurationCallback` verwendet.

Nach der Erzeugung der benötigten Klassen wird die checkerberry db-Umgebung über das Singleton `CheckerberryDb` erzeugt. Danach wird die Methode `setUp` der checkerberry db-Umgebung aufgerufen, um initiale Testdaten zu ermitteln und einzuspielen. Der Methode muss zumindest die aktuelle Instanz der Testklasse übergeben werden. Ob weitere Parameter für den Aufruf der `setUp`-Methode erforderlich sind, hängt von dem verwendeten Testframework ab. In Abschnitt 2.5.3.1, „Aufrufen der setUp-Methode“ wird die `setUp`-Methode für die unterschiedlichen Testframeworks beschrieben.

```
public interface      {
  void setUp(Object testInstance);
  void setUp(Object testInstance, Method testMethod);
  void setUp(Object testInstance, String testMethodName);
  …
}
```

Beispiel 2.74. Setup-Methoden des checkerberry db-Environments

Das Interface `CheckerberryDbEnvironment` stellt drei Setup-Methoden zur Verfügung. Die Setup-Methoden verwenden immer die aktuelle Instanz der Testklasse. Darüber hinaus kann auch der Name der aktuellen Testmethode angegeben werden. Dies ist z.B. für die Verwendung von TestNG und anderen Testframeworks relevant.

Die Erzeugung der checkerberry db-Umgebung erfolgt über das Singleton `CheckerberryDb`. Der Grund hierfür besteht darin, dass alle Tests auf die gleiche checkerberry db-Umgebung zugreifen müssen. Aus diesem Grund verwaltet das Singleton-Objekt die Referenz auf die checkerberry db-Umgebung und liefert die gleiche Objektinstanz an alle Tests zurück.

Die checkerberry db-Umgebung selbst darf kein Singleton sein, da die Initialisierung von mehreren checkerberry db-Umgebungen in einem Test möglich ist, wenn mehr als eine Datenbank für die Durchführung der Tests benötigt wird. Aus diesem Grund wird an dieser Stelle das beschriebene Service Locator Pattern [Service Locator Pattern, 2010] verwendet.

In der Teardown-Phase wird die Methode `tearDown` der checkerberry db-Umgebung aufgerufen. Das folgende Code-Beispiel zeigt eine typische Implementierung.

```
public void tearDown() {

  // Checkerberry db-Umgebung über tearDown informieren.
  environment.tearDown();
  // tearDown an Basisklasse aufrufen.
  super.tearDown();
  // Sicherstellen, dass das Environment nach Testende aufgeräumt werden kann
  environment = null;
}
```

Beispiel 2.75. Beenden der checkerberry db-Umgebung nach einem Test

2.5.3.1. Aufrufen der setUp-Methode

Checkerberry db verwendet zahlreiche Konventionen, um den Konfigurationsaufwand gering zu halten. Die Konventionen greifen dabei regelmäßig auf den Namen der aktuellen Testklasse und den Namen der aktuellen Testmethode zurück. Abhängig vom verwendeten Testframework gibt es unterschiedliche Methoden, auf diese Information zuzugreifen. Während das checkerberry test center in der Lage ist, sie in JUnit3-Umgebungen komplett selbstständig zu ermitteln, erfordert dies in JUnit4- und TestNG-Umgebungen ein wenig zusätzlichen Aufwand seitens des Entwicklers.

In den nächsten Abschnitten wird auf die Erfordernisse der unterschiedlichen Umgebungen detailliert eingegangen.

Aufrufen der setUp-Methode unter JUnit3

Bei der Verwendung von JUnit3 ist die Ermittlung des aktuellen Testmethodennamens einfach möglich, da er über die Methode `getName` direkt an der Instanz der Testklasse ermittelt werden kann. Bei der Erzeugung der checkerberry db-Umgebung ist daher lediglich eine Referenz auf die Testinstanz erforderlich. Folgendes Beispiel zeigt die wesentlichen Aspekte der Erzeugung.

```
private CheckerberryDbEnvironment environment;

public void setUp() {
  // Die genaue Erstellung der Umgebung ist hier nicht relevant.
  environment = CheckerberryDb.getInstance().getEnvironment(…);

  environment.setUp(this);
}

public void tearDown() {
  environment.tearDown();
  environment = null;
}
```

Beispiel 2.76. Ermittlung Testmethodennamen unter JUnit3

Bei dem Setup der Umgebung muss lediglich die Referenz auf die Testinstanz angegeben werden. Den Namen der aktuellen Testmethode ermittelt checkerberry db dann selbständig.

Aufrufen der setUp-Methode unter JUnit4 und Spring

Mit der Einführung von JUnit4 wurde das Feature zur einfachen Ermittlung der aktuellen Testmethode nicht mehr zur Verfügung gestellt. Dieser Fehler wurde mit der JUnit Version 4.7 wieder behoben. Für die Verwendung eines JUnit Version zwischen 4 und 4.7 muss man auf die Hilfe zusätzlicher Frameworks zurückgreifen. Wenn ein Upgrade auf die aktuelle JUnit-Version nicht möglich ist, empfehlen wir die Verwendung von Spring. Folgendes Beispiel zeigt die Konfiguration.

```
@RunWith(SpringJUnit4ClassRunner.class)
//Spring Konfiguration
@ContextConfiguration(locations = { "/application-context-test.xml" })
//Der erste für DependencyInjection, der zweite um checkerberry db
//mit dem Namen der Testmethode zu versorgen.
@TestExecutionListeners({ DependencyInjectionTestExecutionListener.class,
  TestMethodNameResolvingExecutionListener.class })
public class AbstractJUnit4SpringTest {
  private CheckerberryDbEnvironment environment;

  @Before
  public void setUp() {
    // Die genaue Erstellung der Umgebung ist hier nicht relevant.
    environment = CheckerberryDb.getInstance().getEnvironment(…);

    environment.setUp(this);
  }

  @After
  public void tearDown() {
    environment.tearDown();
    environment = null;
  }
}
```

Beispiel 2.77. Ermittlung Testmethodennamen unter JUnit4 und Spring

Die JUnit-Annotation `@RunWith` definiert, mit Hilfe welcher Klasse der Test ausgeführt wird. In diesem konkreten Fall wird Spring und JUnit4 verwendet. Über die Spring-Annotation `@ContextConfiguration` wird der zu verwendende Application-Kontext definiert. Dieser Application-Kontext enthält im Beispiel auch Teile der Bridge-Implementation.

Über die Spring-Annotation `@TestExecutionListeners` werden zwei Listener registriert, die vor der Testausführung involviert werden. Der Spring-Listener `DependencyInjectionTestExecutionListener` ist für die Bearbeitung des Application-Kontexts zuständig. Der Listener `TestMethodNameResolvingExecutionListener` dient der Versorgung von checkerberry mit dem Test-Methodennamen.

Bei dem Setup der Umgebung muss lediglich die Referenz auf die Testinstanz angegeben werden.

Aufrufen der setUp-Methode unter JUnit4 ab Version 4.7

Ab der JUnit Version 4.7 wurde die Ermittlung des Namens der aktuellen Testmethode wieder ermöglicht. Das folgende Beispiel zeigt das erforderliche Vorgehen.

```
@Rule
public MethodNameDeterminationRule rule = new MethodNameDeterminationRule();

private CheckerberryDbEnvironment environment;
@Before
public void setUp() {
  // Die genaue Erstellung der Umgebung ist hier nicht relevant.
  environment = CheckerberryDb.getInstance().getEnvironment(…);

  environment.setUp(this);
}
@After
public void tearDown() {
  environment.tearDown();
  environment = null;
}
```

Beispiel 2.78. Ermittlung Testmethodennamen unter JUnit4.7

Durch die Verwendung der Annotation `@Rule` wird der Name der aktuellen Testmethode ermittelt. Bei dem Setup der Umgebung muss lediglich die Referenz auf die Testinstanz angegeben werden. Den Namen der aktuellen Testmethode ermittelt checkerberry db dann selbständig.

Anmerkung: die `@Rule`-Annotation wird von JUnit4 erst ab Version 4.7 bereitgestellt. Sollten Sie eine ältere JUnit4-Version verwenden, so empfehlen wir das Update auf eine aktuelle Version.

Aufrufen der setUp-Methode unter TestNG

Bei der Verwendung von TestNG kann der Name direkt in der Setup-Phase übergeben werden. Folgendes Beispiel zeigt die wesentlichen Aspekte der Erzeugung.

```
private CheckerberryDbEnvironment environment;

@BeforeMethod
public void setUp(Method testMethod) {
  // Die genaue Erstellung der Umgebung ist hier nicht relevant.
  environment = CheckerberryDb.getInstance().getEnvironment(…);
  environment.setUp(this, testMethod);
}

@AfterMethod
public void tearDown() {
  environment.tearDown();
  environment = null;
}
```

Beispiel 2.79. Ermittlung Testmethodennamen unter TestNG

Die TestNG-Annotation `@BeforeMethod` ermöglicht die Angabe eines Method-Objekts als Parameter. TestNG sorgt dann zur Ausführungszeit dafür, dass der Parameter mit der aktuellen Testmethode aufgerufen wird. Bei dem Setup der Umgebung muss dann die Referenz auf die Testinstanz und auf die aktuelle Testmethode angegeben werden. Den Namen der aktuellen Testmethode ermittelt checkerberry db dann selbständig.

Aufrufen der setUp-Methode unter anderen Testframeworks

Selbstverständlich lässt sich checkerberry db auch in anderen Testframeworks als den genannten einsetzen. Die Ermittlung des Namens der aktuellen Testmethode ist von Framework zu Framework unterschiedlich. Das checkerberry test center stellt keine Funktionalität bereit, um den Namen der Testmethoden in anderen Frameworks zu ermitteln.

Wenn der aktuelle Test-Methodenname ermittelt wurde, übergibt man ihn der `setUp`-Methode als zusätzlichen Parameter, um ihn checkerberry db direkt mitzuteilen: `environment.setUp(this, "testMethod");`

2.5.3.2. Einstellen der Transaktionsverwaltung

Bei der Erstellung der checkerberry db-Umgebung muss in der Regel auch das Transaktionshandling berücksichtigt werden.

Checkerberry db kommuniziert über eine eigene JDBC-Verbindung mit der Datenbank. Für die Ausführung von automatisierten Tests bedeutet dies, dass die checkerberry db-Statements und die Statements der zu testenden Komponenten in unterschiedlichen Transaktionen ausgeführt werden. Aus diesem Grund ist es erforderlich, sich Gedanken über das gewünscht Transaktionsverhalten zu machen.

Wenn die zu testenden Komponenten bereits eine geöffnete Transaktion erwarten, sollte in der Setup-Phase eine neue Transaktion geöffnet werden, die in der Teardown-Phase wieder geschlossen wird. Wenn die zu testenden Komponenten hingegen immer in einer eigenen Transaktion ausgeführt werden, ist die explizite Verwaltung der Transaktion im Test nicht erforderlich. Es ist durchaus möglich, dass in unterschiedlichen Teilprojekten unterschiedliche Transaktionsverhalten erforderlich sind. In vielen Projekten werden Transaktionen immer in der Service- oder Business-Schicht geöffnet und geschlossen. Die involvierten DAO-Objekte gehen daher immer davon aus, dass bereits eine Transaktion vorhanden ist. Für das Testen

bedeutet dies, dass bei DAO-Tests der Test selbst für die Transaktionsverwaltung zuständig ist. Bei dem Testen von Services übernimmt der Service hingegen die Transaktionsverwaltung.

Bei der Erstellung der checkerberry db-Tests ist darauf zu achten, dass alle Transaktionen der zu testenden Komponenten geschlossen werden, bevor checkerberry db die erwarteten Ergebnisse prüft. Anderenfalls bleiben die Änderungen für checkerberry db verborgen. Die folgende Abbildung stellt diesen Sachverhalt dar.

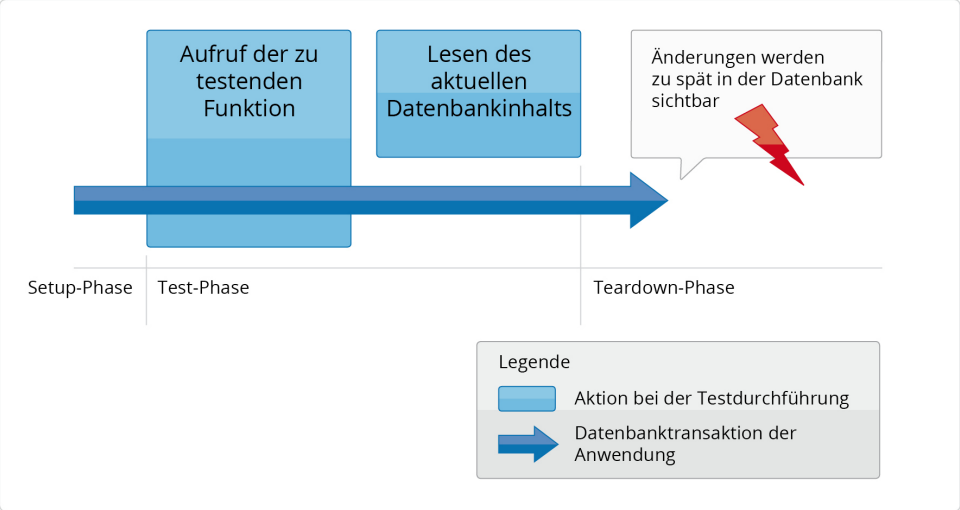


Abbildung 2.38. Transaktionsverhalten

In der Abbildung sind die drei Testphasen Setup, Test und Teardown dargestellt. Die Datenbanktransaktion wird in der Setup-Phase geöffnet und in der Teardown-Phase wieder geschlossen. Innerhalb der Test-Phase werden zunächst die zu testenden Funktionen aufgerufen. Danach erfolgt die Überprüfung der erwarteten Testdaten. Da die Datenbanktransaktion erst in der Teardown-Phase geschlossen wird, werden die Datenbankänderungen auch erst zu diesem Zeitpunkt in der Datenbank sichtbar. Aus diesem Grund liefert die Überprüfung der erwarteten Testdaten nicht das gewünschte Ergebnis.

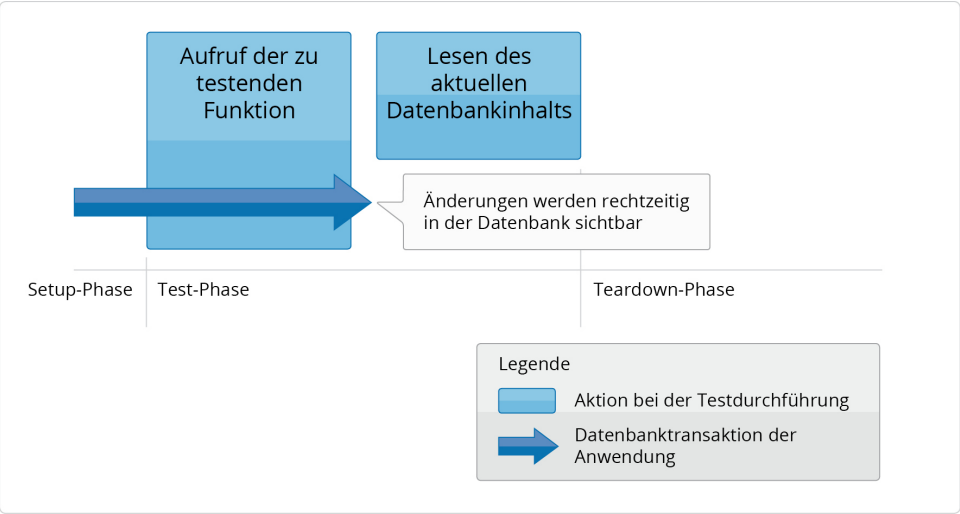


Abbildung 2.39. Angepasstes Transaktionsverhalten

Abbildung 2.39, „Angepasstes Transaktionsverhalten“zeigt die Lösung des zuvor beschriebenen Problems. Bevor die erwarteten Testdaten überprüft werden, muss die Datenbanktransaktion beendet werden. Zu diesem Zweck hat es sich bewährt in der Basis-Testklasse eine Methode zum Schließen der Transaktion zur

Verfügung zu stellen. Diese Methode muss dann vor der Überprüfung der erwarteten Testdaten aufgerufen werden.

Das Transaktionsmanagement ist abhängig von den verwendeten Technologien und den fachlichen Anforderungen an die Anwendung. Aus diesem Grund gibt es keine einheitliche Lösung für die Konfiguration.

2.5.3.3. Spring-Integration

Für die Verwendung der Spring-Frameworks stellt checkerberry db mit dem `SpringCheckerberryDbEnvironmentCreator` eine Klasse bereit, die die Integration von checkerberry vereinfacht.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"/application-context-test.xml"})
@TestExecutionListeners({
    DependencyInjectionTestExecutionListener.class,
    TestMethodNameResolvingExecutionListener.class,
    ApplicationContextResolvingExecutionListener.class })

public abstract class AbstractJUnit4CachingTestCase {
  private CheckerberryDbEnvironment environment;
  @Before
  public void setUp() {
    environment = SpringCheckerberryDbEnvironmentCreator.createEnvironment();
    environment.setUp(this);
  }

  @After
  public void tearDown() {
    environment.tearDown();
    environment = null;
  }
}
```

Beispiel 2.80. Spring-Integration

Das obige Beispiel zeigt die Integration von checkerberry db unter Verwendung von JUnit4 und Spring (siehe „Aufrufen der setUp-Methode unter JUnit4 und Spring"). Neben den zuvor beschriebenen `TestExecutionListenern`, `DependencyInjectionTestExecutionListener` und `TestMethodNameResolvingExecutionListener` wird ein weiterer `TestExecutionListener` verwendet. Der `ApplicationContextResolvingExecutionListener` ermittelt den aktuellen Spring-Application-Context und speichert diesen innerhalb des checkerberry test centers in der Klasse `ApplicationContextResolver`. Über die statische Methode `getTestApplicationContext` steht der Application-Kontext dann zur Verfügung.

In der `setUp`-Methode wird checkerberry db-Umgebung über die Klasse `SpringCheckerberryDbEnvironmentCreator` erzeugt. Der Creator stellt folgende Methoden zur Erstellung der checkerberry db-Umgebung zur Verfügung.

```
/**
 * Erzeugt ein neues Default-Environment über einen {@link BridgeContainer}.
 * Der BridgeContainer wird über den Namen "bridgeContainer" aus dem
 * Application-Kontext von Spring gelesen.
 *
 * @return erzeugtes Environment.
 */
 public static CheckerberryDbEnvironment createEnvironment() {...}

/**
 * Erzeugt ein neues Default-Environment über einen {@link BridgeContainer}.
 * Der BridgeContainer wird über den angegebenen Namen aus dem
 * Application-Kontext von Spring gelesen.
 *
 * @param bridgeContainerBeanName
 *            Bean-Name des Containers im Application-Kontext.
 * @return erzeugtes Environment.
 */
public static CheckerberryDbEnvironment createEnvironment(
    String bridgeContainerBeanName) {...}

/**
* Erzeugt ein neues Environment über einen {@link BridgeContainer}. Der
* BridgeContainer wird über den angegebenen Namen aus dem
* Application-Kontext von Spring gelesen.
*
* @param environmentId
*            Name der Environment-Id.
* @param bridgeContainerBeanName
*            Bean-Name des Containers im Application-Kontext.
* @return erzeugtes Environment.
*/
public static CheckerberryDbEnvironment createEnvironment(
    CheckerberryDbEnvironmentId environmentId,
    String bridgeContainerBeanName) {...}

/**
 * Erzeugt ein neues Default-Environment mit dem Standard {@link
 * de.conceptpeople.checkerberry.db.bridge.resource.ClasspathResourceLoader}
 * . Der {@link DatabaseConnector} wird über den Namen "databaseConnector"
 * aus dem Application-Kontext von Spring gelesen.
 *
 * @param databaseDescriptionCallback
 *            zu verwendendes Callback zur Definition der
 *            Datenbankbeschreibung.
 * @param configurationCallback
 *            zu verwendendes Callback für die Konfiguration.
 * @return erzeugtes Environment.
 */
 public static CheckerberryDbEnvironment createEnvironment(
   DatabaseDescriptionCallback databaseDescriptionCallback,
   DbConfigurationCallback configurationCallback) {...}
```

Beispiel 2.81. SpringCheckerberryDbEnvironmentCreator

Der `SpringCheckerberryDbEnvironmentCreator` greift über die Klasse `ApplicationContextResolver` auf den Application-Kontext von Spring zu und erzeugt anhand einiger Konventionen checkerberry db-Umgebung. Wenn man die Bridge-Komponenten in einen `de.conceptpeople.checkerberry.db.bridge.BridgeContainer` zusammenfasst und diesen im Application-Kontext unter den Bean-Namen „`bridgeContainer`“ registriert, benötigt der `SpringCheckerberryDbEnvironmentCreator` keine weiteren Parameter für die Erzeugung von checkerberry db. Der Aufruf von `createEnvironment()` ohne Parameter ist dann ausreichend.

2.6. Erstellen von Tests

Nach dem Einbinden von checkerberry db kann mit der Erstellung der Tests begonnen werden. Bei der Erstellung des ersten Tests muss die zu verwendende DTD erzeugt werden. Das genaue Vorgehen dazu ist in Abschnitt 2.4.1.5, „Anlegen einer neuen Testdaten-DTD“beschrieben.

Um einen Test zu schreiben, wird eine neue Testklasse erstellt. In der Setup-Phase der Testklasse muss die checkerberry db-Umgebung initialisiert werden. Dies kann entweder direkt in der Testklasse oder in einer Super-Klasse erfolgen. In dem folgenden Beispiel erfolgt dieser Schritt in der Super-Klasse, da das der häufigste Anwendungsfall ist.

```
package de.conceptpeople.sample.dao;

public class UserDaoIntegrationTest extends AbstractSampleTestCase {
  // Zu testendes Data Access Object (DAO)
  private UserDao dao;

  public void testGetUser() throws Exception {
    // Lesen eines Users aus der Datenbank.
    User user = dao.getUser("Homer", "Simpson");
    // Prüfen, ob der User gefunden wurde.
    assertNotNull(user);

    // Datenbanktransaktion schließen.
    persist();

    // Test-Handler holen.
    DbTestHandler testHandler = getEnvironment().getTestHandler();
    // Sicherstellen, dass die initialen Testdaten nicht verändert
    // wurden.
    testHandler.assertInitialDataUnchanged();
  }
  …
}
```

Beispiel 2.82. Beispiel Testklasse

In dem Beispiel wird eine Testklasse mit dem Namen `UserDaoIntegrationTest` erzeugt, die von der abstrakten Oberklasse `AbstractSampleTestCase` erbt. Der Name der Testklasse setzt sich zusammen aus dem Namen der zu testenden Klasse (`UserDao`) und dem Suffix `IntegrationTest`. Für die getrennte Ausführung von Unit- und Integrationstests z.B. in Maven-Umgebungen, ist die Markierung der Integrationstests sinnvoll. Aus diesem Grund wird das Suffix `IntegrationTest` verwendet. Normale Unit-Tests, die keine Integrationsumgebung benötigen, erhalten nur das Suffix `Test` (siehe auch Kapitel Abschnitt 7.1.1, „Namenskonvention").

Zunächst wird über das `UserDao`-Objekt der User für Homer aus der Datenbank gelesen und auf Objekt-Existenz geprüft. Danach wird die Transaktion über die Methode `persist` der Oberklasse geschlossen, damit etwaige Änderungen in der Datenbank persistiert werden. Die Implementierung der `persist` -Methode ist dabei abhängig von der Persistenzschicht des Kunden. Nach dem Abschließen der Transaktion wird überprüft, ob die initialen Testdaten unverändert sind.

Wenn dieser Test ausgeführt wird, schlägt er fehl, da noch keine initialen Testdaten vorhanden sind. Gemäß der Konvention von checkerberry db muss eine Datei `de/conceptpeople/sample/dao/UserDaoIntegrationTest_testGetUser_initial.xml` angelegt werden. In Maven-Projekten ist `src/test/resources` das richtige Verzeichnis für die XML-Testdaten. Wichtig ist, dass die Entwicklungsumgebung so eingestellt ist, dass sie XML-Dateien in den Klassenpfad kopiert, damit die Testdaten in dem Klassenpfad gefunden werden.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC "-//ConceptPeople//DTD sample-db 1.0//EN"
"../sample-db.dtd">

<dataset>
  <USERS NAME="Bart" SURNAME="Simpson"/>
  <USERS NAME="Lisa" SURNAME="Simpson"/>
  <USERS NAME="Maggie" SURNAME="Simpson"/>
  <USERS NAME="March" SURNAME="Simpson"/>
  <USERS NAME="Homer" SURNAME="Simpson"/>
</dataset>
```

Beispiel 2.83. Beispiel Testdaten

Das obige Beispiel zeigt mögliche Testdaten für den Test. In der Setup-Phase des Tests wird die Simpson-Familie in die Datenbank eingetragen. Die Ausführung des Tests ist jetzt erfolgreich, da das `UserDao`-Objekt Homer aus der Datenbank lesen kann.

Kapitel 3. Checkerberry web

3.1. Einleitung

Checkerberry web ist eine Bibliothek, die den Software-Entwickler bei dem funktionalen Testen von Web-Anwendungen unterstützt. Zu diesem Zweck wird ein Browser über den Integrationstest gesteuert. Für diese Fernsteuerung wird das Open-Source-Framework Selenium 2.0 [Selenium Homepage, 2010] verwendet. Auf diese Art und Weise können alle Komponenten einer Webseite angesprochen werden. Werte von Feldern können ausgelesen werden, Links können gedrückt werden etc. Standardmäßig verwendet checkerberry web das Modul Selenium RC (Remote Control) für die Fernsteuerung des Browsers. Es ist jedoch auch einfach möglich, die Fernsteuerung auf WebDriver umzustellen. WebDriver wurde mit der Version 2.0 in Selenium aufgenommen und wird sich vermutlich mittelfristig als bevorzugte Lösung durchsetzen - gerade für die neuen Browsergenerationen. Aktuell ist es jedoch so, dass sowohl Selenium RC als auch WebDriver gleichberechtigt nebeneinander stehen. Die Auswahl des Verfahrens hängt von dem zu testenden Browser und der zu testenden Funktionalität ab. Aus diesem Grund unterstützt checkerberry web beide Ausprägungen.

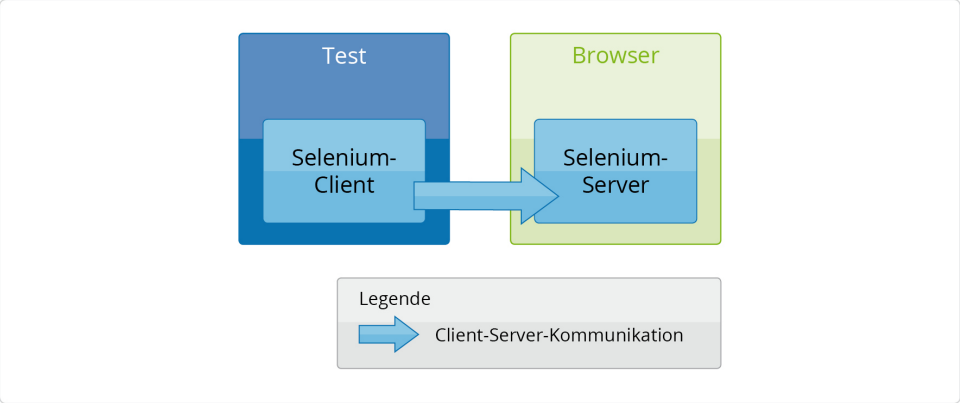


Abbildung 3.1. Selenium Fernsteuerung

Abbildung 3.1, „Selenium Fernsteuerung“ skizziert die Funktionsweise von Selenium. Innerhalb der Testklasse wird eine Client-Bibliothek von Selenium verwendet, die mit einer Server-Komponente im Browser kommuniziert. Die Server-Komponente verwendet JavaScript für die Kommunikation mit dem Browser und für den Zugriff auf die Webseiten.

Das Ziel von checkerberry web ist es, dem Software-Entwickler eine Out-of-the-Box-Lösung für das schnelle und einfache Testen von Web-Anwendungen zur Verfügung zu stellen.

Ein häufig auftretendes Problem bei der Erstellung von automatisierten GUI-Tests besteht in der schlechten Wartbarkeit der Tests. Änderungen an den Oberflächen der Anwendung ziehen häufig übermäßig viel Änderungsaufwand in den Tests nach sich. Wenn diese Änderungen zu häufig auftreten, kommt es früher oder später zu dem Punkt, an dem die Tests nicht mehr gepflegt werden. Die Testabdeckung fällt dann rapide auf 0%.

Um diese Entwicklung zu vermeiden, verwendet checkerberry web einen Modellansatz (Page Object Pattern [Google Page Objects, 2010]), der die Informationen einer Webseite kapselt. Änderungen der Anwendungen haben dadurch lediglich geringe Auswirkungen auf die Tests und die zugehörigen Modelle. Mit checkerberry web erstellte Tests können so über Jahre hinweg gepflegt und erweitert werden.

Die Kombination von checkerberry db und checkerberry web eröffnet dem Software-Entwickler vielfältige Testmöglichkeiten. Durch checkerberry db wird der Datenbestand der Web-Anwendung vor der

Testausführung manipuliert. Auf diese Art und Weise ist es möglich, spezielle Aspekte der Web-Anwendung z.B. eines Online-Shops zu testen. Durch das Einspielen eines gefüllten Warenkorbs könnte man sich in dem Test der Web-Anwendung z.B. auf das Testen des Bezahlvorgangs konzentrieren. Das aufwändige Anlegen eines großen Warenkorbs über die Web-Anwendung ist dann nicht erforderlich. Generell ermöglicht die Kombination von checkerberry db und checkerberry web die Erstellung von Tests in unterschiedlicher Granularität.

Ein weiterer Vorteil besteht in der Möglichkeit, die Auswirkungen der GUI-Tests in der Datenbank zu überprüfen.

3.2. Funktionsweise

Der allgemeine Ablauf von Unit-Tests in den drei Phasen (Setup, Test und Teardown) wurde bereits im Kapitel Abschnitt 1.3.3.1, „Test-Ablauf"beschrieben. Die folgenden Abschnitte beschreiben das Verhalten von checkerberry web in diesen Phasen.

3.2.1. Setup-Phase

In der Setup-Phase werden die Voraussetzungen für die Durchführung der Tests geschaffen. Zu diesem Zweck muss die checkerberry web-Umgebung erzeugt werden, was ausführlich in Abschnitt 3.5.2, „Erzeugen der checkerberry web-Umgebung" beschrieben wird. Die folgende Abbildung beschreibt die Auswirkungen, die die Erzeugung der checkerberry web-Umgebung bei der Verwendung von Selenium RC hat.

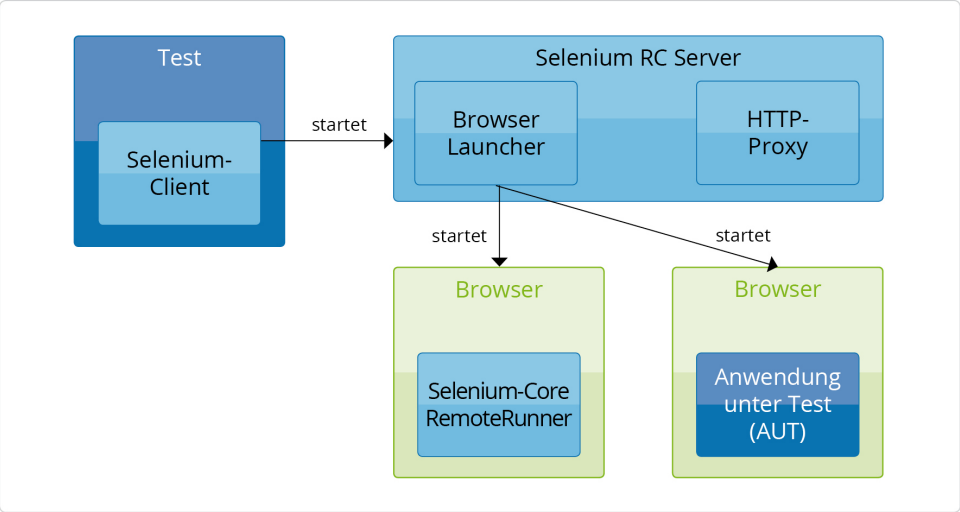


Abbildung 3.2. Checkerberry web Setup-Phase

Die checkerberry web-Umgebung wird in der Setup-Phase durch einen Selenium-Client gestartet, der über eine Bibliothek in den Test eingebunden ist. Durch die Erzeugung der checkerberry web-Umgebung wird eine weitere Selenium-Komponente, der Selenium RC Server, gestartet. Bei dem Server handelt es sich um ein Software-Programm, das zwei weitere Komponenten beinhaltet: Einen Browser-Launcher und einen HTTP-Proxy. Der Browser-Launcher startet zwei neue Browserfenster. In einem Browserfenster wird die HTML-Seite `RemoteRunner.html` geöffnet, die den Selenium-Core darstellt. In dem anderen Browserfenster wird die zu testenden Anwendung (AUT = Anwendung unter Test) gestartet. Korrekterweise handelt es sich bei der Anwendung unter Test um eine zu testende Webseite, die von einem Webserver geladen wurde. Die URL dieser Webseite wurde der checkerberry web-Umgebung beim Start übergeben.

Der Selenium-Core besteht aus einer Sammlung von JavaScript-Funktionen, die zur Kommunikation mit der zu testenden Anwendung benötigt werden. Die Kommunikation der verschiedenen Komponenten wird in der Test-Phase beschrieben.

Beim Start der Browserfenster sorgt Selenium dafür, dass der HTTP-Proxy aus dem Selenium RC Server in den Browsern verwendet wird. Dieses Verfahren wird „Proxy Injection" [Proxy Injection, 2010] genannt und ist eine Lösung zur Umgehung der „Same Origin Policy" [Same Origin Policy, 2010]. Die Bedeutung dieses Vorgehens wird in dem nächsten Kapitel beschrieben, da die Problematik anhand der Test-Phase besser verdeutlicht werden kann.

Die Kommunikation zwischen Test und Browser ist bei der Verwendung von WebDriver deutlich einfacher. Dies liegt daran, dass der Browser selbst ein WebDriver-Plug-In enthält, das die Kommunikation mit dem Test ermöglicht.

3.2.2. Test-Phase

Die Test-Phase beginnt durch den Aufruf einer Testmethode. Innerhalb der Testmethode greift der Test über ein Modell auf die Seiten der Web-Anwendung in der gestarteten Browser-Instanz zu. Die Art der Tests kann dabei sehr stark variieren.

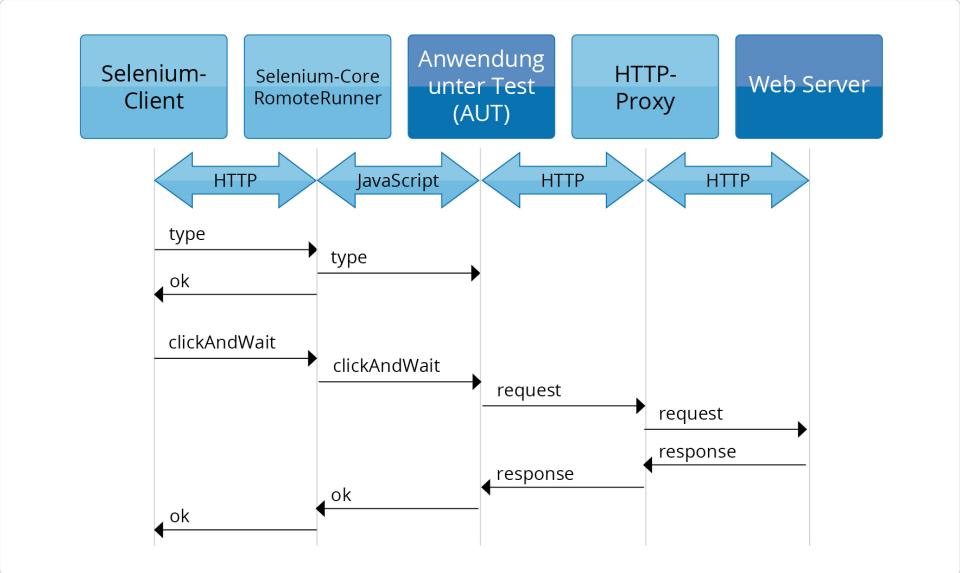


Abbildung 3.3. Checkerberry web Test-Phase

Die obige Abbildung beschreibt die Kommunikation der am Test beteiligten Komponenten. Der Selenium-Client wird aus dem Test heraus verwendet, um die Felder der zu testenden Webseite zu manipulieren. Über das HTTP-Protokoll kommuniziert der Selenium-Client direkt mit dem Selenium-Core innerhalb des Browsers. In dem Beispiel wird der Befehl `type` zur Eingabe eines Textes in ein Feld übergeben. Der Selenium-Core empfängt den Request und führt ihn mit Hilfe von JavaScript in dem zweiten Browserfenster aus. Dies führt dazu, dass in der zu testenden Webseite ein Wert eingetragen wird. Nach der erfolgreichen Bearbeitung sendet der Selenium-Core eine Bestätigung an den Selenium-Client.

Der erste Teil des Beispiels verdeutlicht, dass man Selenium sehr gut als Fernsteuerung interpretieren kann. Durch den Selenium-Client können die Komponenten einer Webseite über eine Testmethode manipuliert werden.

Der zweite Teil des Beispiels aus Abbildung 3.3, „Checkerberry web Test-Phase" beginnt mit der Aktion `clickAndWait`, die eine HTML-Komponente drückt und auf ein Ergebnis wartet. Der Selenium-Client sendet die Anfrage wieder an den Selenium-Core, der das Drücken z.B. eines Buttons auf der Webseite ausführt. Das Drücken des Buttons führt in dem Beispiel zum Laden einer neuen Webseite. Durch die vorgenommene „Proxy Injection" wird der HTTP-Request zum Laden der neuen Webseite über den HTTP-Proxy bearbeitet. Dieser sendet einen eigenen Request an den Web-Server. Die Response des Webservers

wird durch den HTTP-Proxy manipuliert und an den Browser zurückgeliefert. Der Selenium-Core übermittelt dann die erfolgreiche Verarbeitung an den Selenium-Client.

Die „Same Origin Policy" lässt sich anhand dieses Beispiels gut erklären. Alle Browser erlauben die Ausführung von JavaScript-Aufrufen nur, wenn sowohl das JavaScript als auch die zugehörige Webseite von dem gleichen Server geladen wurden [Same Origin Policy, 2010]. Diese Situation ist bei der Verwendung von Selenium zunächst nicht gegeben, da die JavaScript-Funktionen des Selenium-Core nicht über den Web-Server der zu testenden Anwendung geladen wurden. Aus diesem Grund manipuliert der HTTP-Proxy des Selenium RC Servers die Kommunikation zwischen Browser und Web-Server. Auf diese Art und Weise wird dem Browser vorgetäuscht, dass sowohl JavaScript wie auch die Webseite vom gleichen Server geladen wurden, sodass die JavaScript-Funktionen des Selenium-Core auf die zu testende Webseite zugreifen können.

3.2.3. Teardown-Phase

Die folgende Abbildung beschreibt das Vorgehen von checkerberry web in der Teardown-Phase.

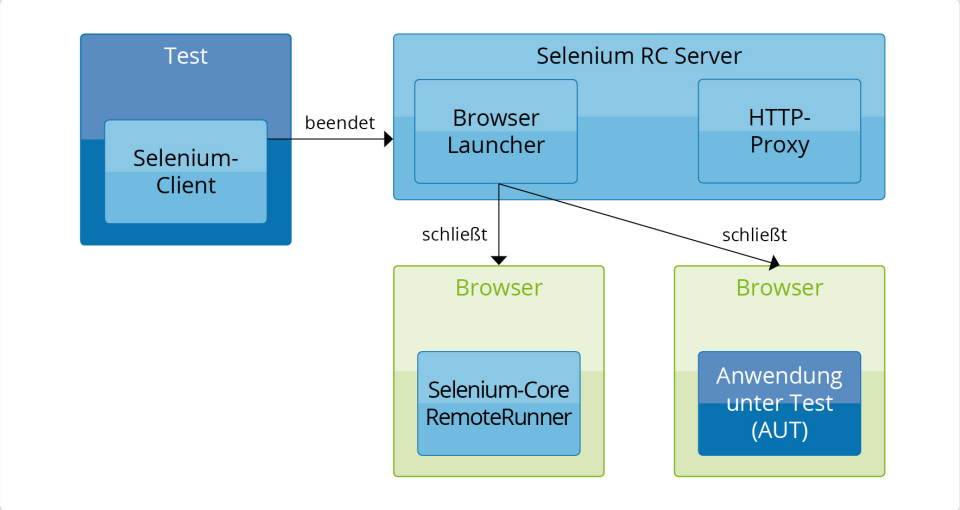


Abbildung 3.4. Checkerberry web Teardown-Phase

In der Teardown-Phase wird der Selenium RC Server beendet. Durch die Beendigung des Servers werden auch die gestarteten Browser-Instanzen geschlossen. Dieses Verhalten ist jedoch konfigurierbar, sodass Browserinstanzen auch wiederverwendet werden können.

3.3. Architektur

Checkerberry web ist für den Einsatz in unterschiedlichen Umgebungen modular aufgebaut. Die folgende Grafik beschreibt die Architektur von checkerberry web.

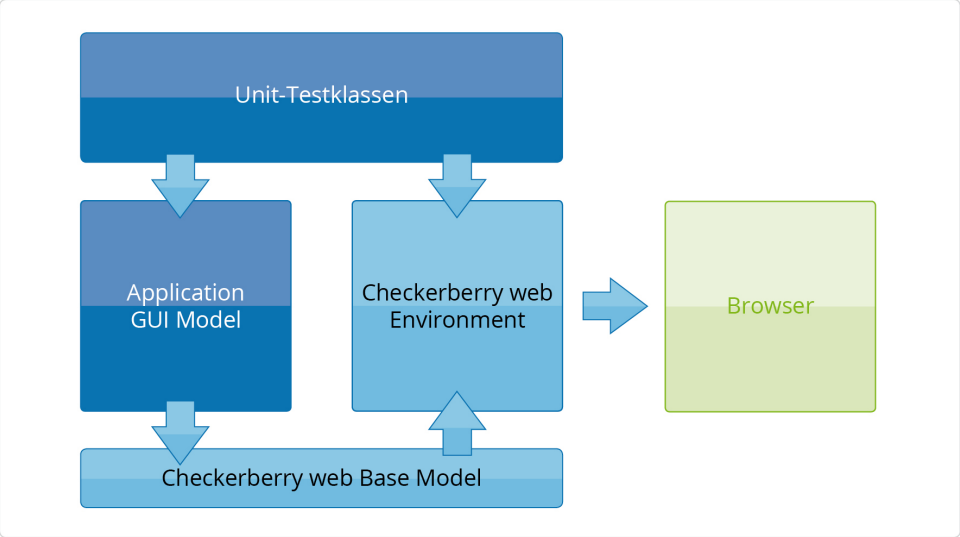


Abbildung 3.5. Architektur checkerberry web

Die Unit-Testklassen greifen in der Setup- und Teardown-Phase direkt auf die Umgebung von checkerberry web zu. Bei der Testdurchführung verwenden die Tests jedoch ausschließlich die Modellklassen für die Kommunikation mit Selenium. Checkerberry web stellt Basisklassen für die schnelle und einfache Erstellung der Modellklassen zur Verfügung. Die Basisklassen verwenden wiederum die Umgebung von checkerberry web für die Kommunikation mit dem Browser.

3.4. Funktionsumfang

Die wesentliche Stärke von checkerberry web liegt in der Verwendung des Modellansatzes, um nachhaltige GUI-Tests zu erstellen. Darüberhinaus gibt es einige Features, die in diesem Kapitel beschrieben werden.

3.4.1. Wiederverwendung von Browser-Instanzen

Die Ausführung der checkerberry web-Tests ist zeitintensiver als die Ausführung von reinen Modultests. Insbesondere der Start der Browser-Instanzen kann einige Sekunden in Anspruch nehmen, was bei einer großen Anzahl von Tests zu merkbaren Verzögerungen führt. Aus diesem Grund unterstützt checkerberry web verschiedene Verfahren, um das Starten einer neuen Browser-Instanz zu steuern.

```
public interface NewBrowserInstancePolicy {

  /**
   * Prüft, ob eine neue Browser-Instanz für den aktuellen Kontext
   * verwendet werden muss.
   *
   * @param lastContext
   *            vorheriger Kontext oder <code>null</code>, wenn vorher
   *            keine Test-Methoden ausgeführt wurden. D.h. die aktuelle
   *            Methode ist der erste Test.
   * @param currentContext
   *            aktueller Kontext.
   * @param invocationCount
   *            Anzahl der Test-Ausführungen, die bereits mit der
   *            Browser-Instanz durchgeführt wurden.
   * @return <code>true</code>, wenn eine neue Browser-Instanz für den
   * aktuellen Kontext verwendet werden soll.<br/>
   * <code>false</code>, sonst.
   */
  boolean isNewBrowserInstanceRequired(WebContext lastContext,
    WebContext currentContext, int invocationCount);
}
```

Beispiel 3.1. Interface NewBrowserInstancePolicy

Über das Interface `NewBrowserInstancePolicy` wird gesteuert, wann eine neue Browser-Instanz gestartet werden muss. Die Methode `isNewBrowserInstanceRequired` wird vor jedem Testfall aufgerufen und kann anhand des aktuellen und des vorherigen Web-Kontext entscheiden, ob eine neue Browser-Instanz erforderlich ist. Des Weiteren wird die Anzahl der Testfälle übergeben, die bereits mit der aktuellen Browser-Instanz gestartet wurden.

Als Standard-Verfahren wird die Klasse `DefaultNewBrowserInstancePolicy` verwendet. Diese Klasse kann konfiguriert werden oder als Basis für eigene Konfigurationen verwendet werden. Folgende Einstellungen können vorgenommen werden:

- `newBrowserForEachClassRequired`: Wenn dieser Wert auf `true` gesetzt ist, wird für jede Klasse eine neue Browser-Instanz gestartet. Der Default ist `true`.
- `browserReuseCount`: Dieser Wert definiert, wie viele Tests maximal in einer Browser-Instanz ausgeführt werden können. Der Default ist 1. Ist der Wert 0 oder negativ, werden beliebig viele Tests in einer Browser-Instanz ausgeführt.

Die folgende Tabelle zeigt mögliche Konfigurationen.

Tabelle 3.1. Konfiguration von checkerberry web

	`newBrowserForEachClassRequired`	
browserReuseCount	`true`	`false`
0	Pro Klasse wird eine neue Browser-Instanz verwendet. Alle Methoden der Klasse verwenden die gleiche Instanz.	Alle Testmethoden verwenden die gleiche Browser-Instanz.
1	Das ist der Default. Pro Klasse und Methode wird eine neue Browser-Instanz verwendet.	Jede Testmethode verwendet eine eigene Browser-Instanz.
5	Pro Klasse wird eine neue Browser-Instanz verwendet. Die Instanz wird für maximal fünf Testmethoden der Klasse wiederverwendet.	Jede Browser-Instanz wird für bis zu 5 Testmethoden verwendet, wobei die Testmethoden aus unterschiedlichen Klassen stammen können.

Es gibt Testklassen oder –methoden, die immer eine eigene Browser-Instanz benötigen. In diesem Fall kann man die Testklasse oder die entsprechenden Testmethoden mit der Annotation `@NewBrowserInstance` markieren. Ist eine Testklasse annotiert, wird für die Klasse eine neue Browser-Instanz gestartet. Die Annotation an der Klasse führt jedoch nicht dazu, dass für jede Testmethode der Klasse eine neue Browser-Instanz gestartet wird. Ist eine Testmethode entsprechend annotiert, wird für diese Methode eine neue Browser-Instanz gestartet.

Eine hohe Wiederverwendung der Browser-Instanzen beschleunigt die Testdurchführung. Bei einigen Browsern kann die Wiederverwendung jedoch auch zu einem hohen Speicherverbrauch im Browser führen, was die Testausführung bremsen kann. Es sollte in jedem Projekt selbst getestet werden, welcher Grad der Wiederverwendung sinnvoll ist.

3.4.2. Festlegen der Test-Url

Die Test-Url kann über die checkerberry web-Konfiguration gesetzt werden. Als weitere Möglichkeit steht das Setzen der Test-Url über eine Annotation zur Verfügung. Das folgende Code-Beispiel zeigt ihre Verwendung.

```
@TestUrl("myClassAnnotatedUrl.html")
public class TestUrlAnnotationTest {

  @Test
  public void testTestUrlAnnotationAtClass() {
    // Für diesen Test wird die Test-Url
    // "myClassAnnotatedUrl.html" genutzt
  }

  @Test
  @TestUrl("myMethodAnnotatedUrl.html")
  public void testTestUrlAnnotationAtClassAndMethod() {
    // Für diesen Test wird die Test-Url
    // "myMethodAnnotatedUrl.html" genutzt
  }
}
```

Beispiel 3.2. Setzen der Test-Url über die Annotation

Wird die Test-Url über mehrere Kanäle konfiguriert, so gilt folgende Reihenfolge:

1. TestUrl-Annotation an der Methode
2. TestUrl-Annotation an der Klasse
3. Test-Url in der checkerberry web-Konfiguration

3.4.3. Lokalen Domain Name Service (DNS) ändern

Bei der Initialisierung der Tests werden in der Klasse `java.net.InetAddress` automatisch alle Namen der lokal verwendeten IP-Adressen aufgelöst. Dies ist in der Regel überflüssig und kann zudem aufgrund hoher Timeouts sehr lange dauern. Aus diesem Grund besteht die Möglichkeit, andere Strategien festzulegen.

```
public interface WebConfiguration {
  ...
 /**
  * Setzt den DNS Modus. In der Klasse {@link java.net.InetAddress} wird
  * statisch ein DNS Name Service aufgelöst, der sehr inperformant die
  * Hostnamen aller IP-Adressen auflöst. Das führt zu erheblichen
  * Verzögerungen beim Start. Aus diesem Grund können performantere Varianten
  * ausgewählt werden.
  *
  * @param dnsMode
  *            zu verwendender DNS Modus.
  * @see DnsNameServiceProvider
  */
  void setDnsMode(DnsMode dnsMode);
  ...
}
```

Beispiel 3.3. Konfiguration der DNS-Strategie

Das obige Code-Beispiel enthält die Methode zur Änderung der DNS-Strategie. Die folgenden Einstellungen können vorgenommen werden.

- `DnsMode.JavaDefault`: Das ist die Default-Einstellung von Java. Diese Einstellung kann gerade bei lokalen Tests ohne lokalen DNS sehr langsam sein. Bei der Auswahl dieses Modus wird im `java.net.InetAddress` nicht manipuliert. Es ist somit auch möglich, in diesem Modus einen ganz anderen DNS Name Service über das allgemeine Service Provider Interface (SPI) anzugeben. Leider wird der DNS Name Service im `java.net.InetAddress` statisch beim Laden der Klasse ermittelt. Es ist somit nicht möglich programmatisch einen anderen SPI zu verwenden. Das muss dann über VM-Argumente geschehen, was gerade bei der Ausführung von Tests sehr hinderlich ist, da das einfache Starten eines Tests über das Kontextmenü nicht mehr möglich ist. Stattdessen muss der Start aufgrund der erforderlichn VM-Argumente erst konfiguriert werden.

- `DnsMode.LocalDns`: Das ist der Default für checkerberry web. Dieser Name Server löst die Namen immer zu "localhost" und die IP-Adressen immer zur lokalen IP-Adresse auf. Durch diese Einstellung starten die Tests sehr schnell. Sollte die statische Namensauflösung zu Problemen führen, muss entsprechend der Anforderungen eine andere Strategie verwendet werden.
- `DnsMode.DnsJava`: Diese Einstellung verwendet einen in Java implementierten DNS Name Service von www.dnsjava.org ([dnsjava, 2012]). Die Timeouts und Retries sind sehr niedrig eingestellt, sodass diese Variante auch schneller als `DnsMode.JavaDefault` ist. Allerdings ist die Variante langsamer als `DnsMode.LocalDns` und loggt auch sehr viel auf `System.out` und `System.err`.

3.4.4. Fernsteuerung auswählen (Selenium RC und/oder WebDriver)

Die Kommunikation mit dem Browser wird innerhalb von checkerberry web über Selenium ([Selenium Homepage, 2010]) erledigt. Dabei kann zwischen zwei Implementierungen der Fernsteuerung gewählt werden. Per Default wird Selenium RC für die Kommunikation mit dem Browser verwendet. Es ist jedoch auch möglich, WebDriver als Fernsteuerung auszuwählen. Das folgende Code-Beispiel zeigt die Methode zur Konfiguration der Fernsteuerung.

```
public interface WebConfiguration {
  ...
 /**
  * Wählt die Implementierung der Fernsteuerung:
  * {@link RemoteControlSelection#SeleniumRC} (default) oder
  * {@link RemoteControlSelection#WebDriver}.
  *
  * @param remoteControlSelection
  *            {@link RemoteControlSelection#SeleniumRC}, wenn der Test mit
  *            Selenium RC verwendet werden soll.<br/>
  *            {@link RemoteControlSelection#WebDriver}, wenn der Test mit
  *            WebDriver verwendet werden soll.
  */
  void setRemoteControlSelection(RemoteControlSelection remoteControlSelection);
  ...
}
```

Beispiel 3.4. Konfiguration der Fernsteuerung

Die Konfiguration legt für eine checkerberry web-Umgebung fest, wie die Kommunikation mit dem Browser erfolgen soll. Es ist dabei zu beachten, dass in einem GUI-Test mehrere checkerberry web-Umgebungen verwendet werden können, sodass parallel mehrere Browser-Instanzen verwendet werden. Jede checkerberry web-Umgebung verwaltet eine Browser-Instanz. Bei der Verwendung von mehreren checkerberry web-Umgebungen können dementsprechend die verwendeten Fernsteuerungen variieren. Es ist in einem Test beispielsweise möglich, mit Selenium RC eine Browser-Instanz vom Internet Explorer zu verwalten und parallel eine aktuelle Version des Firefox-Browsers mit WebDriver zu steuern.

3.4.5. Registrieren neuer WebDriver-Instanzen

Selenium unterstützt bereits einige Browser direkt für die Verwendung von WebDriver. Dazu gehören Firefox und Internet Explorer. Für andere Browser wie z.B. Google Chrome ist der Clientseitige Teil ebenfalls vorhanden. Für die Verwendung von Google Chrome in einem Test ist jedoch die Installation des WebDriver-Plugins für den Browser erforderlich. Für andere Browser wie z.B. Opera enthält Selenium keine Klassen zur Verwendung von WebDriver. Diese Klassen können dann über externe Quellen bezogen und integriert werden. Im Folgenden wird beschrieben, wie sie externe WebDriver-Instanzen in checkerberry web integrieren können.

Bei der Verwendung von WebDriver prüft checkerberry web zunächst, welcher Browsertyp verwendet werden soll. Zu diesem Zweck verwendet checkerberry web die Methode `WebConfiguration.getBrowserType()`. Um ein möglichst einfaches Umschalten von Selenium RC und WebDriver zu ermöglichen, werden die Browsertypen von Selenium RC wiederverwendet. Intern

werden diese Werte dann den bekannten WebDriver-Instanzen zugeordnet. Die folgende Liste enthält die aktuellen Zuordnungen.

- `*googlechrome` startet den WebDriver für den Google Chrome-Browser.
- `*firefox`, `*firefoxproxy` und `*firefoxchrome` startet den WebDriver für den Firefox-Browser.
- `*iexploreproxy`, `*iexplore` und `*piiexplore` startet den WebDriver für den Internet Explorer-Browser.

Zur Verwendung eines WebDrivers für einen anderen Browser, muss der neue WebDriver registriert werden. Dazu existiert in `WebConfiguration` die Methode `registerWebDriverCreator(WebDriverCreator creator)`. Es wird also nicht direkt eine WebDriver-Instanz, sondern ein Creator-Objekt registriert. Auf diese Art und Weise steuert checkerberry web den Zeitpunkt, wann die WebDriver-Instanz erzeugt wird. Dies ist sinnvoll, da mit der Erzeugung der WebDriver-Instanz auch ein Browserfenster geöffnet wird, was nicht bei der Registrierung, sondern vor jedem Test erforderlich ist. Das folgende Code-Beispiel enthält das Interface `WebDriverCreator`.

```
/**
 * Creator für die Erzeugung von {@link WebDriver}-Objekten. Durch die
 * Verwendung eines Creators können verschiedene Implementierungen in der
 * Konfiguration gespeichert werden.
 */
public interface WebDriverCreator {

  /**
   * Erzeugt einen neuen {@link WebDriver}.
   *
   * @return erzeugter {@link WebDriver}.
   */
  WebDriver createWebDriver();

  /**
   * Liste der Browsertypen, die der zugehörige {@link WebDriver} unterstützt.
   *
   * @return Liste der Browsertypen.
   */
  String[] getRegisteredBrowserTypes();

}
```

Beispiel 3.5. Interface WebDriverCreator

Das Interface verfügt über zwei Methoden. Die Methode `getRegisteredBrowserTypes` liefert alle Browsertypen zurück, die durch den entsprechenden WebDriver abgedeckt werden sollen. Es können dabei auch eigene Namen verwendet werden. Bei der Verwendung von eigenen Namen ist jedoch zu beachten, dass ein einfaches Umschalten zwischen WebDriver und Selenium RC dann nicht mehr möglich ist, da Selenium RC die Browsertypen mit den neuen Namen nicht erkennt. In der Regel ist es jedoch so, dass für jeden Test klar festgelegt wird, ob WebDriver oder Selenium RC verwendet werden soll. Das Umschalten zwischen WebDriver und Selenium RC kommt somit in der Praxis eher selten vor.

In jeder `WebConfiguration` wird über die Methode `getBrowserType` ein Browsertyp festgelegt, der für die entsprechenden Tests verwendet werden soll. Diese Methode muss für die Verwendung von WebDriver somit einen der registrierten Namen zurückliefern.

Die Methode `createWebDriver` wird intern durch checkerberry web verwendet, um die konkrete WebDriver-Instanz zu erzeugen.

3.4.6. Konfigurieren von WebDriver-Instanzen

Teilweise ist es erforderlich, dass WebDriver-Instanzen für die entsprechende Umgebung konfiguriert werden. Bei der Registrierung von neuen WebDriver-Instanzen erfolgt diese Konfiguration in den

`WebDriverCreator`-Implementierungen. Bei der Verwendung von bestehenden WebDriver-Instanzen wie z.B. für den Firefox-Browser sind bereits Basisklassen vorhanden, denen die Konfiguration übergeben werden kann. Zu diesem Zweck verfügen die Creator-Instanzen über Konstruktoren mit den gleichen Parametern wie die WebDriver-Instanzen selbst. Das folgende Code-Beispiel enthält die Konstruktoren der `FirefoxDriverCreator`-Implementation.

```
public class FirefoxDriverCreator extends AbstractWebDriverCreator {
  /**
   * Erzeugt einen Creator.
   */
  public FirefoxDriverCreator() {...}

  /**
   * Erzeugt einen Creator.
   *
   * @param firefoxBinary
   *     {@link FirefoxDriver#FirefoxDriver(FirefoxBinary, FirefoxProfile)}
   * @param firefoxProfile
   *     {@link FirefoxDriver#FirefoxDriver(FirefoxBinary, FirefoxProfile)}
   */
  public FirefoxDriverCreator(FirefoxBinary firefoxBinary,
                              FirefoxProfile firefoxProfile) {...}

  /**
   * Erzeugt einen Creator.
   *
   * @param firefoxProfile
   *            {@link FirefoxDriver#FirefoxDriver(FirefoxProfile)}
   */
  public FirefoxDriverCreator(FirefoxProfile firefoxProfile) {...}

  /**
   * Erzeugt einen Creator.
   *
   * @param capabilities
   *            {@link FirefoxDriver#FirefoxDriver(Capabilities)}
   */
  public FirefoxDriverCreator(Capabilities capabilities) {...}
```

Beispiel 3.6. Konstruktoren FirefoxDriverCreator

Die Konstruktoren für den `InternetExplorerDriverCreator` und den `ChromeDriverCreator` enthalten ebenfalls die Parameter, die auch bei der Erzeugung der entsprechenden WebDriver-Instanz erforderlich sind.

Um eine neue Konfiguration für einen bestehenden WebDriver zu ändern, wird die Methode `WebConfiguration.registerWebDriverCreator(WebDriverCreator creator)` verwendet. Dadurch werden die bestehenden Konfigurationen überschrieben.

3.4.7. Kompatibilitätsmodus Selenium RC und WebDriver

Für die Kommunikation mit den Browserinstanzen bietet checkerberry web die Möglichkeit, zwischen zwei unterschiedliche Implementierungen zu wählen: Selenium RC (Remote Control) oder WebDriver. Da beide Implementierungen unterschiedliche Vor- und Nachteile haben, ist die generelle Festlegung auf eine der beiden Implementierungen nicht sinnvoll. Stattdessen deckt die parallele Verwendung beider Implementierungen eine sehr große Bandbreite an Einsatzszenarien ab. Beide Implementationen werden innerhalb von checkerberry web durch das Interface RemoteControl gekapselt. Dadurch bleiben die Tests und die Modellklassen unabhängig von der konkreten Implementierung. Theoretisch ist somit das Umschalten der Implementierung für jeden Test möglich. Dies kann sinnvoll sein, um eine Reihe von Tests für das Testen von unterschiedlilchen Browsern wiederzuverwenden. Dazu wäre es z.B. denkbar, die Parameter für den Test z.B. Betriebssystem, zu verwendender Browser und konkrete Implementierung der Fernsteuerung von außen (z.B. durch Hudson) zu übergeben. Dann würden immer die gleichen Tests auf unterschiedlichen Umgebungen ausgeführt werden.

In der Praxis funktioniert das einwandfrei, sofern man entweder nur WebDriver oder nur Selenium RC verwendet. Dies liegt daran, dass die Methoden aus RemoteControl in Selenium RC und WebDriver teilweise unterschiedlich implementiert sind. Als Beispiel liefert die Methode `getAttribute` für Selenium RC einen Leerstring und für WebDriver `null` zurück, wenn das entsprechende Attribut nicht vorhanden ist. Um diese Unterschiede auszugleichen, verwendet checkerberry web einen Kompatibilitätsmodus. Der Kompatibilitätsmodus ist standardmäßig aktiviert und führt dazu, dass sich Selenium RC und WebDriver ähnlicher verhalten. Die Einstellung für den Kompatibilitätsmodus kann über die Methode `setCompatibilityMode(boolean compatibilityMode)` in `WebConfiguration` angepasst werden.

3.5. Installation

Die Installation von checkerberry web ist schnell, einfach und erfordert keine Konfiguration. Die Installation besteht aus den folgenden Schritten

1. Einbinden der erforderlichen Bibliotheken und
2. Erzeugen der checkerberry web-Umgebung.

3.5.1. Einbinden der checkerberry web-Bibliotheken

Dieses Kapitel beschreibt, wie die checkerberry web-Bibliotheken über Maven oder direkt in ein konkretes Projekt eingebunden werden können.

3.5.1.1. Einbinden der Bibliotheken über Maven

Das Einbinden der Bibliotheken über Maven ist einfacher als das direkte Einbinden der Bibliotheken. Maven verwaltet die Abhängigkeiten der verschiedenen Bibliotheken und lädt fehlende Bibliotheken bei Bedarf nach.

Zur Einbindung von checkerberry web in ein Maven-Projekt müssen die Abhängigkeiten in der Maven-Konfiguration (*pom.xml*) eingetragen werden. Folgendes Code-Beispiel zeigt die erforderlichen Einträge:

```
<dependencies>
…
  <dependency>
    <groupId>de.conceptpeople.checkerberry</groupId>
    <artifactId>checkerberry-web</artifactId>
    <version>3.2.x</version>
    <scope>test</scope>
  </dependency>
…
<dependencies>
```

Beispiel 3.7. Erforderliche Maven-Abhängigkeiten von checkerberry web

Durch die Angabe der checkerberry web-Komponente werden ebenfalls alle abhängigen Bibliotheken in das Projekt eingebunden. Der Scope *test* besagt, dass die Bibliothek und alle abhängigen Bibliotheken nur für die Testdurchführung benötigt werden.

Die Maven-Artefakte des checkerberry test center können über das öffentliche checkerberry Maven-Repository geladen werden (siehe Abschnitt 1.4, „Checkerberry Maven-Repository").

3.5.1.2. Checkerberry web-Bibliotheken

Die folgende Tabelle enthält alle checkerberry web-Bibliotheken.

Tabelle 3.2. Bibliotheken von checkerberry web

checkerberry-web-3.2.x.jar	Klassen von checkerberry web

checkerberry-main-3.2.x.pom	Enthält als Eltern-Projekt aller checkerberry-Projekte wichtige Maven-Einstellungen.
checkerberry-common-3.2.x.jar	Allgemeine Klassen für alle checkerberry-Projekte
checkerberry-test-connector-3.2.x.jar	Schnittstelle für die Abstraktion des zu verwendendes Testframeworks.
checkerberry-test-connector-switch-3.2.x.jar	Ermittelt das aktuelle Testframework und bindet die konkrete Instanz des Test-Connectors ein.
checkerberry-junit3-connector-3.2.x.jar	Klassen für die Verwendung von JUnit3. Diese Bibliothek wird automatisch von dem Test-Connector-Switch eingebunden.
checkerberry-junit4-connector-3.2.x.jar	Klassen für die Verwendung von JUnit4. Diese Bibliothek wird automatisch von dem Test-Connector-Switch eingebunden.
checkerberry-testng-connector-3.2.x.jar	Klassen für die Verwendung von TestNG. Diese Bibliothek wird automatisch von dem Test-Connector-Switch eingebunden.

3.5.2. Erzeugen der checkerberry web-Umgebung

Für die Entwicklung und Durchführung der automatisierten Integrationstests ist die Erzeugung einer checkerberry web-Umgebung erforderlich. Die checkerberry web-Umgebung initialisiert die Fernsteuerung und stellt die Rahmenbedingungen für die GUI-Tests her. Dazu gehört neben dem Starten von Proxy-Servern auch die Verwaltung der Browser-Instanzen.

Die Grundeinstellungen von checkerberry web sind so gewählt, dass in der Regel kein Konfigurationsaufwand für die Installation erforderlich ist.

Checkerberry web greift regelmäßig auf den Namen der aktuellen Testklasse und der aktuellen Testmethode zurück. Die Ermittlung der Information, welcher Test gerade ausgeführt wird, ist abhängig von dem verwendeten Testframework. Während die Information in JUnit3-Umgebungen direkt verfügbar ist, muss sie in JUnit4- und TestNG-Umgebungen erst ermittelt werden.

Zur Bestimmung des Namens der aktuellen Testmethode verwendet checkerberry web die gleichen Mechanismen wie checkerberry db (siehe Abschnitt 2.5.3.1, „Aufrufen der setUp-Methode"). Im Folgenden wird die Erzeugung der checkerberry web-Umgebung unter Verwendung von JUnit3 beschrieben.

3.5.2.1. Checkerberry web-Umgebung

Bevor ein Test unter Verwendung von checkerberry web ausgeführt wird, muss eine checkerberry web-Umgebung vorhanden sein. Aus diesem Grund erfolgt die Erzeugung der Umgebung in der Setup-Phase des Tests. In der Regel ist es sinnvoll, diesen Schritt in einer abstrakten Oberklasse vorzunehmen. Alle GUI-Tests können dann von dieser Klasse erben, ohne selbst die Initialisierung der Umgebung vornehmen zu müssen.

Die checkerberry web-Umgebung wird analog zu checkerberry db über einen Service Locator erzeugt. Der Service Locator ist in dem Singleton `CheckerberryWeb` implementiert. Das folgende Code-Beispiel beschreibt die Erzeugung der Umgebung unter JUnit3.

```
public void setUp() throws Exception {

  // Erzeugen einer neuen (Standard-)Konfiguration.
  WebConfiguration configuration =
    new DefaultWebConfiguration();
  // Umgebung erzeugen.
  webEnvironment = CheckerberryWeb.getInstance().getEnvironment(
    configuration);
  // Umgebung für den Test initialisieren.
  webEnvironment.setUp(this);
}
```

Beispiel 3.8. Erzeugung der checkerberry web-Umgebung mit JUnit3

Zunächst wird eine neue `WebConfiguration` erzeugt. In diesem Fall wird die Standard-Konfiguration verwendet. Bei Bedarf können Parameter in der Standard-Konfiguration gesetzt werden. Alternativ kann auch eine eigene Konfiguration verwendet werden.

Über `CheckerberryWeb` wird eine neue Umgebung mit dieser Konfiguration erzeugt und über die Methode `setUp` initialisiert. In der `setUp`-Methode startet checkerberry web eine neue Browser-Instanz und navigiert zu der konfigurierten URL. Wenn die zugehörige Webseite vollständig geladen wurde, ist die Setup-Phase beendet und die Testmethode wird aufgerufen.

Nach dem Test wird in der Teardown-Phase die Methode `tearDown` der checkerberry web-Umgebung aufgerufen. Das folgende Code-Beispiel zeigt eine typische Implementierung.

```
public void tearDown() throws Exception {
  webEnvironment.tearDown();
  webEnvironment = null;
}
```

Beispiel 3.9. Beenden der checkerberry web-Umgebung mit JUnit3

3.5.2.2. checkerberry web-Umgebung unter Verwendung anderer Testframeworks

Selbstverständlich lässt sich checkerberry web auch in anderen Testframeworks als den genannten einsetzen. Die Ermittlung des Namens der aktuellen Testmethode ist von Framework zu Framework unterschiedlich. Das checkerberry test center stellt für die Ermittlung des Namens keine Funktionalität bereit, sondern erwartet die Übergabe des Namens in der `setUp`-Methode:

```
webEnvironment.setUp(this, "testMethod");
```

3.5.2.3. Konfiguration von checkerberry web

Die Konfiguration von checkerberry web ist optional. Die Grundeinstellungen sind so gewählt, dass in der Regel keine Anpassungen erforderlich sind. Durch das Implementieren des Interfaces `WebConfiguration` und der Verwendung dieser Klasse bei der Erzeugung der checkerberry web-Umgebung, kann die Konfiguration angepasst werden. Wenn keine eigene Konfiguration angegeben wird, wird die Klasse `DefaultWebConfiguration` verwendet. Die in diesem Kapitel dargestellten Standard-Konfigurationen beziehen sich daher auf diese Klasse.

In der checkerberry web-Konfiguration werden globale Einstellungen vorgenommen, die sich auf alle oder zumindest einen großen Anteil der Tests beziehen. Neben der Konfiguration steht mit dem Web-Kontext eine Möglichkeit bereit, Konfigurationen für einzelne Tests anzupassen. Für jeden Test wird ein neuer Web-Kontext erzeugt, sodass Änderungen an dem Kontext keine Auswirkungen auf folgende Tests nach sich ziehen.

Checkerberry web-Konfiguration

Im Folgenden werden die Methoden der checkerberry web-Konfiguration beschrieben.

void configure(RemoteControlConfiguration)

Innerhalb dieser Methode erfolgt die Konfiguration von Selenium RC über die *RemoteControlConfiguration*. Alle Einstellungen der *RemoteControlConfiguration* können in dieser Methode angepasst werden. In der Standard-Konfiguration werden folgende Einstellungen gesetzt:

- Timeout in seconds: 2.000
- Port: 41444
- Trust all SSL certificates: true.

void configure(RemoteControl)

Innerhalb dieser Methode erfolgt die Konfiguration der Selenium-Fernsteuerung. In der Standard-Konfiguration werden folgende Einstellungen gesetzt:

- Timeout: Für den Zugriff von Selenium auf HTML-Komponenten wird ein Timeout definiert, der die maximale Ausführungsdauer dieses Zugriffs festlegt. Dieser Timeout kann hier in Millisekunden angelegt werden (Default 2.000ms).
- Speed: Dieser Wert gibt die Anzahl in Millisekunden an, die vor der Ausführung eines Selenium-Requests gewartet wird. Als Default ist dieser Wert auf 0 gesetzt. Eine Erhöhung des Werts führt zu einer längeren Laufzeit der Tests, was insbesondere zu Präsentations- oder Debug-Zwecken gewünscht sein kann. Dieser Wert hat bei der Verwendung von WebDriver keine Auswirkungen.

String getBrowserType()

Über diese Methode wird der Browser konfiguriert, der zum Testen verwendet werden soll. Die vollständige Liste der unterstützen Browser finden Sie in der FAQ auf der Selenium-Webseite unter [SeleniumRC FAQ, 2010]. Am häufigsten werden folgende Einstellungen verwendet:

- `*firefox`: Startet den Firefox-Browser aus dem Standard-Installationsverzeichnis. Diese Einstellung ist in checkerberry web als Default ausgewählt.
- `*iexplore`: Startet den Internet Explorer aus dem Standard-Installationsverzeichnis.
- `*googlechrome`: Startet den Google-Chrome-Browser aus dem Standard-Installationsverzeichnis.
- `*custom`: Startet einen manuell konfigurierten Browser. Bei der Angabe dieses Browser-Typs muss zusätzlich ein Installationspfad angegeben werden z.B. *`*custom C:\Program Files\Mozilla Firefox\firefox.exe`*. Checkerberry web startet dann den entsprechenden Browser.

String getServerHost()

Diese Methode liefert den Namen oder die IP-Adresse des Rechners zurück, auf dem der HTTP-Proxy angesprochen werden soll. In der Standard-Konfiguration ist das „localhost".

int getServerPort()

Diese Methode liefert den Port des HTTP-Proxies zurück. In der Standard-Konfiguration ist das 41444.

int getTimeout()

Diese Methode liefert den Timeout in Millisekunden zurück, der innerhalb von Selenium verwendet werden soll. In der Standard-Konfiguration ist das 2.000ms.

double getScreensaverNonZeroBytesRatio()

Eine Anpassung dieser Methode ist in der Regel nicht erforderlich. Checkerberry web ermittelt anhand eines Screenshots, ob der Bildschirmschoner aktiv ist. In diesem Fall enthält der Screenshot vornehmlich Null-Bytes. Über diese Methode kann das Verhältnis von nicht Null-Bytes zur Gesamtanzahl der Bytes eingestellt werden. Die Standard-Konfiguration ist 0,05, d.h. wenn mindestens 5% der Bytes keine Null-Bytes sind, handelt es sich um einen „echten" Screenshot. Umgekehrt bedeutet dies, dass der Screenshot einen schwarzen Bildschirm aufgenommen hat, wenn über 95% der Bytes Null-Bytes sind.

int getScreensaverDetectionInterval()

Die Ermittlung eines aktiven Bildschirmschoners ist zeitaufwendig. Aus diesem Grund kann das Ergebnis der Überprüfung für einige Zeit gespeichert und wiederverwendet werden. Über diese Methode wird definiert, wie lange die ermittelte Information wiederverwendet werden kann. Eine erneute Auswertung des Screenshots wie oben beschrieben ist dann nicht erforderlich. Das Intervall wird in Millisekunden angegeben. Der Standardwert ist 5.000ms.

NewBrowserInstancePolicy getNewBrowserInstancePolicy()

Durch die Methode `getNewBrowserInstancePolicy()` kann das Verfahren für die Ermittlung, wann eine neue Browser-Instanz zu verwenden ist, geändert werden (siehe Abschnitt 3.4.1, „Wiederverwendung von Browser-Instanzen").

String getTestUrl()

Die Test-URL definiert, welche URL innerhalb der Setup-Phase in der Browser-Instanz geladen werden soll. Wenn der Großteil der Tests die gleiche Test-URL verwenden, kann diese in der Konfiguration angegeben werden. Eine weitere Möglichkeit besteht in der Angabe der Test-URL über die Annotation `@TestUrl` (siehe Abschnitt 3.4.2, „Festlegen der Test-Url").

String getLogoutUrl()

Bei der Wiederverwendung von Browser-Instanzen muss die letzte Session des Browsers gelöscht werden, damit der vorherige Test keine Auswirkungen auf einen folgenden Test hat. Anderenfalls kann es z.B. vorkommen, dass ein Test die Anzeige einer Login-Seite erwartet, die jedoch nicht dargestellt wird, da der vorherige Test bereits ein Login durchgeführt hat. Wird eine Browser-Instanz wiederverwendet, löscht checkerberry web alle Session-Cookies. Es kann jedoch zusätzlich erforderlich sein, dass eine URL aufgerufen werden muss, um die Session im Webserver zu beenden. Zu diesem Zweck kann eine Logout-URL definiert werden, die vor einer Wiederverwendung der Browser-Instanz aufgerufen wird.

WebContext createContext(TestConnector)

Der Web-Kontext enthält Informationen zur Durchführung der aktuellen Testmethode. Für jede Testmethode wird ein neuer Kontext über die checkerberry web-Konfiguration erzeugt. Dieses Vorgehen ermöglicht die Verwendung projektspezifischer Kontexte, die weitere Informationen beinhalten. Als Standard wird die Klasse *DefaultWebContext* als Kontext verwendet.

Web-Kontext

Der Web-Kontext beinhaltet Informationen, die lediglich für die Dauer einer Testmethode gültig sind. Im Folgenden werden die Methoden des Interfaces *WebContext* beschrieben.

TestConnector getTestConnector()

Diese Methode liefert den aktuellen Test-Connector zurück. Der Test-Connector definiert zum einen, welches Testframework verwendet wird. Diese Information ist bei der Implementierung eines konkreten Tests jedoch bereits bekannt. Zum anderen stellt der Test-Connector den Namen der aktuellen Testklasse und -methode über die Methoden `getClassName` und `getMethodName` bereit.

`WebConfiguration getConfiguration()`

Diese Methode liefert die aktuelle checkerberry web-Konfiguration zurück. Der Web-Kontext ermöglicht das Überschreiben von Einstellungen aus der checkerberry web-Konfiguration. Wenn ein Wert im Kontext nicht überschrieben wurde, wird der entsprechende Wert aus der Konfiguration verwendet.

`void setTestUrl(String)` und `String getTestUrl()`

Diese Methoden sind Setter und Getter für die zu verwendende Test-URL. Es ist somit möglich die Test-URL für eine Testklasse oder -methode anzupassen.

`void setLogoutUrl(String)` und `String getLogoutUrl()`

Diese Methoden sind Setter und Getter für die zu verwendende Logout-URL. Es ist somit möglich die Logout-URL für eine Testklasse oder -methode anzupassen.

`void setNewBrowserInstancePolicy(NewBrowserInstancePolicy)` und `NewBrowserInstancePolicy getNewBrowserInstancePolicy()`

Diese Methode sind Setter und Getter für das Verfahren zur Ermittlung, für welche Tests eine neue Browser-Instanz verwendet werden muss. Das Verfahren kann für einzelne Testklassen über den Kontext angepasst werden. Eine detaillierte Beschreibung des Verfahrens ist im Rahmen der checkerberry web-Konfiguration im vorherigen Abschnitt vorhanden.

3.6. Erstellen von Tests

Das Erstellen von GUI-Tests mit checkerberry web wird am Beispiel einer einfachen Login-Seite verdeutlicht. Die folgende Grafik stellt die zu testende Login-Seite dar.

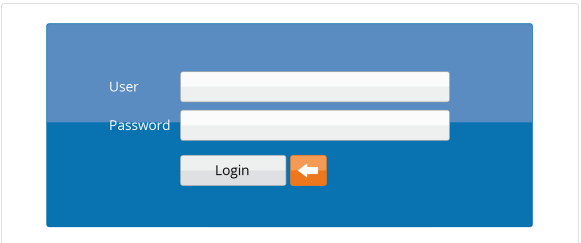


Abbildung 3.6. Login-Maske

Das zu testende HTML-Formular verfügt im Wesentlichen über drei relevante Komponenten: ein Eingabefeld für den Namen, ein Eingabefeld für das Passwort und einen Login-Button zum Absenden des Requests. Die zugehörige HTML-Datei sieht wie folgt aus:

```
<html>
  <form action="/login" method="get">
    <table>
      <tr>
        <td>User</td>
        <td><input name="user" id="userId" type="text"></td>
      </tr>
      <tr>
        <td>Password</td>
        <td><input name="password" id="passwordId" type="password"></td>
      </tr>
      <tr>
        <td> </td>
        <td><input id="loginId" value="Login" type="submit"></td>
      </tr>
    </table>
  </form>
</html>
```

Beispiel 3.10. login.html

Bevor der eigentliche Test implementiert werden kann, muss ein Modell für die zu testende Webseite erstellt werden. Das Modell enthält Getter-Methoden für alle Komponenten der Webseite, die durch den Test angesprochen werden sollen. Die Methoden liefern eine Instanz der Klasse *RemoteControlComponent* zurück. Des Weiteren enthält das Modell fachliche Methoden, die in verschiedenen Tests wiederverwendet werden können. Weitere Details zum Modellansatz finden sie in Abschnitt 3.7, „Modellansatz".

Das folgende Beispiel enthält das Modell für die Login-Seite.

```
public class LoginPage extends AbstractRemoteControlPage {

  // Konstuktor mit dem ContextProvider.
  public LoginPage(ContextProvider contextProvider) {
    super(contextProvider);
  }

  // Fernsteuerung für eine HTML-Komponente, das User-Id-Feld,
  // zurückliefern
  public RemoteControlComponent getUserTextField() {
    // Für die Erzeugung wird nur die HTML-ID benötigt.
    return createComponentProxy("userId");
  }

  // Fernsteuerung für eine HTML-Komponente, das Passwort-Feld,
  // zurückliefern
  public RemoteControlComponent getPasswordTextField() {
    // Für die Erzeugung wird nur die HTML-ID benötigt.
    return createComponentProxy("passwordId");
  }

  // Fernsteuerung für eine HTML-Komponente, den Login-Button,
  // zurückliefern
  public RemoteControlComponent getLoginButton() {
    // Für die Erzeugung wird nur die HTML-ID benötigt.
    return createComponentProxy("loginId");
  }
}
```

Beispiel 3.11. LoginPage.java

Die Klasse `LoginPage` erbt von der `AbstractRemoteControlPage`, die durch checkerberry web zur Verfügung gestellt wird. Diese Klasse stellt Basis-Funktionen für die Seite zur Verfügung. Der Konstruktor nimmt einen `ContextProvider` entgegen, der die Fernsteuerung (das `RemoteControl`-Objekt), den Kontext und die Konfiguration beinhaltet und reicht die Werte an die abstrakte Seite weiter. Das `RemoteControl`-Objekt bildet die Schnittstelle des Tests zu Selenium.

Für die Erstellung der Komponenten-Getter ist lediglich ein sogenannter Locator erforderlich. Ein Locator definiert, wie eine Komponente auf der Webseite eindeutig gefunden werden kann. In dem konkreten Login-

Beispiel werden dazu die HTML-IDs verwendet. Mehr ist bei der Erstellung eines Modells nicht zu tun. Nachdem die Erstellung des Modells abgeschlossen ist, kann mit der Erstellung des Tests begonnen werden.

```
package de.conceptpeople.demo;

@TestUrl("http://localhost:8080/login.html")
public class LoginTest extends AbstractCheckerberryWebTest {

  public void testLogin() {

    // Erstellen des Modells für die Login-Seite. Der Test selbst
    // ist ein ContextProvider.
    LoginPage loginPage = new LoginPage(this);

    // Prüfen, dass User- und Passwort-Feld nicht vorbelegt sind.
    assertEquals("", loginPage.getUserTextField().getValue());
    assertEquals("", loginPage.getPasswordTextField().getValue());

    // Fernsteuerung für das User-Feld holen und „Homer“ eintragen.
    loginPage.getUserTextField().type("Homer");
    // Fernsteuerung für das Passwort-Feld holen und „Duff“ eintragen.
    loginPage.getPasswordTextField().type("Duff");
    // Fernsteuerung für den Login-Button holen, Button drücken und
    // warten bis die neue Seite geladen wurde.
    loginPage.getLoginButton().clickAndWaitForPage();
  }
}
```

Beispiel 3.12. Test-Implementierung der Login-Seite

Obiges Code-Beispiel zeigt eine mögliche Umsetzung. Der Test erbt von der abstrakten Klasse `AbstractCheckerberryWebTest`, in der die Initialisierung von checkerberry web erfolgt. In der Klasse `LoginTest` wird über die Annotation `@TestUrl` die Test-URL auf den Wert http://localhost:8080/login.html gesetzt.

Bei der Ausführung des Tests wird zunächst in der Setup-Phase eine Browser-Instanz gestartet, die die URL `http://localhost:8080/login.html` aufruft. In der Test-Phase wird die Methode `testLogin` aufgerufen. Dort wird zunächst das Modell der Login-Seite erzeugt, um auf die Komponenten der Login-Seite zugreifen zu können. Dann wird geprüft, ob die Werte der Eingabefelder für den User und das Passwort beide nicht belegt sind. Danach wird in das Eingabefeld für den User der Wert „Homer“ und als Passwort „Duff“ eingetragen. Die Eingaben werden dann über den Login-Button abgesendet.

Nach dem Test wird die Browser-Instanz in der Teardown-Phase wieder beendet.

Auf den ersten Blick überprüft der Test nur die initiale Belegung der Eingabefelder. Bei der Verwendung von checkerberry web muss man jedoch zwischen expliziten und impliziten Prüfungen unterscheiden. Der Aufruf `getValue` prüft zum Beispiel implizit, ob die zugehörige Komponente in der Webseite vorhanden ist und ob dort die Methode `getValue` aufgerufen werden kann. Sollte dies nicht der Fall sein, wird eine Exception geworfen, die die Ausführung des Tests beendet. Dieser Sachverhalt ermöglicht das einfache Happy-Path-Testen. Bei einem Happy-Path-Test werden Funktionalitäten einer Web-Anwendung mit klar definierten Eingabewerten getestet, wobei keine Fehlersituationen erwartet oder überprüft werden.

```
package de.conceptpeople.demo;

@TestUrl("http://localhost:8080/login.html")
public class LoginTest extends AbstractCheckerberryWebTest {

  public void testLogin() {

    // Erstellen des Modells für die Login-Seite. Der Test selbst
    // ist ein ContextProvider.
    LoginPage loginPage = new LoginPage(this);

    // Es ist sinnvoll, fachliche Aspekte in einzelne Methoden
    // auszulagern, um diese in vielen Tests wiederzuverwenden.
    loginPage.performLogin("Homer", "Duff");

    // Wenn Homer sogar der Standard-User ist:
    loginPage.performDefaultLogin();

    // Gerade bei größeren Formularen ist das Builder-Pattern
    // auch ein sinnvoller Ansatz
    loginPage.enterUser("Homer").enterPassword("Duff")
      .clickAndWaitForPage();
  }
}
```

Beispiel 3.13. Test-Implementierung der Login-Seite (erweitert)

Das obige Beispiel zeigt die Verwendung eines erweiterten Modells. Durch die Auslagerung von fachlichen Methoden in die Modelle, sind diese in vielen Tests wiederverwendbar. Das Builder-Pattern kann dabei sehr hilfreich sein, wenn große Formulare existieren und man immer nur eine Teilmenge der Felder eingeben möchte. Die Lesbarkeit bleibt durch die Verwendung des Builder-Patterns sehr hoch.

3.7. Modellansatz

In den vorherigen Kapiteln wurde bereits erwähnt, dass checkerberry web einen Modellansatz der zu testenden Webseiten verwendet. In diesem Abschnitt wird die Motivation für diesen Ansatz erläutert.

Auf der Selenium-Webseite [Selenium Homepage, 2010] wird empfohlen, das Teilprodukt Selenium IDE als Einstiegspunkt für Selenium zu verwenden. Die Selenium IDE ist ein Firefox-Plug-in, das die Interaktionen des Benutzers mit der aktiven Webseite in verschiedenen Dialekten z.B. als JUnit-Source-Code aufzeichnet. Dieser Source-Code kann als Basis für einen JUnit-Test verwendet werden. Dieser Weg führt auf lange Sicht jedoch in eine Sackgasse. Schauen wir uns dazu das vorherige Beispiel der Login-Seite auf Basis der Selenium IDE an. Die Aufzeichnung über die Selenium IDE erzeugt folgenden Source-Code:

```
// In das Feld mit der ID userId wird der Wert ‚Homer' eingetragen.
selenium.type("userId", "Homer");
// In das Feld mit der ID passwordId wird der Wert ‚Duff' eingetragen.
selenium.type("passwordId", "Duff");
// Die Komponente mit der ID loginId wird gedrückt.
selenium.click("loginId");
// Warten auf das Laden der neuen Seite mit einem Timeout von 30.000ms.
selenium.waitForPageToLoad("30000");
```

Beispiel 3.14. Test-Aufzeichnung mit Selenium IDE

Auf den ersten Blick sieht das Ergebnis ähnlich zu dem bereits geschriebenen Test aus. Als erstes fällt auf, dass in dem Test direkt die Locatoren der Komponenten (z.B. `userId`) verwendet werden. Das ist für sich alleine betrachtet unproblematisch. Was passiert allerdings, wenn zahlreiche Tests implementiert wurden? In dem Fall werden viele Locatoren in vielen Tests dupliziert, was große Probleme verursachen kann, wenn es Änderungen an der Webseite gibt. Die Locatoren werden innerhalb der JUnit-Tests als Strings angegeben, wobei das Refactoring von Texten in Java nur über „Suchen & Ersetzen" möglich ist. Dieses Vorgehen birgt jedoch ein erhebliches Fehlerpotential. Das Naheliegende ist in dieser Situation die Verwendung von Konstanten: Anstatt die Locatoren direkt zu verwenden, wird für jeden Locator eine Konstante eingefügt. Das

umgeht das Problem, allerdings steigt dadurch der manuelle Aufwand bei der Verwendung der Selenium IDE.

Weiterhin fällt auf, dass der Timeout von „30000" Millisekunden hartkodiert in den Test aufgenommen wird. Auch das ist nicht wünschenswert, da dieser Wert ebenfalls zahlreich in verschiedenen Tests dupliziert wird.

Der programmierte Test verwendet die Methode `clickAndWaitForPage`, während dieser Aufruf bei dem aufgezeichneten Test zwei Zeilen *`click`* und *`waitForPageToLoad`* erfordert. Dieser Sachverhalt zeigt die größere Flexibilität, die der Modellansatz bietet. Das Modell kann beliebig an die Besonderheiten der zu testenden Web-Anwendung angepasst werden, indem z.B. neue Methoden erstellt werden, die generelle Aspekte zentral behandeln.

Das Login-Beispiel ist gut geeignet, um einen Einstieg in das Schreiben von automatisierten GUI-Tests zu liefern. In der Praxis sind die zu testenden Webseiten jedoch meist komplexer.

Das Login-Beispiel verwendet HTML-IDs um die Komponenten zu definieren. Das ist der beste Weg, um Selenium zu verwenden. Es kommt jedoch auch vor, dass nicht alle zu testenden Komponenten eine HTML-ID besitzen. In diesen Fällen kann der Locator XPath oder CSS-Pfade verwenden, um die Eindeutigkeit zu gewährleisten. Das Ergebnis einer Aufzeichnung sieht dann z.B. so aus:

```
// In das Feld zu dem XPath wird der Wert ‚Homer' eingetragen.
selenium.type("xpath=//tr/td/div/input", "Homer");
// In das Feld mit der CSS-Beschreibung wird der Wert ‚Duff' eingetragen.
selenium.type("css=in32", "Duff");
// Die Komponente zu dem XPath wird gedrückt.
selenium.click("xpath=//input[@value='Login']");
// Warten auf das Laden der neuen Seite mit einem Timeout von 30.000ms.
selenium.waitForPageToLoad("30000");
```

Beispiel 3.15. Test-Aufzeichnung mit Selenium IDE und kryptischen Locatoren

In dem Beispiel sieht man sehr gut, dass die Lesbarkeit der Tests stark abnimmt, wenn die Locatoren kryptisch sind. Stellen Sie sich jetzt einen umfangreichen Test mit ca. 30 Zeilen vor. Der Test ist nicht mehr lesbar und daher kaum noch wartbar. Im Gegensatz dazu werden im Modellansatz die kryptischen Locatoren in den Modellen gekapselt, sodass der Test nach wie vor lesbar bleibt.

Bei der Entwicklung von Portalen ergibt sich ein weiteres Problem, das durch die Selenium IDE nicht adressiert ist: Die HTML-IDs der Komponenten sind dynamisch!

Per Definition können in einem Portal mehrere Portlets auf einer HTML-Seite angezeigt werden – insbesondere also auch gleiche Portlets. Um die Eindeutigkeit der HTML-IDs zu gewährleisten, werden die IDs manipuliert, indem z.B. jede Portlet-Instanz ein Präfix verwendet, das vor jede HTML-ID gehängt wird. Aus der ID *`loginId`* wird so z.B. die ID *`P4711:loginId`*., wobei das Präfix *`P4711`* dynamisch ist. Statische Locatoren können jetzt nicht mehr verwendet werden.

Der Aufzeichnungsansatz stößt spätestens hier an seine Grenzen. Da der Aufzeichnungsansatz keine Abstraktion der Locatoren verwendet, muss die erforderliche Funktionalität zur Dynamisierung der Locatoren direkt in den Tests implementiert werden. Im Modellansatz lässt sich die erforderliche Logik hingegen in den Modellklassen kapseln. Ein allgemeiner Lösungsweg ist unter Abschnitt 6.3.2, „Wie kann ich Portal-Anwendungen testen?" beschrieben.

Die Vorteile des Modellansatzes basieren auf der guten Abstraktion der zu testenden Webanwendung. Innerhalb des Modells lässt sich generelle Funktionalität kapseln und verstecken, die sich auf alle Tests bezieht. Die Tests bleiben dadurch lesbar und wartbar.

Der Modellansatz bildet die Basis für die Entwicklung von nachhaltigen GUI-Tests. Änderungen an der Webanwendung können leicht an den Modellklassen nachgezogen werden. Änderungen von Locatoren werden direkt in den Modellklassen korrigiert. Verschiebungen von HTML-Komponenten von einer Webseite in eine andere Webseite werden ebenfalls im Modell angepasst. Dadurch entstehen Compiler-Fehler, die den Entwickler direkt zu den anzupassenden Tests führen.

Da es sich bei den Modellklassen um Java-Klassen handelt, steht dem Entwickler die gesamte Funktionalität der Entwicklungsumgebung für die Bearbeitung der Modelle zur Verfügung. Umbenennungen von Modellkomponenten werden über Refactorings durchgeführt, sodass alle Tests automatisch aktualisiert werden. Über die Navigation der Entwicklungsumgebung lassen sich schnell alle Tests auffinden, die bestimmte Komponenten oder Webseiten verwenden.

Ein weiterer Vorteil des Modellansatzes liegt in der hohen Effizienz bei der Testerstellung. Die Entwicklung der Modellklassen ist schnell, da deren Struktur sehr einfach ist. Sobald die Modelle der Webseiten vorhanden sind, kann der Entwickler die GUI-Tests wie normalen Source-Code programmieren. Der Entwickler bewegt sich somit auf vertrautem Terrain, sodass die Testerstellung sehr schnell geht.

Der Modellansatz bietet zahlreiche Vorteile im Vergleich zum Aufzeichnungsansatz. Daher verwendet checkerberry web den Modellansatz für die effiziente Erstellung von nachhaltigen GUI-Tests.

Der Modellansatz ist mittlerweile unter dem Namen Page Object Pattern [Google Page Objects, 2010] bekannt und als Best-Practice–Ansatz für GUI-Tests etabliert.

Kapitel 4. Checkerberry business view

4.1. Einleitung

Scrum ist heutzutage das Standardvorgehen in agilen Softwareprojekten. Die Anforderungen werden dabei häufig in Form von sogenannten User Stories beschrieben. Eine User Story beinhaltet dabei die Informationen, wer was warum möchte. Z.B. als Sachbearbeiter möchte ich mich an dem System anmelden, damit ich nur Daten sehe, für die ich legitimiert bin. Ein weiteres Vorgehen, was sich immer häufiger etabliert, ist die Akzeptanztestgetriebene Entwicklung (ATDD – Acceptance Test Driven Development). Dabei werden innerhalb der User Story fachliche Akzeptanzkriterien definiert, die für die Abnahme der Funktionalität durch die Fachabteilungen zwingend erforderlich sind. Z.B. ist das Anmelden an das System bei der Eingabe eines falschen Passworts nicht möglich. Diese Kriterien werden in Form von konkreten Beispielen als Akzeptanztests in den User Stories festgehalten.

Mit dem checkerberry test center hat der Benutzer die Möglichkeit, die definierten Akzeptanztests automatisiert zu testen. Durch die Automatisierung kann so zum einen sichergestellt werden, dass die erforderlichen Akzeptanzkriterien erfüllt sind. Zum anderen werden diese Tests auch in Regressionstests verwendet, sodass permanent die korrekte fachliche Funktionsweise der gesamten Anwendung sichergestellt werden kann.

Das Hudson-Plugin checkerberry business view stellt die Verbindung zwischen den fachlichen Akzeptanztests und den automatisierten Tests her. Das Plugin stellt den Fortschritt bei der Erstellung der Akzeptanztests als Trend dar, sodass der Projektfortschritt auch für den ProductOwner und die Fachabteilungen transparent wird.

Der fachliche Projektfortschritt kann neben der Anzeige im Hudson auch als eigenständiger HTML-Report erzeugt werden.

4.2. Ansichten des checkerberry business views

Erweitert man Hudson um das `checkerberry-atdd-hudson-plugin` stehen innerhalb des Hudson neue Ansichten zur Verfügung. In den beiden folgenden Abschnitten werden zwei Ansichten beschrieben, die je nach gewählter Job-Konfiguration sowohl auf der Gesamtprojektebene als auch für jedes einzelne Maven-Modul verfügbar sind. Dabei unterscheiden sich die Darstellungen für die beiden unterschiedlichen Ebenen, zwar in deren Inhalt aber nicht in deren Aufbau. Daher wird die Darstellung nur einmal erläutert.

Im dritten Abschnitt 4.2.3, „Dashboard View", wird ein Portlet vorgestellt, das eine Übersicht über alle Projekte liefert, zu denen Akzeptanztestresultate existieren. Das Portlet stellt auf einen Blick den Status der Projekte, bezogen auf deren Akzeptanztests, dar.

Im vierten Abschnitt 4.3.3, „HTML-Maven-Plugin" wird die Anzeige im eigenständige HTML-Report beschrieben.

4.2.1. Übersichtsseite

Nach der Installation des checkerberry business views wird im Hudson auf der Übersichtsseite eines Projekts bzw. Moduls der User Story Trend der letzten Builds angezeigt.[1] Anhand des User Story Trends wird

[1] Setzt auf Gesamtprojektebene die Konfiguration des "Publish combined Acceptance-Test Report" voraus.

ersichtlich, wie sich die Implementierung der zugehörigen Akzeptanztests von Build zu Build verändert. Dabei wird unterschieden, ob Akzeptanztests noch "nicht implementiert" wurden (grau dargestellt), "fehlgeschlagen" sind (rot dargestellt) oder "erfolgreich" durchlaufen wurden (grün dargestellt). Über eine User Story kann in der Regel keine genaue Aussage getroffen werden, wenn ein oder mehrere Tests dieser User Story nicht ausgeführt wurden. Daher erhält diese User Story den Status "nicht gelaufen" (gelb dargestellt). Die folgende Abbildung zeigt einen Beispieltrend.

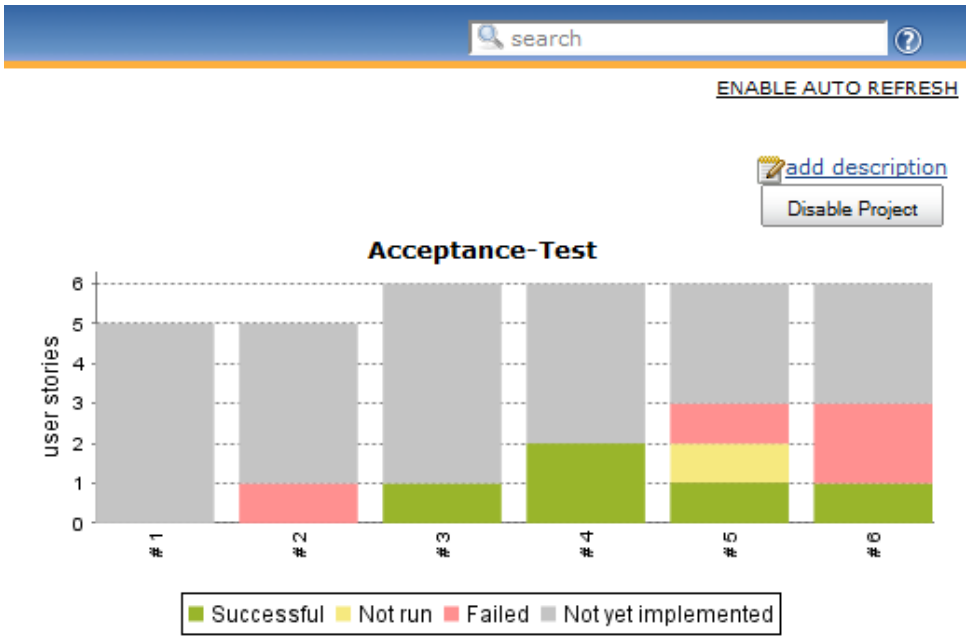


Abbildung 4.1. User Story Trend in der Übersicht

In dem Beispiel wurde zu Beginn der Entwicklung fünf User Stories formuliert. Im ersten Build wurde für keine der User Stories ein Akzeptanztest umgesetzt. Im zweiten Build wurde für eine der User Stories mindestens ein Akzeptanztest implementiert. Allerdings schlägt mindestens einer der Tests fehl, wodurch die User Story ebenfalls den Status "fehlgeschlagen" besitzt. Aus dem Beispieltrend wird zudem ersichtlich, dass beim dritten Build eine weitere User Story hinzugefügt wurde. Mit dem vierten Build sind bereits zwei der sechs User Stories erfolgreich umgesetzt. Beim fünften Build wurden nicht alle Tests ausgeführt. Daher kann für eine der User Stories keine genaue Aussage über deren Status getroffen werden. Sowohl beim fünften als auch beim sechsten Build sind für drei User Stories alle zugehörigen Tests implementiert.

Innerhalb des Hudsons verfügt jede Übersichtsseite zu einem Projekt oder Modul über eine Sidebar. Dieser Sidebar wurde eine weitere Verlinkung hinzugefügt, mit der man zur Detailansicht des checkerberry business views gelangt (rot umrandeter Eintrag in Abbildung 4.2, „Erweiterte Sidebar").

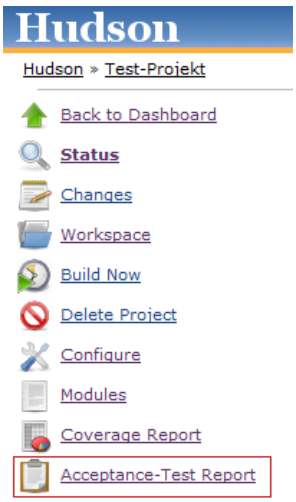


Abbildung 4.2. Erweiterte Sidebar

4.2.2. Detailansicht

In der Detailansicht können zusätzliche Informationen eingesehen werden, die eine genauere Aussage über den Projektfortschritt erlauben. Während die Übersichtsseite stets die Informationen des letzten Builds darstellt, kann die Detailansicht zu jedem aufbewahrten Build betrachtet werden. Im Folgenden wird die Detailansicht exemplarisch dargestellt und erläutert.

Die Detailansicht lässt sich in drei Bereiche unterteilen, wobei am Anfang eine Zusammenfassung der Akzeptanztests dargestellt ist. Ein Akzeptanztest entspricht einer Testmethode, die mit der Annotation `@AcceptanceTest` versehen wurde. Die Menge der Akzeptanztests bildet somit eine Teilmenge aller Testmethoden. Neben der Gesamtanzahl der erfolgreichen, fehlgeschlagenen, nicht gelaufenen und nicht implementierten Akzeptanztests wird zudem die Veränderung zum vorherigen nicht fehlgeschlagenen Build angezeigt. Diese Werte werden hinter der Gesamtanzahl in Klammern angegeben.

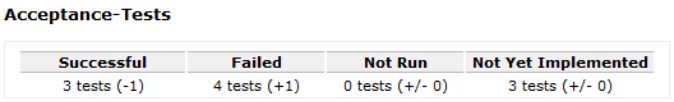


Abbildung 4.3. Änderung der Akzeptanztests

Anders als die Übersichtsseite, welche den Entwicklungsverlauf der User Stories ausgehend vom letzten Build zeigt, wird in der Detailansicht der Trend auf der Grundlage des betrachteten Builds dargestellt. Für jeden aufbewahrten Build, der vor dem betrachteten liegt, ist eine Säule im Säulendiagramm aufgetragen. Aus dieser geht hervor, wie viele User Stories den Akzeptanzteststatus "erfolgreich", "fehlgeschlagen", "nicht gelaufen" oder "nicht implementiert" besitzen. Dabei ist der Akzeptanzteststatus einer User Story nur dann "erfolgreich", wenn alle zugehörigen Akzeptanztests erfolgreich durchgelaufen sind. Eine User Story hat den Status "fehlgeschlagen", wenn mindestens einer der Tests fehlgeschlagen ist. Beinhaltet die User Story mindestens einen nicht gelaufenen Akzeptanztest, so ist die User Story als "nicht gelaufen" markiert. In allen anderen Fällen gilt die User Story als "nicht implementiert".

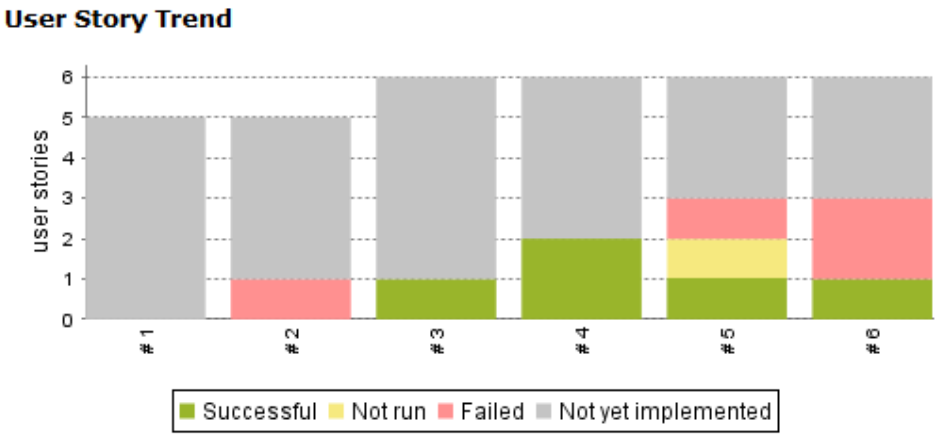
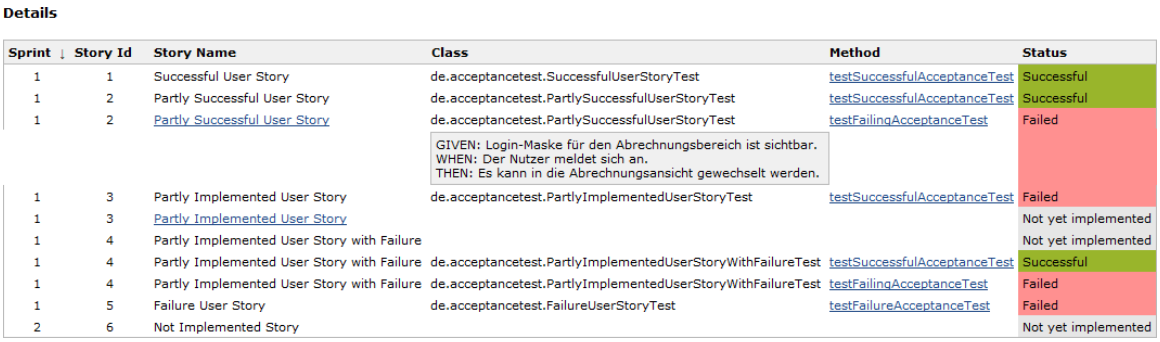


Abbildung 4.4. User Story Trend

Der untere Bereich besteht aus einer detaillierten Auflistung der einzelnen Akzeptanztests. Ein Akzeptanztest ist durch den vollqualifizierten Klassennamen und den Methodennamen eindeutig. In der Tabelle ist ersichtlich, zu welchem Sprint und zu welcher User Story ein Test gehört. In der letzten Spalte ist der Status des Akzeptanztests vermerkt. Ist checkerberry business view korrekt konfiguriert (siehe Hinweis im Abschnitt 4.3.2.2, „Ansichten aktivieren“[119]), kann über die Verlinkungen an den Methodennamen zu den Surefire-Testergebnissen navigiert werden. Der User Story-Name ist ebenfalls eine Verlinkung, wenn zu dem Akzeptanztest eine Beschreibung angegeben wurde. Mit einem Klick auf den Story Namen öffnet sich die Beschreibung.[2]

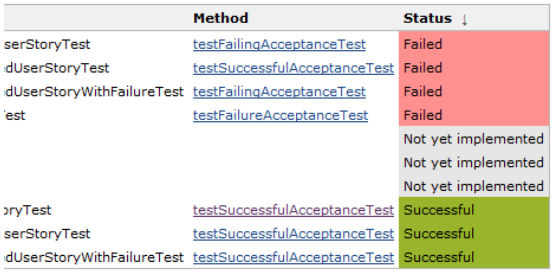


Details

Sprint ↓	Story Id	Story Name	Class	Method	Status
1	1	Successful User Story	de.acceptancetest.SuccessfulUserStoryTest	testSuccessfulAcceptanceTest	Successful
1	2	Partly Successful User Story	de.acceptancetest.PartlySuccessfulUserStoryTest	testSuccessfulAcceptanceTest	Successful
1	2	Partly Successful User Story	de.acceptancetest.PartlySuccessfulUserStoryTest	testFailingAcceptanceTest	Failed
			GIVEN: Login-Maske für den Abrechnungsbereich ist sichtbar. WHEN: Der Nutzer meldet sich an. THEN: Es kann in die Abrechnungsansicht gewechselt werden.		
1	3	Partly Implemented User Story	de.acceptancetest.PartlyImplementedUserStoryTest	testSuccessfulAcceptanceTest	Failed
1	3	Partly Implemented User Story			Not yet implemented
1	4	Partly Implemented User Story with Failure			Not yet implemented
1	4	Partly Implemented User Story with Failure	de.acceptancetest.PartlyImplementedUserStoryWithFailureTest	testSuccessfulAcceptanceTest	Successful
1	4	Partly Implemented User Story with Failure	de.acceptancetest.PartlyImplementedUserStoryWithFailureTest	testFailingAcceptanceTest	Failed
1	5	Failure User Story	de.acceptancetest.FailureUserStoryTest	testFailureAcceptanceTest	Failed
2	6	Not Implemented Story			Not yet implemented

Abbildung 4.5. Detailansicht der Akzeptanztests

Mit einem Klick auf den Spaltennamen lässt sich die Sortierreihenfolge der Tabellendaten verändern. Auf diese Weise lassen sich die Informationen leichter ablesen. In der Abbildung 4.6, „Tabellensortierung“ wurden die Akzeptanztests nach ihrem Status sortiert.



	Method	Status ↓
serStoryTest	testFailingAcceptanceTest	Failed
dUserStoryTest	testSuccessfulAcceptanceTest	Failed
dUserStoryWithFailureTest	testFailingAcceptanceTest	Failed
est	testFailureAcceptanceTest	Failed
		Not yet implemented
		Not yet implemented
		Not yet implemented
ryTest	testSuccessfulAcceptanceTest	Successful
serStoryTest	testSuccessfulAcceptanceTest	Successful
dUserStoryWithFailureTest	testSuccessfulAcceptanceTest	Successful

Abbildung 4.6. Tabellensortierung

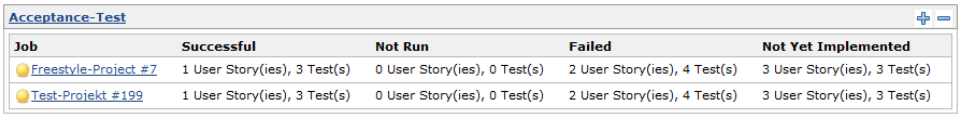
[2] Auf Gesamtprojektebene navigiert der Link lediglich zur Übersichtsseite des Surefire-Reports.

4.2.3. Dashboard View

Der Hudson bietet für seine Hauptseite die Möglichkeit, individuelle Dashboard Views zu definieren. Eine View bietet eine besondere Ansicht auf die ausgewählten Jobs. Mit den Reitern, die für jede View erzeugt werden, kann zwischen den Ansichten gewechselt werden. Eine Dashboard View besteht aus einer Menge von Portlets, die sich beliebig hinzufügen und frei kombinieren lassen. Jedes Portlet bekommt eine Position in der Hauptseite zugewiesen und zeigt dort spezifische Informationen zu ausgewählten Jobs an.

Ist das Portlet aus dem checkerberry business view in eine Dashboard View eingebunden, wird zu den ausgewählten Jobs eine Übersicht über den aktuellen Akzeptanzteststatus angezeigt. Allerdings werden nur Jobs mit Projekten berücksichtigt, für die "Publish combined Acceptance-Test Report" aktiviert wurde (siehe Abschnitt 4.3.2.2, „Ansichten aktivieren").

Jede Zeile enthält einen Hudson-Job mit den Ergebnissen des letzten erfolgreichen Builds. Zu jedem Status ist die entsprechende Anzahl an User Stories und Akzeptanztests notiert. Über die Verlinkung an dem Namen des Jobs gelangt man zur Ansicht des letzten erfolgreichen Builds.


Acceptance-Test

Job	Successful	Not Run	Failed	Not Yet Implemented
Freestyle-Project #7	1 User Story(ies), 3 Test(s)	0 User Story(ies), 0 Test(s)	2 User Story(ies), 4 Test(s)	3 User Story(ies), 3 Test(s)
Test-Projekt #199	1 User Story(ies), 3 Test(s)	0 User Story(ies), 0 Test(s)	2 User Story(ies), 4 Test(s)	3 User Story(ies), 3 Test(s)

Abbildung 4.7. Darstellung des Acceptance-Test Portlet

4.2.4. HTML-ATDD-Report

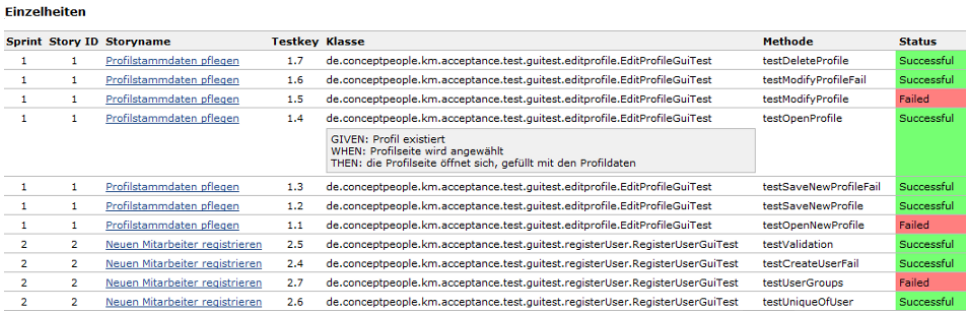
Der HTML-Report bietet die Möglichkeit die Ergebnisse der Akzeptanztests außerhalb des Hudson darzustellen.

Der HTML-Report stellt die Ergebnisse eines Builds auf zwei verschiedene Arten dar. Als erstes wird eine Grafik gezeigt, die mittels eines Tortendiagrammes aufzeigt, wieviele User Stories erfolgreich waren, fehlgeschlagen sind, nicht gelaufen sind oder noch nicht implementiert wurden. Das Diagramm selbst stellt das Verhältnis dar, während die Anzahl und der prozentuale Anteil unter der Grafik aufgeführt werden.

Abbildung 4.8. Tortendiagramm HTML-ATDD-Report

Der zweite Bereich des HTML-Reports ist die tabellarische Auflistung der einzelnen Akzeptanztests. Es wird angezeigt, zu welchem Sprint und zu welcher User Story ein Akzeptanztest gehört. Des Weiteren wird die Id

des Akzeptanztest und der zugehörige Methoden- und Klassennamen aufgeführt. Der User Story-Name hat eine Verlinkung unter der man sich die Beschreibung der User Story ansehen kann, wenn diese angegeben wurde.



Einzelheiten

Sprint	Story ID	Storyname	Testkey	Klasse	Methode	Status
1	1	Profilstammdaten pflegen	1.7	de.conceptpeople.km.acceptance.test.guitest.editprofile.EditProfileGuiTest	testDeleteProfile	Successful
1	1	Profilstammdaten pflegen	1.6	de.conceptpeople.km.acceptance.test.guitest.editprofile.EditProfileGuiTest	testModifyProfileFail	Successful
1	1	Profilstammdaten pflegen	1.5	de.conceptpeople.km.acceptance.test.guitest.editprofile.EditProfileGuiTest	testModifyProfile	Failed
1	1	Profilstammdaten pflegen	1.4	de.conceptpeople.km.acceptance.test.guitest.editprofile.EditProfileGuiTest GIVEN: Profil existiert WHEN: Profilseite wird angewählt THEN: die Profilseite öffnet sich, gefüllt mit den Profildaten	testOpenProfile	Successful
1	1	Profilstammdaten pflegen	1.3	de.conceptpeople.km.acceptance.test.guitest.editprofile.EditProfileGuiTest	testSaveNewProfileFail	Successful
1	1	Profilstammdaten pflegen	1.2	de.conceptpeople.km.acceptance.test.guitest.editprofile.EditProfileGuiTest	testSaveNewProfile	Successful
1	1	Profilstammdaten pflegen	1.1	de.conceptpeople.km.acceptance.test.guitest.editprofile.EditProfileGuiTest	testOpenNewProfile	Failed
2	2	Neuen Mitarbeiter registrieren	2.5	de.conceptpeople.km.acceptance.test.guitest.registerUser.RegisterUserGuiTest	testValidation	Successful
2	2	Neuen Mitarbeiter registrieren	2.4	de.conceptpeople.km.acceptance.test.guitest.registerUser.RegisterUserGuiTest	testCreateUserFail	Successful
2	2	Neuen Mitarbeiter registrieren	2.7	de.conceptpeople.km.acceptance.test.guitest.registerUser.RegisterUserGuiTest	testUserGroups	Failed
2	2	Neuen Mitarbeiter registrieren	2.6	de.conceptpeople.km.acceptance.test.guitest.registerUser.RegisterUserGuiTest	testUniqueOfUser	Successful

Abbildung 4.9. Übersicht HTML-ATDD-Report

4.3. Konfiguration

Im nachfolgenden werden die notwendigen Konfigurationsschritte erläutert, um die Auswertung der User Stories und die Darstellung der Ergebnisse im Hudson oder als eigenständigen HTML-Report darzustellen.

4.3.1. Maven-Plugin

Das `checkerberry-atdd-maven-plugin` wird benötigt, um die User Stories einzulesen und die darin enthaltenen Akzeptanzkriterien (definiert durch eine Menge von Akzeptanztests) mit den Surefire-Ergebnissen zu vergleichen. Die Resultate der Auswertung schreibt das Maven-Plugin in die XML-Datei `acceptancetest.xml`. Auf Basis dieser Datei werden die aufbereiteten Ergebnisse durch checkerberry business view in den Build integriert und dargestellt.

4.3.1.1. Plugin Beschreibung

Um eine Auswertung der User Stories vorzunehmen, ist es erforderlich, das `checkerberry-atdd-maven-plugin` mit dem Maven-Goal **analyze-tests** aufzurufen. Der vollständige Name des Maven-Goals lautet `de.conceptpeople.checkerberry:checkerberry-atdd-maven-plugin:3.2.x:analyze-tests`. Das Plugin verwendet einen separaten Parser, um die Informationen einer User Story auszulesen. Der mitgelieferte Standard-Parser liest die Informationen aus einer Textdatei im Dateisystem (siehe Abschnitt 4.4.2, „Verwendung des Standard-Parsers"). Im Abschnitt 4.4.1, „Verwendung des Jira-UserStory-Retrievers" wird beschrieben, wie die User Story-Informationen aus Atlassian Jira ausgelesen werden können. Darüber hinaus ist es auch möglich, eigene Parser zu definieren, die unterschiedliche Formate unterstützen oder die die User Story-Informationen aus anderen Quellen wie z.B. einer Datenbank lesen. Nähere Information dazu finden sich in Abschnitt 4.4.3, „Eigene User Story-Parser schreiben".

In der Konfiguration des Plugins ist die Angabe von mindestens einem Parser Pflicht. Die Angabe der Parser erfolgt über das `<parsers>`-Tag, welches wiederum ein oder mehrere `<parser>`-Tags enthält (siehe Abschnitt 4.3.1.2, „Beispielkonfiguration").

Damit die Status der Akzeptanztests zu den eingelesenen User Stories auswertbar sind, müssen die Surefire-Ergebnisse richtig eingelesen werden. Sind die Surefire-Reports wie üblich unter `target/surefire-reports` abgelegt, ist keine zusätzliche Konfiguration notwendig. Befinden sich die Surefire-Ergebnisse in einem abweichenden Ordner oder sind über mehrere Ordner verteilt, kann über den optionalen Parameter `<reportDirectories>` abweichende Pfade gesetzt werden. Um sicherzustellen, dass die Surefire-Ergebnisse beim Auswerten der User Stories bereits vorliegen, wird das Goal **analyze-tests** des Plugins standardmäßig in der Maven-Lebenszyklusphase `post-integration-test` ausgeführt.

Das Encoding der Ausgabedatei `acceptancetest.xml` kann über den Parameter `<encoding>` gesetzt werden. Darüber hinaus kann über den Parameter `<outputDirectory>` bestimmt werden, wohin die Ausgabedatei geschrieben wird. In den beiden nachfolgenden Tabellen sind die notwendigen und die optionalen Parameter zusammengefasst.

Benötigte Parameter

Name	Type	Description
parsers	Parser[]	Ein oder mehrere Parser, die dazu verwendet werden, die User Stories einzulesen.

Optionale Parameter

Name	Type	Description
encoding	String	Setzt das Encoding, welches beim Schreiben der Ausgabedatei verwendet wird. **Default value is:** UTF-8
outputDirectory	File	Pfad zu dem Ausgabeordner für die `acceptancetest.xml`. **Default value is:** target
reportDirectories	File[]	Pfade zu den `reports` Ordnern. **Default value is:** target/surefire-reports

4.3.1.2. Beispielkonfiguration

Nachfolgend sind einige Konfigurationsbeispiele aufgeführt.

Neben den üblichen Angaben wie `groupId`, `artifactId`, `version` und `executions`, existieren mit `dependencies` und `configuration` zwei wichtige Konfigurationsabschnitte in dem `checkerberry-atdd-maven-plugin`. Wie eingangs erwähnt, lassen sich dem Plugin unterschiedliche Parser hinzufügen, die das Interface `UserStoryParser` implementieren. Um dem Plugin einen Parser bekannt zu machen, muss dieser als `dependency` eingetragen werden. Erst dann ist es möglich, den Parser über die `configuration` zu verwenden. In dem unteren Beispiel wird das `checkerberry-atdd-userstory-fileparser.jar` in der Version 3.2.x eingebunden. Dieses enthält die Parser-Klasse `de.conceptpeople.checkerberry.atdd.parser.UserStoryFileParser`, welche in der Konfiguration als Parser ausgewählt wird. Über `properties` können Einstellungen an den Parser übergeben werden, die dann beim Parsen zur Verfügung stehen.

```
<plugins>
  <plugin>
    <groupId>de.conceptpeople.checkerberry</groupId>
    <artifactId>checkerberry-atdd-maven-plugin</artifactId>
    <version>3.2.x</version>
    <executions>
     <execution>
      <id>analyze-tests</id>
      <goals>
       <goal>analyze-tests</goal>
      </goals>
     </execution>
    </executions>
    <dependencies>
     <dependency>
      <groupId>de.conceptpeople.checkerberry</groupId>
      <artifactId>checkerberry-atdd-userstory-fileparser</artifactId>
      <version>3.2.x</version>
     </dependency>
    </dependencies>
    <configuration>
     <parsers>
      <parser>
       <parserClass>
         de.conceptpeople.checkerberry.atdd.parser.UserStoryFileParser
       </parserClass>
       <properties>
         <userStoryDirectory>src/test/resources/stories</userStoryDirectory>
       </properties>
      </parser>
     </parsers>
    </configuration>
  </plugin>
</plugins>
```

Beispiel 4.1. Beispielkonfiguration des Maven-Plugins

Die Angaben zu einem Parser bestehen aus zwei Teilen. Mit der `parserClass` muss auf eine gültige Klasse verwiesen werden, die das Interface `UserStoryParser` implementiert. Die Angabe der `properties` ist optional. Ob und welche zusätzlichen Informationen ein Parser benötigt, um die User Stories erfolgreich einzulesen, hängt vom jeweiligen Parser ab. Daher ist insbesondere bei der Implementierung eigener Parser darauf zu achten, dass notwendigen Properties für den Benutzer transparent sind.

Sollen unterschiedliche Typen von User Stories eingelesen werden, lassen sich innerhalb von `configuration` mehrere Parser angeben. Die Parser werden während der Ausführung des Plugins in der angegebenen Reihenfolge aufgerufen. Es ist darauf zu achten, dass jeder verwendete Parser in den `dependencies` des Plugins bekannt gemacht wurde. Das Beispiel zeigt schematisch das Einbinden von mehreren Parsern.

```
...
  <parsers>
    <parser>
      <parserClass>...</parserClass>
      <properties>...</properties>
    </parser>
    <parser>
      <parserClass>...</parserClass>
      <properties>...</properties>
    </parser>

      ...

  </parsers>
...
```

Beispiel 4.2. Einbinden mehrerer User Story Parser

4.3.2. Hudson Plugin

Die wesentliche Aufgabe von checkerberry business view besteht darin, den Projektfortschritt anhand von User Stories und deren Akzeptanztests darzustellen. Um dies zu ermöglichen, ist eine Installation des Plugins in den Hudson und die Konfiguration der unterschiedlichen Ansichten notwendig.

4.3.2.1. Hudson-Integration

Checkerberry business view wird über die "Manage Plugins"- Konfiguration (Plugins verwalten) in den Hudson integriert. In den erweiterten Einstellungen kann die Datei `checkerberry-atdd-hudson-plugin.hpi` zum Upload ausgewählt werden. Um das Plugin verwenden zu können, ist ein anschließender Neustart des Hudsons erforderlich. Nach erfolgreichem Neustart verfügt der Hudson über die zusätzlichen Ansichten. Um die Dashboard-Sicht von checkerberry business view nutzten zu können, ist zudem das allgemeine "Dashboard View"-Plugin (`dashboard-view.hpi`) für den Hudson erforderlich.

4.3.2.2. Ansichten aktivieren

Die Konfiguration von checkerberry business view gestaltet sich im Allgemeinen recht einfach. Globale Einstellungen müssen nicht vorgenommen werden. Ob das Plugin ausgeführt werden soll, kann individuell für jeden Hudson-Job, in dessen Konfiguration, bestimmt werden. Derzeit werden alle gängigen Projekttypen unterstützt. Dabei unterscheiden sich die Projekttypen hinsichtlich ihrer Konfiguration nicht. Lediglich für Maven2/3-Projekte steht eine zusätzliche Option zur Verfügung, mit der die Auswertung der Akzeptanztests auch auf Ebene der Module angezeigt werden kann.

Um die Auswertung für ein gesamtes Projekt anzuzeigen, muss unabhängig vom Projekttyp in der Projektkonfiguration unter "Post-build Actions" der Haken bei "Publish combined Acceptance-Test Report" gesetzt werden. Zudem ist die Angabe eines Patterns erforderlich, um die `acceptancetest.xml` Dateien ausfindig zu machen. Besteht beispielsweise ein Projekt aus mehreren Modulen, können entsprechend mehrere relevante `acceptancetest.xml` Dateien existieren. Ohne die Angabe eines Patterns, wird der Standardwert `**/target/acceptancetest.xml` verwendet.

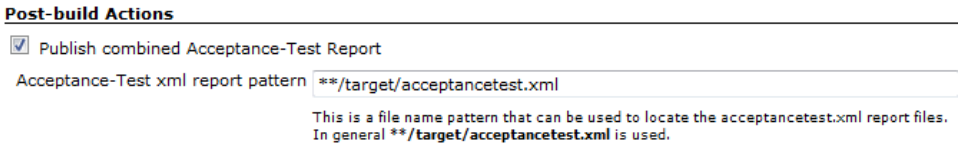


Abbildung 4.10. Publish combined Acceptance-Test Report

Für ein Maven2/3-Projekt besteht zusätzlich die Option, eine Auswertung getrennt für die einzelnen Module anzuzeigen. Dafür ist unter "Build Settings" des Moduls lediglich der Haken bei "Publish Acceptance-Test Report" zu setzen.

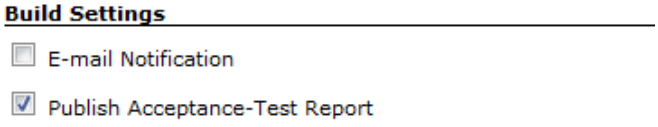


Abbildung 4.11. Publish Acceptance-Test Report für Module

Es ergibt sich somit für Maven2/3-Projekte die Möglichkeit zu wählen, ob die Auswertung nur für das Gesamtprojekt, nur für die Module oder für beides vorgenommen werden soll. Für alle anderen Projekttypen ist nur eine Auswertung bezogen auf das Gesamtprojekt möglich.

Hinweis: Vorgreifend sei erwähnt, dass die Darstellung der Akzeptanztestauswertung, Verlinkungen auf die Surefire-Testergebnisse enthält. Damit diese funktionieren können, muss insbesondere für Freestyle-Projekte

in der Konfiguration unter "Post-build Actions" der Haken bei "Publish JUnit test result report" gesetzt sein und ein Pattern eingetragen werden, das auf die XML-Dateien des Surefire-Reports verweist.

4.3.2.3. Dashboard View einrichten

Checkerberry business view verfügt über ein Portlet, das sich ohne besondere Konfiguration in eine Dashboard View einbinden lässt. Soll das Portlet in eine neue View eingebunden werden, kann diese auf der Hauptseite mit einem Klick auf den "+" Reiter erzeugt werden. In dem dann folgenden Wizard ist es erforderlich, einen Name für die View anzugeben und "Dashboard" auszuwählen.

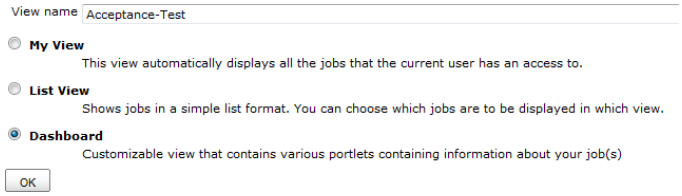


Abbildung 4.12. Dashboard View erstellen

Mit "Add Dashboard Portlet to ..." kann das Portlet an jeder beliebigen Stelle in der View eingebunden werden. Über den Jobfilter lässt sich bestimmen, welche Jobs in der Ansicht dargestellt werden. Damit ein Hudson-Job in dem Portlet des checkerberry business views angezeigt werden kann, muss er zudem über eine Auswertung der Akzeptanztests auf Gesamtprojektebene verfügen (Publish combined Acceptance-Test Report muss ausgewählt sein).

4.3.3. HTML-Maven-Plugin

Das `checkerberry-atdd-html-maven-plugin` ermöglicht es aus den XML-Dateien, die bei der Auswertung der User Stories entstehen, einen HTML-Report zu generieren. Dazu wird die Datei `acceptancetest.xml` benötigt, die von dem `checkerberry-atdd-maven-plugin` erzeugt wird (siehe Abschnitt 4.3.1, „Maven-Plugin").

4.3.3.1. Plugin Beschreibung

Um einen HTML-Report aus der Datei `acceptancetest.xml` zu generieren, ist es notwendig, das `checkerberry-atdd-html-maven-plugin` mit dem Maven-Goal `html-acceptance-tests` aufzurufen. Der vollständige Name des Maven-Goals lautet `de.conceptpeople.checkerberry:checkerberry-atdd-maven-plugin:3.1.x:html-acceptance-tests`. Das Plugin erzeugt aus der Datei `acceptancetest.xml` einen HTML-Report. Die Akzeptanztests werden zum einen tabellarisch angezeigt und zum anderen auch als Tortengrafik dargestellt. Diese Grafik wird ebenfalls von dem Plugin erzeugt und im gleichen Ordner wie die HTML-Datei des Reports abgespeichert.

Der informative Inhalt der HTML-Datei wird aus der Datei `acceptancetest.xml` generiert. Diese Datei wird von dem `checkerberry-atdd-maven-plugin` erzeugt (siehe dazu Abschnitt 4.3.1, „Maven-Plugin"). Standardmäßig wird die Datei in `src/resources/acceptancetest.xml` erwartet. Über den Parameter `<pathXmlTestFile>` lässt sich ein alternativer Pfad setzen.

Mit der Datei `acceptanceTestHtmlTemplate.vm` wird die Struktur des HTML-Reports bestimmt. Es ist möglich ein eigenes Template anzugeben und so das Aussehen des HTML-Reports zu verändern. Standardmäßig liegt die Datei `acceptanceTestHtmlTemplate.vm` in `src/main/rescources` und kann über den Parameter `<pathTemplateFile>` angepasst werden.

Ebenfalls ist es mögliche andere CSS-Dateien anzugeben um den Style des HTML-Reports anzupassen. Die Farben für die Statusanzeige werden in der Datei `styleColor.vm`, angepasst, während die Datei

`style.vm`, den eigentlichen Style des HTML-Reports festgelegt. Standardmäßig findet man diese Dateien in `src/main/resources`. Wenn eine andere Datei oder ein anderes Verzeichnis verwenden werden soll, kann das über den Parameter `<pathColorCssFile>` bzw. den Parameter `<pathCssFile>` gesetzt werden.

Über den Parameter `<siteName>` ist es möglich den Namen des HTML-Reports zu setzen. Standardmäßig wird der Name „Akzeptanztests“ verwendet.

Das Encoding der Ausgabedatei `Akzeptantztest.html` kann über den Parameter `<encoding>` gesetzt werden. Darüber hinaus kann über den Parameter `<outputDirectory>` bestimmt werden, wohin die Ausgabedatei geschrieben wird. In der nachfolgenden Tabelle sind die verschiedenen Parameter zusammengefasst.

Name	Type	Description
encoding	String	Setzt das Encoding, welches beim Schreiben der Ausgabedatei verwendet wird. **Default value is:** UTF-8
siteName	String	Setzt den Dateinamen des HTML-Reports. **Default value is:** Akzeptanztests
pathColorCssFile	String	Setzt den Pfad für die CSS-Datei, die die Farbeinstellungen für den Status regelt. **Default value is:** `src/main/resources/templates/styleColor.vm`
pathCssFile	String	Setzt den Pfad für die CSS-Datei, die für den Style des HTML-Reports verantwortlich ist. **Default value is:** `src/main/resources/templates/style.vm`
outputDirectory	String	Setzt den Pfad zu dem Zielverzeichnis des HTML-Reports (inklusive der Grafik). **Default value is:** `src/main/resources`
pathXmlTestFile	String	Setzt den Pfad auf die XML-Datei mit den Testfällen. **Default value is:** `src/main/resources/acceptancetest.xml`
pathTemplateFile	String	Setzt den Pfad für die Template-Datei aus der der HTML-Report generiert wird. **Default value is:** `src/main/resources/templates/acceptanceTestHtmlTemplate.vm`

4.3.3.2. Beispielkonfiguration

Nachfolgend wird ein Konfigurationsbeispiel für das `checkerberry-atdd-html-maven-plugin` gezeigt. In diesem Beispiel werden in den meisten Fällen die Standardwerte verwendet, es wird nur ein anderer Pfad für die Datei `acceptancetest.xml` angegeben.

```
<plugins>
  <plugin>
    <groupId>de.conceptpeople.checkerberry</groupId>
    <artifactId>checkerberry-atdd-html-maven-plugin</artifactId>
    <version>3.2.x</version>
    <executions>
      <execution>
        <id>html-acceptance-tests</id>
        <goals>
          <goal>html-acceptance-tests</goal>
        </goals>
      </execution>
    </executions>
    <configuration>
      <pathXmlTestFile>”tests/acceptancetest.xml”</pathXmlTestFile>
    </configuration>
  </plugin>
</plugins>
```

Beispiel 4.3. Beispielkonfiguration des HTML-Maven-Plugins

4.4. User Stories schreiben

Vor dem Schreiben der User Stories steht die Entscheidung, welcher User Story Parser eingesetzt werden soll. Je nach gewählten Parser kann sich das Format, in dem die User Stories verfasst werden, stark unterscheiden. Das `checkerberry-atdd-maven-plugin` bietet sowohl die Möglichkeit auf den bestehenden Standard-Parser zurückzugreifen als auch die Verwendung einer eigenen Parser-Implementierung. Wird eine eigene Implementierung gewählt, kann der Benutzer beispielsweise selbst bestimmen, in welchem Format die User Stories gespeichert werden.

Im Folgenden wird zunächst die Verwendung des Jira- und Standard-Parsers erläutert und anschließend auf die Implementierung eigener Parser genauer eingegangen.

4.4.1. Verwendung des Jira-UserStory-Retrievers

Atlassian Jira ist ein Projektverfolgungstool für Teams. In diesem können Einträge (sogenannte Vorgänge) hinterlegt werden, die eine Aufgabe für den Entwickler beschreiben. Die Vorgänge bestehen unter anderem aus einem Vorgangstyp zur besseren Klassifizierung, einer Priorität für die Entwicklung, einer Beschreibung was entwickelt werden soll und benutzerbezogene Daten wie z.B. wer hat den Vorgang erstellt und wer soll ihn bearbeiten (siehe Abbildung 4.13, „Ein Vorgang in Jira“).

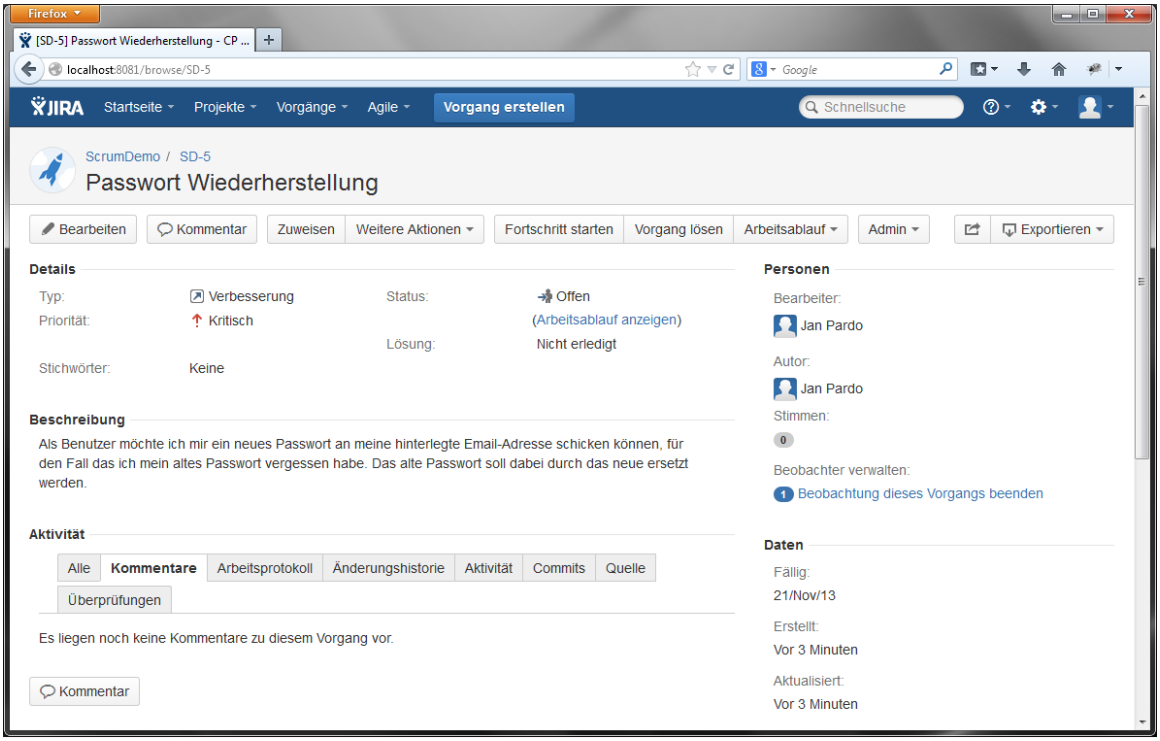


Abbildung 4.13. Ein Vorgang in Jira

Der Jira Userstory Retriever kann die User Story Informationen, die für checkerberry business view benötigt werden, direkt aus den Jira Vorgängen auslesen.

Um den Jira Userstory Retriever nutzen zu können, muss Jira installiert sein und der Jira-REST-Service muss aktiv sein. Desweiteren benötigt man einen Benutzer in Jira, welcher Zugriff zu den benötigten Vorgängen besitzt. Es wird empfohlen, dass dieser Benutzer nur lesenden Zugriff auf Jira hat. Da der Jira-UserStory-Retriever seine Konfiguration während des build-Vorgangs von Maven bekommt, ist die Verwendung von Maven zwingend erforderlich.

Die nächsten Abschnitte beschreiben wie der Jira-UserStory-Retriever konfiguriert und eingesetzt wird.

4.4.1.1. Konfiguration des Maven-Projekts

Der Jira-UserStory-Retriever bezieht seine Konfigurationsdaten während des build-Vorgangs aus der `pom.xml` von Maven. Um ihn in das Projekt einzubinden, muss er über ein neues Plugin in der `pom.xml` konfiguriert werden (siehe Beispiel Maven-Konfiguration). Dort ist zu sehen wie das neue Plugin eingefügt wird. Es handelt sich dabei um ein `checkerberry-atdd-maven-plugin` welches in der Phase `analyze-tests` ausgeführt wird. Als Abhängigkeit wird der `checkerberry-atdd-jira-userstory-retriever` angegeben. In dem Konfigurationsblock wird der JiraUserStoryRetriever als Parserklasse definiert. Konfiguriert wird dieser dann in dem properties-Block. Abschnitt 4.4.1.2, „Konfiguration des Jira-UserStory-Retriever“ geht näher auf die Konfiguration des Jira-UserStory-Retrievers ein. Für nähere Informationen zur `pom.xml`-Konfiguration, siehe auch Abschnitt 4.3, „Konfiguration“.

```
<plugin>
  <groupId>de.conceptpeople.checkerberry</groupId>
  <artifactId>checkerberry-atdd-maven-plugin</artifactId>
  <version>3.2.x</version>
  <executions>
    <execution>
      <id>analyze-tests</id>
      <goals>
        <goal>analyze-tests</goal>
      </goals>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>de.conceptpeople.checkerberry</groupId>
      <artifactId>checkerberry-atdd-jira-userstory-retriever </artifactId>
      <version>3.2.x</version>
    </dependency>
  </dependencies>
  <configuration>
    <parsers>
      <parser>
        <parserClass>
          de.conceptpeople.checkerberry.atdd.jira.JiraUserStoryRetriever
        </parserClass>
        <properties>
          […]
        </properties>
      </parser>
    </parsers>
  </configuration>
</plugin>
```

Beispiel 4.4. Beispiel

4.4.1.2. Konfiguration des Jira-UserStory-Retriever

Die Konfiguration des Jira-UserStory-Retriever teilt sich in zwingend erforderliche Elemente und optionale Elemente. Die zwingend erforderlichen Elemente werden für eine erfolgreiche Verbindung zu Jira benötigt, die optionalen Elemente erlauben die Anpassung des Verhaltens. Fehlen die optionalen Werte, wird der Jira-UserStory-Retriever auf ein Standardverhalten zurückgreifen. Im Detail lassen sich folgende Elemente konfigurieren:

Name	Erforderlichkeit	Beschreibung
url	Zwingend	Die Url des laufenden Jira-Servers, z.B. http://localhost:8080.
Name	Zwingend	Ein Jira-Benutzername, der zumindest lesende Rechte besitzt, z.B. user1
Password	Zwingend	Das Passwort des Jira-Benutzers, z.B. password1
projectNames	Optional	Eine Liste von Jira-Projektnamen, die beim Auslesen der Vorgänge berücksichtigt werden sollen. Mehrere Projektnamen werden mit einem Komma voneinander getrennt z.B. project1,project2,project3. Fehlt dieses Feld, werden alle Projekte berücksichtigt.
backlogSprint	Optional	Dieses Feld wird nur bei Jira-Agile benutzt. Wird die Sprint-Information aus Jira-Agile ausgelesen und der Vorgang besitzt keinen zugeordneten Sprint, so wird der Vorgang dem virtuellen Sprint *Backlog*

Name	Erforderlichkeit	Beschreibung
		zugeordnet. Der Name dieses virtuellen Sprints kann hier angepasst werden, z.B. NichtZugewiesen.
unknownSprint	Optional	Liegt kein Jira-Agile vor und es wurden keine Sprint-Informationen im Beschreibungstext hinterlegt, so wird der Vorgang dem virtuellen Sprint *Unbekannt* zugeordnet. Der Name dieses virtuellen Sprints kann hier angepasst werden, z.B. NichtZugewiesen.
issueType	Optional	Legt fest welche Vorgangstypen als User Stories interpretiert werden. Standardmäßig werden alle Vorgänge die den Jira-Vorgangstyp Story besitzen berücksichtigt. Das Verhalten kann an dieser Stelle angepasst werden, indem man z.B. den Typ Feature konfiguriert.

Die entsprechenden Konfigurationseinträge werden in der `pom.xml` des Projektes als properties-Unterpunkte des plugins vorgenommen (siehe Abschnitt 4.4.1.1, „Konfiguration des Maven-Projekts“).

Die Konfiguration Jira-UserStory-Retriever zeigt eine Beispielkonfiguration in dem alle Elemente gesetzt werden. Neben den Zugangsdaten wird das Verhalten so angepasst, dass nur die beiden Projekte *ScrumDemo1* und *ScrumDemo2* ausgelesen werden. Vorgänge, die in Jira Agile Sprint-Informationen besitzen, aber zu keinem Sprint zugeordnet wurden, werden automatisch dem Sprint *Noch zu erledigen* zugeteilt. Vorgänge welche keine Sprint-Informationen besitzen, werden dem Sprint *Unzugeordnet* zugeteilt. Als User Stories werden nur Vorgänge betrachtet, welche den Vorgangstyp *Feature* besitzen.

```
<plugin>
  […]
  <configuration>
    <parsers>
      <parser>
        <parserClass>
          de.conceptpeople.checkerberry.atdd.jira.JiraUserStoryRetriever
        </parserClass>
        <properties>
          <url>http://localhost:8081/</url>
          <name>testUser</name>
          <password>testPassword</password>
          <projects>ScrumDemo1,ScrumDemo2</projects>
          <backlogSprint>Noch zu erledigen</backlogSprint>
          <unknownSprint>Unzugeordnet</unknownSprint>
          <issueType>Feature</issueType>
        </properties>
      </parser>
    </parsers>
  </configuration>
</plugin>
```

Beispiel 4.5. Konfiguration Jira-UserStory-Retriever

4.4.1.3. Jira User Story Format

Der Jira UserStory Retriever benötigt für eine User Story eine Id, einen Namen, eine Priorität und Informationen über die Akzeptanztests. Beim Einlesen eines Jira-Vorgangs übernimmt der Jira UserStory Retriever die Id, den Namen und die Priorität vom Vorgang. Weiterhin wird die Information benötigt in welchem Sprint sich der Vorgang befindet. Liegt Jira Agile vor, kann diese Information aus dem Sprint-Feld des Vorgangs ausgelesen werden. Andernfalls muss diese Information im Beschreibungstext angegeben werden. Fehlt die Sprint-Information gänzlich, so wird der Vorgang einem virtuellen Sprint (*Unbekannt*) zugeordnet.

Der Jira UserStory Retriever liest standardmäßig alle Vorgänge eines Projekts ein, die den Jira-Typ Story besitzen (zur Anpassung dieses Verhaltens siehe Abschnitt 4.4.1.2, „Konfiguration des Jira-UserStory-Retriever"). Aus diesen wird dann der Beschreibungstext interpretiert. Finden sich in dem Beschreibungstext Informationen zu der Id, dem Namen, dem Sprint oder der Priorität, so haben diese Vorrang vor den Informationen die direkt aus dem Jira-Vorgang erhalten werden. Dies erlaubt es, die im Vorgang befindlichen Informationen zu überschreiben. Im Beschreibungstext können alle Informationen des Jira Vorgangs für checkerberry business view angepasst werden. Insbesondere sind dies:

- Die User Story Id (@UserStory("..."))
- Der User Story Name (@UserStoryName("..."))
- Die Priorität (@Priority("..."))
- Der Sprint (@Sprint("..."))

Auch werden hier dann die für die User Story nötigen Akzeptanzkriterien definiert. Das Format der Akzeptanzkriterien folgt dabei dem des Standardparsers. Für nähere Informationen dazu siehe Abschnitt 4.4.1.1, „Konfiguration des Maven-Projekts".

Ein Beispielhafter Vorgang ist in Abbildung 4.14, „Wichtige Felder eines Jira Vorgangs" zu sehen. Dort sieht man den Jira Vorgang mit der Id SD-2 und dem Namen *Zweite Demo-Story*. Der Vorgangstyp ist *Story* und die Priorität *Kritisch*. Für die User Story von checkerberry business view wird im Beschreibungstext der Name der User Story allerdings durch einen neuen Namen überschrieben (*Ich möchte mich an dem System anmelden können*). Außerdem wird dort der User Story ein Sprint zugeordnet (*Der erste*). Danach folgen die Akzeptanzkriterien. Dem Akzeptanzkriterium 2 (*Hinzufügen eines Benutzers*) ist dabei ein Akzeptanztest (*AT2*) zugeordnet.

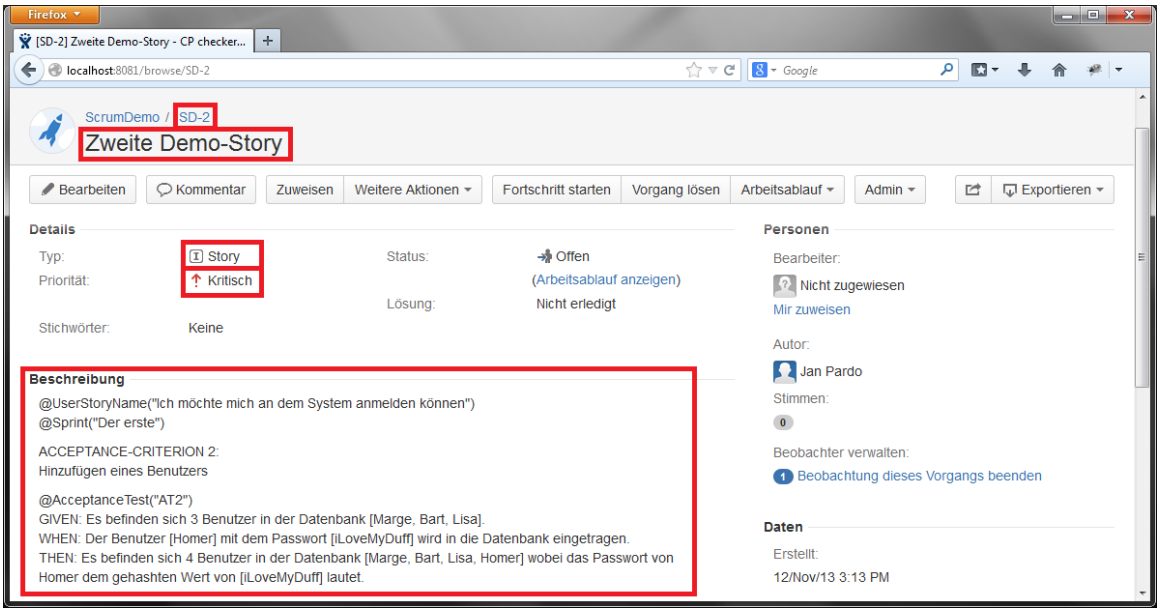


Abbildung 4.14. Wichtige Felder eines Jira Vorgangs

Abbildung 4.15, „checkerberry ATDD Hudson Plugin" zeigt unter anderem das Aussehen des in Abbildung 4.14, „Wichtige Felder eines Jira Vorgangs" zu sehenden Vorgangs im checkerberry business view Plugin für Hudson. Aus dem Vorgang wurde die Story Id übernommen. Der zugeordnete Sprint und der Story Name wurden ebenso wie das Akzeptanzkriterium aus dem Beschreibungstext ausgelesen. In dem Beispiel gibt es in Jira insgesamt fünf Vorgänge, von denen drei in Jira als Story markiert sind und somit vom System erkannt wurden. Zwei davon wurden implementiert (Stories SD-1 und SD-2), der dritte (SD-4) muss noch implementiert werden.

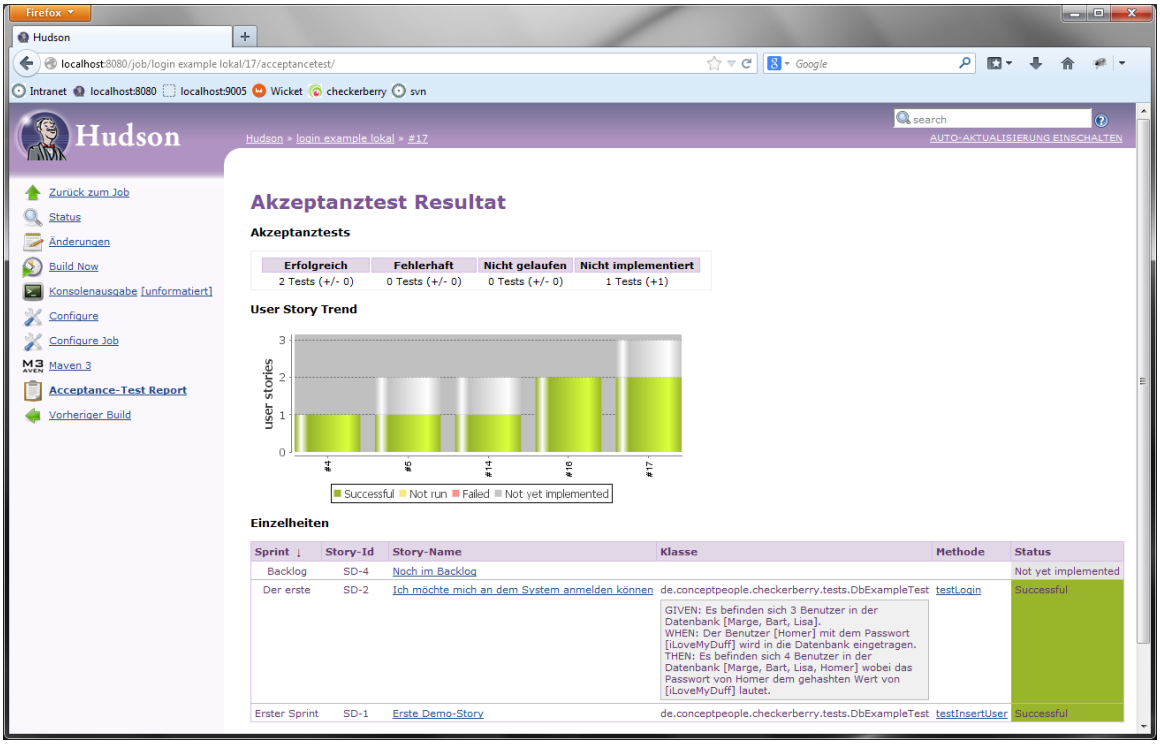


Abbildung 4.15. checkerberry ATDD Hudson Plugin

4.4.2. Verwendung des Standard-Parsers

Zu jedem Parser existiert ein spezielles Format, in dem die User Stories beschrieben werden. Im Falle des Standard-Parsers handelt es sich um eine einfache Textdatei. Wie diese aufgebaut ist, wird im nächsten Abschnitt beschrieben. Anschließend wird auf die Eigenschaften des Parsers näher eingegangen.

4.4.2.1. User Story Format

Das Format der User Stories ist klar definiert. Es handelt sich um Textdateien mit der Dateiendung `.story`. Die Beschreibung der User Story erfolgt nach einer festen Struktur. Nachfolgend ist eine User Story beispielhaft dargestellt.

```
@UserStory("1")
@UserStoryName("Name")
@Sprint("3")
@Priority("2")

DESCRIPTION:
Beschreibung der User Story.

REASON:
Grund für die User Story.

ACCEPTANCE-CRITERION 1:
Beschreibung der Anforderungen, die erfüllt sein müssen.

@AcceptanceTest("1.1")
GIVEN: Voraussetzung.
WHEN:  Durchführung.
THEN:  Überprüfung.

ACCEPTANCE-CRITERION 2:
Beschreibung der Anforderungen, die erfüllt sein müssen.

@AcceptanceTest("1.2")
GIVEN: Voraussetzung.
WHEN:  Durchführung.
THEN:  Überprüfung.
```

Beispiel 4.6. Beispiel einer User Story des Standard-Parsers

Jede User Story beginnt mit den vier Pflichtparametern:

@UserStory("...")

@UserStoryName("...")

@Sprint("...")

@Priority("...")

Mit `@UserStory` wird der User Story eine ID gegeben. In der Regel werden die User Stories mit fortlaufenden Nummern versehen. Unterschiedliche `story`-Dateien mit derselben ID werden als eine User Story betrachtet. Als `@UserStoryName` kann eine beliebige Zeichenkette angegeben werden. Der `UserStoryName` dient dem Anwender zur leichteren Unterscheidung der einzelnen User Stories. Mit `@Sprint` findet eine Zuordnung der User Story zu einem Sprint statt, in der die User Story umgesetzt werden soll. Der Entwicklungsprozess einer Software beginnt mit dem ersten Sprint, wobei die nachfolgenden Sprints in aufsteigender Reihenfolge nummeriert werden. Über `@Priority` wird für die User Story eine Priorität festgelegt. Die Priorität wird in der Regel durch den Product Owner vergeben. Innerhalb eines Sprints werden die User Stories gemäß ihrer Priorität umgesetzt.

Anschließend folgen mit `DESCRIPTION` und `REASON` zwei optionale Abschnitte, in denen die User Story beschrieben und der Grund für die User Story angegeben werden kann.

Nachfolgend sind eine Menge von Akzeptanzkriterien (`ACCEPTANCE-CRITERION`) aufgelistet, die alle erfüllt sein müssen, damit die User Story abgeschlossen ist. Jedes Akzeptanzkriterium kann wiederum aus einer beliebigen Anzahl von Akzeptanztests bestehen. Ein Akzeptanztest wird stets durch `@AcceptanceTest("...")` eingeleitet. Innerhalb der runden Klammern muss ein Schlüsselwert angegeben werden. Anhand der Schlüsselwerte findet eine Zuordnung von Akzeptanztests der User Story zu Methoden von Testklassen statt. Dabei werden nur die Methoden einer Klasse berücksichtigt, die mit der Annotation `@AcceptanceTest("...")` versehen sind. Die Annotation `@AcceptanceTest` ist Bestandteil der `checkerberry-atdd-common.jar` und über den vollqualifizierten Klassennamen `de.conceptpeople.checkerberry.atdd.common.annotations.AcceptanceTest` zu erreichen. In Maven-Projekten wird die `checkerberry-atdd-common.jar` wie folgt eingebunden.

```
<dependency>
  <groupId>de.conceptpeople.checkerberry</groupId>
  <artifactId>checkerberry-atdd-common</artifactId>
  <version>3.2.x</version>
</dependency>
```

Beispiel 4.7. @AcceptanceTest-Annotation verfügbar machen

Das folgende Beispiel zeigt eine annotierte Testmethode, die eine Zuordnung zum ersten Akzeptanzkriterium der oben dargestellten User Story herstellt.

```
@AcceptanceTest("1.1")
public void testMeTest(){
  ...
}
```

Beispiel 4.8. Testmethode mit @AcceptanceTest-Annotation

Wie das nachfolgende Beispiel zeigt, können über die Annotation `@AcceptanceTest` mehrere Schlüsselwerte angegeben werden. Dies ist erforderlich, wenn eine Testmethode mehrere Akzeptanzkriterien validiert.

```
@AcceptanceTest({"1.1", "1.2", "2.4"})
public void testVariousTest(){
  ...
}
```

Beispiel 4.9. @AcceptanceTest-Annotation mit mehreren Schlüsselwerten

Ein Akzeptanztest kann durch die Angabe von `GIVEN`, `WHEN` und `THEN` näher beschrieben werden. `GIVEN` beschreibt die Voraussetzungen, welche vor der Durchführung des Tests vorhanden sind. `WHEN` beschreibt, welche Aktionen durchgeführt werden. `THEN` gibt an, welche Zustände nach der Durchführung geprüft bzw. vorhanden sein müssen.

Hinweis: Da das @-Zeichen stets ein Schlüsselwort markiert, darf das Zeichen nur an den vorgesehenen Stellen verwendet werden. Für den Fall, in dem das Zeichen kein Schlüsselwort einleiten soll, ist stattdessen der enkodierte Wert für das @-Zeichen zu verwenden. Der enkodierte Wert ist `&#64;`.

4.4.2.2. Parser Eigenschaften

Mit dem `UserStoryFileParser` aus dem `checkerberry-atdd-userstory-fileparser.jar` wird bereits eine Implementierung eines Parsers bereitgestellt, die in dem `checkerberry-atdd-maven-plugin` verwendet werden kann. Um den Parser verwenden zu können, muss folgende Abhängigkeit im Plugin angegeben werden.

```
<dependency>
  <groupId>de.conceptpeople.checkerberry</groupId>
  <artifactId>checkerberry-atdd-userstory-fileparser</artifactId>
  <version>3.2.x</version>
</dependency>
```

Beispiel 4.10. Standard-Parser bekannt machen

Der vollqualifizierte Klassenname des Parsers, welcher in der Konfiguration des `checkerberry-atdd-maven-plugin` anzugeben ist, lautet `de.conceptpeople.checkerberry.atdd.parser.UserStoryFileParser`. Falls nicht anders über `properties` konfiguriert, erwartet der Parser die User Stories im Verzeichnis `src/test/resources/stories` relativ zur pom-Datei. Der Pfad sowie das verwendete Encoding lassen sich mit `<userStoryDirectory>` bzw. `<encoding>` verändern. Beispielkonfigurationen zum Standard-Parser finden sich in Abschnitt 4.3.1.2, „Beispielkonfiguration“.

Beim Parsen der User Stories werden nicht alle enthaltenen Informationen benötigt. Viele Angaben sind optional und werden vom Parser ignoriert. Auch wenn empfohlen wird, auf die zusätzlichen Informationen nicht zu verzichten, da sie für ein besseres Verständnis beim Benutzer sorgen, könnte eine User Story auch wie folgt aussehen:

```
@UserStory("1")
@UserStoryName("Name")
@Sprint("3")
@Priority("2")

@AcceptanceTest("1.1")
@AcceptanceTest("1.2")
```

Beispiel 4.11. Beispiel einer minimalen User Story des Standard-Parsers

Aus der Sicht des Parser sind lediglich die Angaben beginnend mit dem `@`-Zeichen von Bedeutung. Darüber hinaus gibt es keine Vorgaben, in welcher Reihenfolge die Angaben gemacht werden.

4.4.3. Eigene User Story-Parser schreiben

Wie bereits in den vorherigen Kapiteln erwähnt, lassen sich dem `checkerberry-atdd-maven-plugin` eigene Parser hinzufügen. Voraussetzung dafür ist, dass der Parser das Interface `de.conceptpeople.checkerberry.atdd.common.io.UserStoryParser` aus der Library `checkerberry-atdd-common.jar` implementiert. In dem folgenden Codebeispiel ist das vollständige Interface dargestellt.

```
public interface UserStoryParser {

  /**
   * Liest User Stories ein.
   *
   * @param properties
   *            eine Map mit Parsereinstellungen
   * @return eine Liste von <code>UserStory</code>-Objekten. Werden keine
   *         User Stories gefunden, wird eine leere Liste zurückgegeben.
   * @throws UserStoryParseException
   *             verursacht durch einen Parserfehler
   */
  List<UserStory> parse(Map<String, String> properties)
    throws UserStoryParseException;
}
```

Beispiel 4.12. UserStoryParser-Interface

Das Interface definiert die Methode `parse(Map<String, String> properties)` mit einer Liste von `UserStory`-Objekten als Rückgabewert. Über den Parameter `properties` werden dem Parser alle in der pom definierten Properties übergeben.

Da damit gerechnet werden muss, dass ein Parser falsch oder nur unvollständig konfiguriert sein kann, sollten für die notwendigen Properties Standardwerte gesetzt werden bzw. eine entsprechende Fehlerbehandlung vorhanden sein. Dabei sollten alle Exceptions innerhalb der `parse`-Methode gefangen oder durch die `UserStoryParseException` gekapselt werden.

Beim Parsen der User Stories ist zu vermeiden, dass Stories doppelt eingelesen werden. Dies könnte zu einer falschen Auswertung des Projektfortschritts führen. Da bei der Verwendung des `checkerberry-atdd-maven-plugin` mehrere Parser zum Einsatz kommen können, ist außerdem darauf zu achten, dass die Parser disjunkte Mengen von User Stories einlesen. Im Falle eines File-Parser muss daher der zugehörige File-Filter so angepasst sein, dass nur die betreffenden User Stories eingelesen werden.

Sind für einen angegebenen Parser keine User Stories vorhanden, ist gefordert, dass dieser eine leere Liste von `UserStory`-Objekten zurückliefert.

Bei dem `UserStory`-Objekt handelt es sich um ein einfaches POJO, dessen Konstruktor fünf Parameter erwartet. Die ersten vier Parameter enthalten notwendige Informationen und dürfen daher nicht `NULL` sein.

```
/**
 *
 * Erzeugt eine neue UserStory Instanz.
 *
 * @param id
 *            Identifikator der User Story
 * @param name
 *            Name der User Story
 * @param sprint
 *            Der Sprint in dem die User Story umgesetzt werden soll
 * @param priority
 *            Priorität der User Story
 * @param acceptanceTests
 *            Die Schlüsselwerte der Map entsprechen den Schlüsselwerten der
 *            identifizierten Akzeptanztests. Der Wert in der Map entspricht
 *            der Akzeptanztest-Beschreibung. Da die Beschreibung optional ist,
 *            kann der Wert NULL oder leer sein.
 */
public UserStory(String id, String name, String sprint, String priority,
  Map<String, String> acceptanceTests) {
  ...
}
```

Beispiel 4.13. Konstruktor der UserStory-Klasse

Kapitel 5. Checkerberry cockpit

5.1. Einleitung

Bei dem checkerberry cockpit handelt es sich um ein eigenständiges Programm, das über eine grafische Oberfläche verfügt. Mit dem checkerberry cockpit XML-Konverter können umfangreiche Änderungen an den XML-Testdaten vorgenommen werden, die bei der Verwendung von checkerberry db zum Einsatz kommen.

5.2. Starten des checkerberry cockpits

Das checkerberry cockpit wird über das ausgelieferte JAR-Archiv `checkerberry-cockpit-3.2.x-jar-with-dependencies.jar` gestartet. Dieses ist direkt ausführbar, sodass das checkerberry cockpit bei einem Doppelklick auf das JAR-Archiv gestartet wird.

5.3. checkerberry cockpit XML-Konverter

Das Ziel bei der Entwicklung von automatisierten Tests besteht in der Erreichung einer möglichst hohen Testabdeckung. Im Fall von checkerberry db gilt dabei: Je höher die Testabdeckung, desto mehr XML-Testdaten werden verwendet. Die hohe Anzahl der Testdaten wird dann problematisch, wenn massive Änderungen an der Datenbankstruktur vorgenommen werden wie z.B. die Umbenennung von Tabellen- und Spaltennamen in der Datenbank. In diesem Fall müssen die entsprechenden Tags und Attribute in den XML-Testdaten ebenfalls angepasst werden.

In der Regel sinkt die Gefahr von umfangreichen Datenbankänderungen mit der Laufzeit des Projektes. Inkonsistenzen in dem Datenbankdesign werden nach und nach erkannt und behoben. In einigen Unternehmen wird das Datenbankdesign jedoch von eigenen Abteilungen geprüft und korrigiert. Dies geschieht nicht immer zeitnah. Im schlimmsten Fall beziehen sich die Änderungen auf weite Teile der Datenbank, sodass ggf. eine große Anzahl von Tabellen- und Spaltennamen umbenannt werden müssen. Somit müssen alle XML-Testdaten, also ggf. mehrere hundert Dateien, aktualisiert werden. Diese Aufgabe ist manuell kaum zu lösen.

Aus diesem Grund stellt das checkerberry test center den XML-Konverter als Teil von checkerberry cockpit zur Verfügung, das die Massenänderung von XML-Testdaten ermöglicht. Über eine einfache Excel-Datei können verschiedene Operationen auf den XML-Testdaten ausgeführt werden. Dazu gehören das Umbenennen von Tabellen- und Spaltennamen, das Hinzufügen von neuen Spalten und das Löschen bestehender Spalten. Darüber hinaus können die gewünschten Migrationsschritte auch manuell im checkerberry cockpit XML-Konverter vorgenommen werden.

Im Folgenden wird die Bedienung des XML-Konverters beschrieben.

5.3.1. Aufgaben

Die Aufgabe des checkerberry cockpit XML-Konverters besteht in der Manipulation von XML-Dateien für Testdaten.

Im Folgenden werden die verschiedenen Änderungsmöglichkeiten anhand der folgenden Beispieldateien beschrieben.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC "-//ConceptPeople//DTD sample-db 1.0//EN"
"../sample-db.dtd">

<dataset>
  <USERS SURNAME="Simpson" NAME="Homer" BIRTHDATE="1946-09-16"/>
  <USERS SURNAME="Simpson" NAME="Maggie" BIRTHDATE="2006-09-15"/>
  <SHOP_USERS SURNAME="Smithers" NAME="Waylon" RESERVED=""/>
  <SHOP_USERS SURNAME="Burns" NAME="Montgomery" RESERVED=""/>
</dataset>
```

Beispiel 5.1. Testdaten-Beispiel für checkerberry cockpit XML-Konverter

Die Testdaten enthalten zwei Tabellen (`USERS` und `SHOP_USERS`) mit jeweils zwei Einträgen, die in den folgenden Abschnitten manipuliert werden.

5.3.1.1. Tags umbenennen

Die Tags in den XML-Testdaten entsprechen den Tabellennamen innerhalb der Datenbank. Ändert sich ein Tabellenname, muss somit das entsprechende Tag in den Testdaten umbenannt werden.

Nehmen wir für das obige Beispiel an, dass die Tabelle `USERS` in `BUDDIES` umbenannt wurde. Durch die Übergabe des alten und des neuen Tabellen- bzw. Tag-Namens konvertiert der XML-Konverter die obige Testdatendatei wie folgt:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC "-//ConceptPeople//DTD sample-db 1.0//EN"
"../sample-db.dtd">

<dataset>
  <BUDDIES SURNAME="Simpson" NAME="Homer" BIRTHDATE="1946-09-16"/>
  <BUDDIES SURNAME="Simpson" NAME="Maggie" BIRTHDATE="2006-09-15"/>

  <SHOP_USERS SURNAME="Smithers" NAME="Waylon" RESERVED=""/>
  <SHOP_USERS SURNAME="Burns" NAME="Montgomery" RESERVED=""/>
</dataset>
```

Beispiel 5.2. Testdaten-Beispiel nach Umbenennung der Tags

Der Tabellenname `USERS` wurde in den Testdaten durch den Namen `BUDDIES` ersetzt. Die Tabelle `SHOP_USERS` bleibt hingegen unverändert, obwohl der Name dieser Tabelle auch die Zeichenkette `USERS` beinhaltet.

5.3.1.2. Attribute umbenennen

Die Attribute der Tags in den Testdaten entsprechen den Spaltennamen in der Datenbank. Ändert sich ein Spaltenname in der Datenbank, muss das entsprechende Attribut geändert werden. In unserem Beispiel gehen wir davon aus, dass die Spalte `NAME` der Tabelle `USERS` in `FIRSTNAME` umbenannt wurde.

Dem checkerberry cockpit XML-Konverter wird der Name des alten Attributs (`NAME`), der Name des zugehörigen Tags (`USERS`) und der neue Name des Attributs (`FIRSTNAME`) übergeben. Das Ergebnis der Umbenennung sieht wie folgt aus:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC "-//ConceptPeople//DTD sample-db 1.0//EN"
"../sample-db.dtd">

<dataset>
  <USERS SURNAME="Simpson" FIRSTNAME="Homer" BIRTHDATE="1946-09-16"/>
  <USERS SURNAME="Simpson" FIRSTNAME="Maggie" BIRTHDATE="2006-09-15"/>

  <SHOP_USERS SURNAME="Smithers" NAME="Waylon" RESERVED=""/>
  <SHOP_USERS SURNAME="Burns" NAME="Montgomery" RESERVED=""/>
</dataset>
```

Beispiel 5.3. Testdaten-Beispiel nach Umbenennung der Attribute

Der Attributname NAME wurde nur für die USERS-Tags umbenannt, obwohl das SHOP_USERS-Tag den Attributnamen NAME ebenfalls verwendet. An dieser Stelle wird deutlich, dass der XML-Konverter kontextsensitiv arbeitet. Die manuelle Umstellung wäre in diesem Fall sehr aufwändig und fehleranfällig. Aufgrund der Situation ist ein reines „Suchen & Ersetzen" von NAME durch FIRSTNAME nicht möglich. Einige Editoren ermöglichen das „Suchen & Ersetzen" über reguläre Ausdrücke. Dieses Vorgehen ist jedoch komplex und somit fehleranfällig.

5.3.1.3. Neue Attribute einfügen

In der Praxis tritt hin und wieder die Situation auf, dass neue Spalten in eine Datenbanktabelle eingefügt werden müssen. Wenn die Werte die Anwendung beeinflussen oder wenn es sich um Pflichtfelder handelt (not null), müssen diese neuen Spalten auch in den Testdaten berücksichtigt werden. Für das obige Beispiel nehmen wir an, dass in der Tabelle USERS die Spalte COUNTRY mit dem Default-Wert DE eingefügt wird.

Für die Anpassung der Testdaten wird dem checkerberry cockpit der Name des Tags (USERS), der Name des neuen Attributs (COUNTRY) und der Wert des Attributs (DE) übergeben. Das Ergebnis der Konvertierung sieht wie folgt aus:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC "-//ConceptPeople//DTD sample-db 1.0//EN"
"../sampledb.dtd">

<dataset>
  <USERS SURNAME="Simpson" NAME="Homer" BIRTHDATE="1946-09-16" COUNTRY="DE"/>
  <USERS SURNAME="Simpson" NAME="Maggie" BIRTHDATE="2006-09-15" COUNTRY="DE"/>

  <SHOP_USERS SURNAME="Smithers" NAME="Waylon" RESERVED=""/>
  <SHOP_USERS SURNAME="Burns" NAME="Montgomery" RESERVED=""/>
</dataset>
```

Beispiel 5.4. Testdaten-Beispiel nach dem Einfügen eines neuen Attributs mit Default-Wert

Die Definition eines Default-Werts ist in vielen Situationen ausreichend. Es besteht jedoch auch die Möglichkeit die Werte aus anderen Attributen des gleichen Datensatzes zu referenzieren. Eine Kombination von festen Texten und referenzierten Werten ist ebenfalls möglich. Als Beispiel könnte für die Tabelle USERS eine neue Spalte FULLNAME eingefügt werden, die den Vor- und Nachnamen beinhaltet. Zu diesem Zweck wird dem checkerberry cockpit XML-Konverter der Default-Wert „ ${NAME} ${SURNAME} " übergeben. Das Ergebnis der Konvertierung sieht dann wie folgt aus:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC "-//ConceptPeople//DTD sample-db 1.0//EN"
"../sample-db.dtd">

<dataset>
  <USERS SURNAME="Simpson" NAME="Homer" BIRTHDATE="1946-09-16"
    FULLNAME="Homer Simpson"/>
  <USERS SURNAME="Simpson" NAME="Maggie" BIRTHDATE="2006-09-15"
    FULLNAME="Maggie Simpson"/>

  <SHOP_USERS SURNAME="Smithers" NAME="Waylon" RESERVED=""/>
  <SHOP_USERS SURNAME="Burns" NAME="Montgomery" RESERVED=""/>
</dataset>
```

Beispiel 5.5. Testdaten-Beispiel nach dem Einfügen eines neuen Attributs mit Referenzen

Bei dem Einfügen eines neuen Attributs werden die referenzierten Attribute aus dem entsprechenden Datensatz extrahiert und als Wert des neuen Attributs eingefügt.

5.3.1.4. Bestehende Attribute löschen

Bei der Änderung der Datenbankstruktur kann es ebenfalls vorkommen, dass Spalten wegfallen. In den Testdaten müssen die entsprechenden Attribute ebenfalls entfernt werden, da anderenfalls die DTD-Validierung fehlschlägt. Spätestens beim Einspielen in die Datenbank kommt es zu einem Fehler, wenn die einzuspielende Spalte in der Datenbank unbekannt ist.

Für das Beispiel nehmen wir an, dass die Spalte `RESERVED` in der Tabelle `SHOP_USERS` entfallen soll. Zu diesem Zweck werden dem checkerberry cockpit XML-Konverter der Name des Tags (`SHOP_USERS`) und der Name des zu löschenden Attributs (`RESERVED`) übergeben. Das Ergebnis der Konvertierung sieht wie folgt aus:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE dataset PUBLIC "-//ConceptPeople//DTD sample-db 1.0//EN"
"../sample-db.dtd">

<dataset>
  <USERS SURNAME="Simpson" NAME="Homer" BIRTHDATE="1946-09-16"/>
  <USERS SURNAME="Simpson" NAME="Maggie" BIRTHDATE="2006-09-15"/>

  <SHOP_USERS SURNAME="Smithers" NAME="Waylon"/>
  <SHOP_USERS SURNAME="Burns" NAME="Montgomery"/>
</dataset>
```

Beispiel 5.6. Testdaten-Beispiel nach dem Löschen von Attributen

Das Attribut wurde aus den Testdaten entfernt.

5.3.1.5. Adressierung von Tags und Attributen

Der checkerberry cockpit XML-Konverter benötigt bei der Konvertierung den Namen des Tags und teilweise auch den Namen des Attributs. Bei diesen Werten handelt es sich immer um die alten Namen der Werte. Folgendes Beispiel verdeutlicht die Problematik. Nehmen wir an, dass in unserem obigen Beispiel die Tabelle `USERS` in `BUDDIES` umbenannt werden soll. Des Weiteren soll in dieser Tabelle eine neue Spalte `FULLNAME` eingefügt werden. Bei der Definition der Parameter für die Anlage des Attributs `FULLNAME` stellt sich nun die Frage, ob das Tag noch über `USERS` oder schon über `BUDDIES` angesprochen werden muss. Wenn die Konvertierung innerhalb eines Schritts, d.h. innerhalb eines definierten Mappings durchgeführt wird, muss der alte Name der Tabelle referenziert werden. Wenn die Konvertierung über zwei verschiedene Aufrufe des XML-Konverters nacheinander erfolgt, muss verständlicherweise der neue Name verwendet werden, da der bisherige Name dann nicht mehr in den Testdaten bekannt ist.

Bei dem Hinzufügen von neuen Attributen können ebenfalls nur bestehende Attribute referenziert werden. Das Referenzieren von neuen Attributen liefert unvorhersehbare Ergebnisse.

5.3.2. Bedienung der grafischen Oberfläche

In diesem Abschnitt wird die Bedienung des checkerberry cockpit XML-Konverters beschrieben. Die folgende Abbildung zeigt die GUI des XML-Konverters.

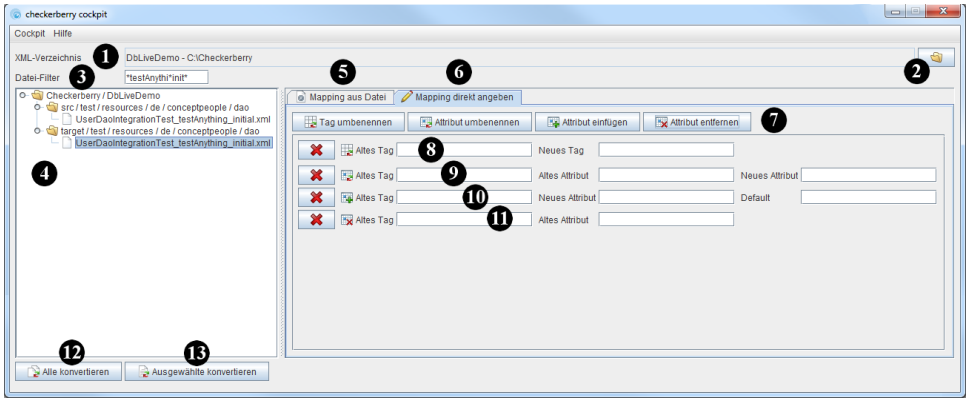


Abbildung 5.1. Grafische Oberfläche checkerberry cockpit XML-Konverter

Der checkerberry cockpit XML-Konverter kann mehrere XML-Dateien nacheinander bearbeiten. Zu diesem Zweck wählt der Benutzer ein XML-Verzeichnis (1) über den Button (2) aus. Daraufhin erscheint ein Dialog zur Auswahl eines Verzeichnisses. Nach der Auswahl des Verzeichnisses werden die gefundenen XML-Dateien in dem Baum (4) angezeigt. Über den Dateifilter (3) kann der Benutzer die Auswahl der XML-Dateien weiter einschränken. Durch die Eingabe von `*initial.xml` würden zum Beispiel nur Dateien mit der Endung `initial.xml` angezeigt werden.

Im nächsten Schritt muss das zu verwendende Mapping definiert werden. Innerhalb des Mappings wird definiert, welche Aktionen (Umbenennungen, Ergänzungen oder Löschungen) der XML-Konverter ausführen soll. Die Mapping-Informationen können auf zwei Wegen definiert werden. Durch die Auswahl des Reiters „Mapping aus Datei" (5) kann das Mapping über eine Excel-Datei definiert werden (siehe Abschnitt 5.3.2.1, „Angabe von Mapping-Dateien"). Über den Reiter „Mapping direkt angeben" (6) kann das Mapping direkt in der grafischen Oberfläche angegeben werden. Zu diesem Zweck existieren vier Buttons (7): „Tag umbenennen", „Attribut umbenennen", „Attribut einfügen", und „Attribut entfernen". Durch das Drücken eines der Buttons wird die Liste der darunterliegenden Komponenten erweitert. In diesen Komponenten müssen zusätzliche Informationen angegeben werden.

Durch das Drücken des Buttons „Tag umbenennen" werden zwei Eingabefelder für den alten und neuen Namen des Tags eingeblendet (8) (siehe Abschnitt 5.3.1.1, „Tags umbenennen"). Generell ist am Anfang jeder Zeile ein Button zum Löschen der Komponenten dieser Zeile vorhanden.

Durch das Drücken des Buttons „Attribut umbenennen" werden drei Eingabefelder für den alten Namen des Tags, den alten Namen des Attributs und den neuen Namen des Attributs eingeblendet (9) (siehe Abschnitt 5.3.1.2, „Attribute umbenennen").

Durch das Drücken des Buttons „Attribut einfügen" werden drei Eingabefelder für den alten Namen des Tags, den neuen Namen des Attributs und den Default-Wert des Attributs eingeblendet (10) (siehe Abschnitt 5.3.1.3, „Neue Attribute einfügen").

Durch das Drücken des Buttons „Attribut entfernen" werden zwei Eingabefelder für den alten Namen des Tags und den alten Namen des Attributs eingeblendet (11) (siehe Abschnitt 5.3.1.4, „Bestehende Attribute löschen").

Nach der Definition der Mapping-Informationen kann die Konvertierung gestartet werden. Der Benutzer kann dabei entweder alle angezeigten Dateien (12) oder lediglich die im Baum ausgewählte Dateien (13) für die Konvertierung verwenden.

5.3.2.1. Angabe von Mapping-Dateien

Der checkerberry cockpit XML-Konverter unterstützt neben der manuellen Angabe der Mapping-Informationen auch dateibasierte Mapping-Informationen. Aktuell werden diese Informationen im MS-Excel-Format (< MS-Excel 2007) angegeben. Die Datei muss zwei Arbeitsblätter mit den Namen „Tags“ und „Attributes“ enthalten.

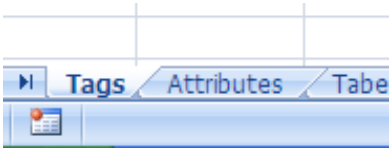


Abbildung 5.2. Arbeitsblätter für Mapping-Informationen

Das Arbeitsblatt „Tags“ enthält die Umbenennungen der zugehörigen Datenbanktabellen. Zu diesem Zweck sind zwei Spalten mit den Überschriften „Old Tag“ und „New Tag“ in dem Arbeitsblatt vorhanden. Das Arbeitsblatt in Abbildung 5.3, „Arbeitsblatt "Tags"“enthält eine Umbenennung: Das Tag `USERS` soll in den Namen `TOONS` umbenannt werden.

	A	B
1	Old Tag	New Tag
2	USERS	TOONS
3		

Abbildung 5.3. Arbeitsblatt "Tags"

Das Arbeitsblatt „Attributes“ enthält die Aktionen, die die Tabellenspalten betreffen. Zu diesem Zweck enthält das Arbeitsblatt die Spalten „Old Tag“, „Old Attribute“, „New Attribute“, „Delete“ und „Default“. Es ist abhängig von der durchzuführenden Aktion, welche Spalten mit Werten belegt werden müssen. In Abbildung 49 wird in Zeile 2 die Tabellenspalte `NAME` der Tabelle `USERS` in `FIRSTNAME` umbenannt. In Zeile 3 wird die Spalte `BIRTHDATE` der Tabelle `USERS` gelöscht. Zu diesem Zweck enthält die Excel-Spalte „Delete“ den Wert „y“. In Zeile 4 wird in der Tabelle `USERS` eine neue Spalte mit dem Namen `NEWNAME` eingefügt. Der Wert wird dabei aus der bestehenden Spalte `SURNAME` übernommen. In Zeile 5 wird ebenfalls eine neue Spalte angelegt. Der Name der Spalte ist `NEWCONST` und enthält immer den konstanten Wert `4711`.

	A	B	C	D	E
1	Old Tag	Old Attribute	New Attribute	Delete	Default
2	USERS	NAME	FIRSTNAME		
3	USERS	BIRTHDATE		y	
4	USERS		NEWNAME		${SURNAME}
5	USERS		NEWCONST		4711
6					

Abbildung 5.4. Arbeitsblatt "Attributes"

Bei der Verwendung des checkerberry cockpit XML-Konverters kann der Benutzer entscheiden, ob die Mapping-Informationen direkt oder über eine Excel-Datei angegeben werden. Die Auswahl erfolgt über den selektierten Reiter (siehe Abbildung 5.5, „checkerberry cockpit XML-Konverter – Auswahl der Mapping-Quelle“). Der Reiter zur Eingabe einer Mapping-Datei enthält lediglich ein Feld zur Eingabe der zu verwendenden Datei (1). Über einen Button (2) wird ein Datei-Browser geöffnet, über den die Mapping-Datei ausgewählt werden kann.

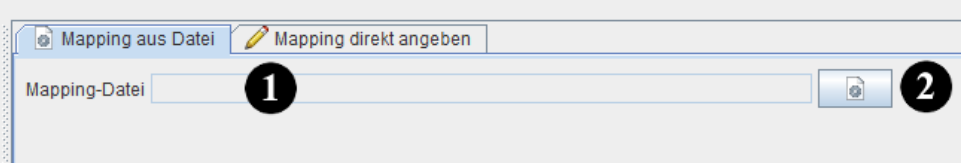


Abbildung 5.5. checkerberry cockpit XML-Konverter – Auswahl der Mapping-Quelle

Kapitel 6. Fragen und Antworten

6.1. Checkerberry test center

6.1.1. Wie kann ich die Versionsnummer einer externen Bibliothek ändern?

Die Versionsangaben in den Maven-Konfigurationen des checkerberry test centers beziehen sich nie auf feste Versionen, sondern geben immer die empfohlene Versionsnummer an. Wird das checkerberry test center in ein Maven-Projekt des Kunden eingebunden, können die Versionsnummern einfach in der Maven-Konfiguration überschrieben werden. Maven richtet sich bei der Auswahl der Versionsnummer an der Entfernung zum Wurzelprojekt, welches in diesem Beispiel das Kundenprojekt ist. Das bedeutet, dass die Definition der Versionen in den `pom.xml` Dateien des Kundenprojekts (Distanz 1) eine höhere Priorität haben, als die Versionen des checkerberry test center (Distanz 2). Dies liegt daran, dass die Maven-Konfigurationen des checkerberry test centers in dem Kundenprojekt lediglich referenziert werden.

Neben dem Standardverfahren von Maven zur Ermittlung der besten Version kann für jede externe Bibliothek die verwendete Versionsnummer geändert werden. Für alle externen Bibliotheken existiert eine Property, die den Wert der Version enthält. Das Property kann durch eine Konfiguration in der `settings.xml` überschrieben werden. Folgendes Code-Beispiel beschreibt die erforderliche Konfiguration der `settings.xml`.

```
…
<profiles>
  <profile>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <org.dbunit.checkerberry.version>2.4.7</org.dbunit.checkerberry.version>
    </properties>
  </profile>
</profiles>
…
```

Beispiel 6.1. Änderung einer Versionsnummer in der settings.xml

Um ein Property in der `settings.xml` zu setzen, muss ein neues Profil angelegt werden. In dem angegebenen Code-Beispiel wird das Profil standardmäßig aktiviert (`<activeByDefault>true</activeByDefault>`). Dann wird die Versionsnummer von DbUnit auf den Wert 2.4.7 gesetzt. Diese Einstellung führt dazu, dass der in dem checkerberry test center konfigurierte Wert überschrieben wird.

Es ist darauf zu achten, dass die `settings.xml` immer dem aktuellen Benutzer zugeordnet ist. Daher muss diese Einstellung bei allen Nutzern angepasst werden. Dies betrifft insbesondere auch die Benutzer, unter deren Namen die Continuous Integration Systeme gestartet werden.

6.1.2. Wie konfiguriere ich die Logging-Einstellungen?

Das checkerberry test center verwendet `slf4j` [slf4j Homepage, 2013] als Logging-Api. Dadurch kann der Benutzer durch eine Bindung (z.B. `slf4j-log4j12-1.7.5.jar`) die konkrete Implementierung des Logging Frameworks wählen, z.B. Log4j [Log4j Homepage, 2010]. Das folgende Code-Beispiel enthält eine mögliche Konfiguration in Form einer `log4j.properties` Datei.

```
log4j.rootLogger=INFO, MyConsApp, MyFileApp
log4j.appender.MyConsApp=org.apache.log4j.ConsoleAppender
log4j.appender.MyConsApp.layout=org.apache.log4j.PatternLayout
log4j.appender.MyConsApp.layout.ConversionPattern=%d{ISO8601} %-5p [%t] %c:
%m%n

log4j.appender.MyFileApp=org.apache.log4j.DailyRollingFileAppender
log4j.appender.MyFileApp.datePattern='.'yyyy-MM-dd_HH-mm
log4j.appender.MyFileApp.file=checkerberry.log
log4j.appender.MyFileApp.layout=org.apache.log4j.PatternLayout
log4j.appender.MyFileApp.layout.ConversionPattern=%d{ISO8601} %-5p [%t] %c:
%m%n

log4j.logger.de.conceptpeople=DEBUG
```

Beispiel 6.2. log4j.properties

Das Code-Beispiel enthält eine mögliche log4j-Konfiguration. Die Konfiguration aktiviert einen Appender, der die Ausgabe auf die Konsole steuert (`MyConsApp`). Des Weiteren wird ein Appender definiert, der Log-Informationen in die Datei checkerberry.log schreibt (`MyFileApp`).

Über den `rootLogger` wird der Standard-Log-Level definiert, der bei der Logging-Ausgabe berücksichtigt wird. In dem Beispiel werden nur Logging-Ausgaben auf INFO-Level und darüber ausgegeben. Als Ausnahme wird für alle Packages, die mit `de.conceptpeople` beginnen, der Log-Level auf DEBUG gesetzt.

Eine detaillierte Auflistung der Konfigurationsmöglichkeiten ist auf der Log4j-Homepage vorhanden.

6.1.2.1. Konfiguration des Logging von Selenium

Selenium verwendet die Java Logging API. Die Konfigurationsdatei von der der Java Logging API liegt unter `JAVA_HOME/jre/lib`. Um eine andere Konfiguration zu verwenden kann die Datei über das Property `java.util.logging.config.file` angegeben werden. Der entsprechende Aufruf mit Maven wäre:

```
mvn clean install -Djava.util.logging.conf.file=C:\settings\logging.properties
```

Beispiel 6.3. Maven Aufruf mit Angabe einer alternativen logging.properties

Ein Konfigurationsbeispiel für die Datei logging.properties ist in Beispiel 6.4, „logging.properties“ angegeben.

```
handlers=java.util.logging.ConsoleHandler

# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level=INFO
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
```

Beispiel 6.4. logging.properties

6.1.3. Welche generellen Fehlermeldungen gibt es?

Tabelle 6.1. Fehlermeldungen des checkerberry test centers

CB-1000	Das Verzeichnis 'XXX' konnte nicht erstellt werden.
Um eine Datei zu speichern, z.B. eine neue DTD, einen Datenbankdump, einen Diff-Report, o.ä., sollte ein neues Verzeichnis erstellt werden. Das Erstellen dieses Verzeichnisses ist fehlgeschlagen.	
CB-1001	Spring hat eine Exception geworfen.

Das checkerberry test center wird in Verbindung mit Spring genutzt. Diese Fehlermeldung kapselt eine Exception aus dem Springframework und tritt z.B. auf, wenn der ApplicationContext nicht geladen werden konnte.	
CB-1002	Unbekanntes Testframework. Bitte lesen sie die Dokumentation dieser Exception (entweder im Benutzerhandbuch oder im Quelltext) für weitere Hinweise zu möglichen Lösungswegen.
Das Testframework konnte nicht ermittelt werden. Wie Sie das checkerberry test center für verschiedene Testframeworks konfigurieren, können Sie unter Abschnitt 2.5.3, „Erzeugen der checkerberry db-Umgebung“ nachlesen.	
CB-1004	Unerwarteter Programmzustand. Diese Exception sollte niemals auftreten. Bitte wenden Sie sich an die ConceptPeople consulting gmbh.
Wir entschuldigen uns für die Unannehmlichkeiten. Bitte teilen Sie uns mit, dass diese Exception aufgetreten ist.	
CB-1005	Der TestConnectorCreator 'XXX' konnte nicht instanziiert werden.
Der über die TestConnectorConfiguration Annotation angegebene *TestConnectorCreator* konnte nicht instanziiert werden.	
CB-1006	Der Name der aktuellen Testmethode konnte nicht bestimmt werden. Bitte lesen sie die Dokumentation dieser Exception (entweder im Benutzerhandbuch oder im Quelltext) für weitere Hinweise zu möglichen Lösungswegen.

Das checkerberry test center konnte den Namen der aktuellen Testmethode nicht bestimmen. Dies hängt damit zusammen, dass einige Testframeworks den Zugriff auf den Methodennamen nur über Umwege ermöglichen. Das checkerberry test center bietet daher für unterschiedliche Frameworks jeweils passende Methoden zur Bestimmung des Namens:
• **JUnit3** erlaubt den direkten Zugriff auf den Methodennamen. Es sind keinerlei andere Methoden nötig.
• **JUnit4** bietet ab der Version 4.7 mit der `@Rule`-Annotation eine einfache Möglichkeit, den Methodennamen zu bestimmen. Um dem checkerberry test center dies zu ermöglichen, fügen Sie bitte folgenden Code in Ihre Testklasse ein:
`@Rule`
`public MethodNameDeterminationRule rule = new MethodNameDeterminationRule();`
In den Versionen zwischen 4.0 bis 4.7 bietet JUnit4 leider keinerlei Möglichkeit, die Testmethode zu bestimmen. Es wird daher empfohlen auf das Spring-Framework zurückzugreifen (s.u.) oder auf eine aktuellere Version von JUnit umzusteigen.
• **JUnit4+Spring:** Das Spring Framework stellt ebenfalls einen Mechanismus zur Verfügung, mit dem sich der Methodenname bestimmen lässt. Um diesen Weg unter JUnit4 zu nutzen, annotieren Sie Ihre Testklasse bitte folgendermassen:
`@RunWith(SpringJUnit4ClassRunner.class)`
`@TestExecutionListeners({ TestMethodNameResolvingExecutionListener.class })`	

- **TestNG:** In TestNG lässt sich die Testmethode einfach bei der Verwendung der Annotation `@BeforeMethod` angeben:

```
@BeforeMethod
public void setUp(Method testMethod) {
...
environment.setUp(this, method);
}
```

Bei jeder dieser Methoden reicht es aus, sie im abstrakten Basistestfall anzuwenden. Auf diese Weise können alle abgeleiteten Testfälle die Methode ohne Code-Duplizierung nutzen. (siehe Abschnitt 2.5.3, „Erzeugen der checkerberry db-Umgebung“)

CB-1007	Die Testmethode 'XXX' existiert nicht.
Die Testmethode mit dem Namen XXX konnte nicht gefunden werden. Möglicherweise gibt es ein Problem mit der Einstellung des Testframeworks (siehe Abschnitt 2.5.3, „Erzeugen der checkerberry db-Umgebung“).	
CB-1008	Der Bezeichner 'XXX' enthält das XML Steuerzeichen 'YYY' und kann deshalb nicht verwendet werden.
Es wurde der String XXX für einen konfigurierbaren Bezeichner gewählt. Dieser wird in XML Dateien verwendet und darf deshalb kein XML Steuerzeichen enthalten. Bitte entfernen Sie den Substring YYY aus dem Bezeichner.	
CB-1009	Der Wert 'XXX' konnte nicht in den Typ 'YYY' konvertiert werden.
Bei der Verwendung der Validatoren werden aus String-Representationen spezielle Java-Klassen z.B. java.util.Date erzeugt. Wenn eine Zeichenkette XXX nicht in einen Typ YYY konvertiert werden konnte, wird diese Fehlermeldung erzeugt.	
CB-1010	Der Wert "XXX" konnte nicht in eine BCD-Darstellung konvertiert werden, da die oberen 4 Bit ("Y") oder die unteren 4 Bit ("Z") nicht durch eine Ziffer darstellbar sind.
Dieser Fehler tritt auf, wenn in checkerberry db die Verwendung des BCD-Formats aktiviert ist und der Wert XXX nicht im BCD-Format dargestellt werden kann. Dies ist der Fall, wenn der Wert mindestens ein Byte enthält, bei dem das obere (Y) oder untere Nibble (Z) einen Wert enthält der nicht zwischen 0 und 9 liegt. Die Werte Y und Z werden als Zahl angegeben z.B. 9 oder 11.	
CB-1011	Die BCD-Darstellung >XXX< konnte nicht in ein Byte-Array konvertiert werden, da sie ein ungültiges Zeichen ("Y") enthält.
Dieser Fehler tritt auf, wenn ein String im BCD-Format ein ungültiges Zeichen Y enthält z.B. „X'1ö'“ und daher nicht in ein Byte-Array konvertiert werden konnte.	

CB-1012	Die BCD-Darstellung >XXX< hat eine ungültige Länge. Die Anzahl der Ziffern muss ein Vielfaches von 2 sein.
Dieser Fehler tritt auf, wenn ein String im BCD-Format mit ungültiger Länge angegeben wird z.B. „X'123'".	
CB-1013	Die BCD-Darstellung >XXX< hat ein ungültiges Präfix. Das gültige Präfix lautet >Y<.
Dieser Fehler tritt auf, wenn ein String im BCD-Format das falsche Präfix verwendet z.B. „X123". Das erwartete Präfix („X'") wird in der Meldung ebenfalls angezeigt.	
CB-1014	Die BCD-Darstellung >XXX< hat ein ungültiges Suffix. Das gültige Suffix lautet >Y<.
Dieser Fehler tritt auf, wenn ein String im BCD-Format das falsche Suffix verwendet z.B. „X'123". Das erwartete Präfix („'") wird in der Meldung ebenfalls angezeigt.	

6.2. Checkerberry db

Dieses Kapitel listet eine Reihe von häufig gestellten Fragen und deren Antworten auf.

6.2.1. Wie erstelle ich eine neue DTD?

Die Erzeugung einer neuen DTD erfolgt über die Methode `createDtd` der Klasse `DbTestHandler`. Die Methode kann temporär innerhalb eines Testfalls ausgeführt werden und erzeugt dann eine neue DTD anhand der aktuellen Datenbankstruktur.

Zusätzlich erzeugt checkerberry db automatisch eine neue DTD, wenn initiale Testdaten nicht in die Datenbank eingespielt werden können. Der Name der neuen DTD hat das Präfix „`new-`" und wird im Klassenpfad abgelegt. Danach muss die erstellte DTD umbenannt und in den Source-Baum kopiert werden. Weitere Informationen sind in Abschnitt 2.4.1.5, „Anlegen einer neuen Testdaten-DTD"beschrieben.

6.2.2. Wie kann ich die Datenbank-Statements loggen?

Die Konfiguration zum Loggen der Batch-Statements ist in Abschnitt 2.4.17, „Aktivieren des Datenbank-Loggings" beschrieben.

6.2.3. Warum bekomme ich bei einem Datenbank-Dump einen OutOfMemoryError?

Bei der Erstellung eines Datenbank-Dumps werden die Daten aus der Datenbank in den Speicher geladen. Wenn die Datenbank viele Daten beinhaltet, reicht der reservierte Speicherplatz nicht aus, sodass es zu einem `OutOfMemoryError` kommt. Das Problem kann durch eine Konfigurationsänderung in DbUnit gelöst werden (siehe Abschnitt 2.4.20, „Anpassen von DbUnit-Properties und Features").

6.2.4. Meine Testdatei ist da, wird aber nicht gefunden?

Dieses Verhalten kann mehrere Ursachen haben. Zum einen muss geprüft werden, ob die Datei tatsächlich mit dem korrekten Namen im Klassenpfad liegt. Wurde ggf. etwas an der Namenskonvention zum Auffinden der Testdaten geändert? Die Log-Ausgaben sollten auf diese Punkte entsprechende Hinweise geben. Eine weitere Ursache kann in der Länge des absoluten Dateinamens liegen. Wenn der Dateiname inklusive der Pfadangaben über 256 Zeichen lang ist, kann das Betriebssystem die Datei nicht mehr lesen, sodass die Datei nicht gefunden wird. Die einzige Lösung besteht darin, den Namen der Testdatei zu kürzen.

6.2.5. Meine inkludierten Testdaten (LOCATION=RELATIVE) werden nicht gefunden?

Checkerberry db sucht die inkludierten Testdaten relativ zu den einbindenden Testdaten. Es ist zu beachten, dass die einzubindenden Testdaten aus dem Klassenpfad verwendet werden. Die inkludierten Testdaten müssen somit relativ zu der Testdatendatei in dem Klassenpfad vorhanden sein. Bitte prüfen Sie unbedingt die Log-Ausgaben.

6.2.6. Wie wird die Verbindung zwischen Datenbank und erwarteten Daten hergestellt?

Bei der Überprüfung der erwarteten Ergebnisse muss checkerberry db eine Zuordnung zwischen erwarteten und tatsächlichen Daten herstellen. Auf diese Art und Weise werden neue, fehlende und geänderte Datensätze erkannt. Zu diesem Zweck muss für jede zu überprüfende Tabelle ein Eintrag in den Tabellenbeschreibungen vorgenommen werden. Pro Tabelle werden Lookup-Keys definiert, die für die Identifizierung einzelner Zeilen verwendet werden. Bei den Lookup-Keys handelt es sich um Namen der Spalten der Tabelle, die für die Identifizierung herangezogen werden. Lookup-Keys sind konzeptionell somit identisch zu Primary Keys.

Es wird empfohlen fachliche Schlüssel für die Lookup-Keys zu verwenden (siehe Abschnitt 7.2.7, „Verwende fachliche Lookup-Keys"). Da Primary Keys häufig technische Schlüssel sind, verzichtet checkerberry db darauf, diese generell als Lookup-Keys zu verwenden.

6.2.7. Warum liest checkerberry db nicht die korrekten Werte aus der Datenbank?

Wenn checkerberry db bei der Überprüfung von erwarteten Testdaten unerwartete Werte aus der Datenbank liest, liegt dies mit hoher Wahrscheinlichkeit an einem Fehler im Transaktionsverhalten. Checkerberry db kommuniziert über eine eigene JDBC-Verbindung mit der Datenbank, sodass diese Statements nicht Teil der Transaktion der zu testenden Komponenten sind. Wenn die Transaktion noch geöffnet ist, während checkerberry db die erwarteten Testdaten prüft, sind die Änderungen durch die zu testenden Komponenten für checkerberry db nicht sichtbar. Diese Situation tritt auf, wenn die Transaktion der zu testenden Komponenten in der Setup-Phase geöffnet und erst nach dem Vergleich z.B. in der Teardown-Phase geschlossen wird. In dieser Konstellation muss die Transaktion vor der Überprüfung durch checkerberry db manuell geschlossen werden, damit die Änderungen in der Datenbank sichtbar werden.

Eine detaillierte Beschreibung der Problematik ist in Kapitel Abschnitt 2.5.3.2, „Einstellen der Transaktionsverwaltung" aufgeführt.

6.2.8. Was sind Lookup-Keys?

Bei der Überprüfung von Testdaten mit Datenbankinhalten ist die Zuordnung von Datensätzen aus den Testdaten zu Datensätzen aus der Datenbank erforderlich. Zu diesem Zweck werden die sogenannten Lookup-Keys verwendet. Das Konzept der Lookup-Keys ähnelt dem der Primärschlüssel. Für jede Tabelle wird eine Menge von Spalten definiert, die den Lookup-Key für diese Tabelle bilden. Durch die Spaltenwerte sind die einzelnen Datensätze eindeutig identifizierbar. Dadurch ist eine Zuordnung von Testdaten auf Datenbankinhalte möglich.

Im Gegensatz zu Primärschlüsseln werden für Lookup-Keys in der Regel fachliche und keine technischen Schlüssel verwendet. Insbesondere bedeutet dies, dass der Lookup-Key von dem Primärschlüssel einer Tabelle abweichen kann. Dies ist der wesentliche Grund, warum die neue Bezeichnung „Lookup-Key" eingeführt wurde.

Darüber hinaus können Lookup-Keys einer Tabelle variable pro Testmethode angepasst werden. Dies kann sinnvoll sein, wenn der standardmäßig definierte Lookup-Key in speziellen Testsituationen nicht eindeutig ist.

6.2.9. Muss ich immer eine DTD für die Testdaten verwenden?

Bei der Verwendung von checkerberry db ist die Verwendung einer DTD vorgeschrieben. Checkerberry ist an dieser Stelle somit restriktiver als DbUnit. Warum ist das so?

Die DTD definiert die Struktur der zu testenden Datenbank. Sie enthält die Namen der Tabellen in der Reihenfolge ihrer Abhängigkeiten, definiert für jede Tabelle die vorhanden Spalten und legt fest, welche Spalten Pflichtangaben (`not nullable`) enthalten.

DbUnit unterstützt bei der Überprüfung von Testdaten mit der Datenbank zwei verschiedene Vorgehensweisen. Zum einen kann eine DTD verwendet werden. DbUnit prüft dann für jede angegebene Tabelle die Gleichheit der Werte für alle Spalten.

Zum anderen kann DbUnit ohne DTD verwendet werden. In dieser Situation werden die zu überprüfenden Spalten einer Tabelle direkt aus den Testdaten ermittelt. Der erste Eintrag der Testdaten legt fest, welche Spalten bei der Überprüfung berücksichtigt werden sollen. Enthalten die Testdaten zum Beispiel vier Datensätze für die Tabelle `USERS` und der erste Eintrag beinhaltet die Spalten `ID`, `NAME` und `SURNAME`, dann werden nur diese drei Spalten überprüft – auch für die drei folgenden Datensätze. Der Vorteil dieses Vorgehens besteht darin, dass dynamisch festgelegt werden kann, welche Spalten überprüft werden sollen. Der Nachteil liegt darin, dass dieses Verhalten sehr intransparent ist.

Es existieren im Wesentlichen zwei Probleme bei dem Verzicht auf eine DTD. Das erste Problem besteht darin, dass durch die Definition des ersten Testdatensatzes aus Versehen Spalten von der Überprüfung ausgeschlossen werden. In dem obigen Beispiel würden die Werte der Spalte `BIRTHDATE` nicht geprüft werden, wenn sie in den drei folgenden Datensätzen verwendet werden würden. In diesem Fall würde der Test erfolgreich durchlaufen, obwohl Abweichungen in Spalten vorhanden sind. Das zweite Problem betrifft die Behandlung von `null`-Werten. Innerhalb der Testdaten werden `null`-Werte dadurch definiert, dass die zugehörigen Spalten in dem Datensatz nicht aufgenommen werden. Dies führt zu einer doppelten Bedeutung nicht vorhandener Spalten. In dem ersten Datensatz einer Tabelle bedeuten nicht vorhandene Spalten, dass diese nicht überprüft werden sollen. In den folgenden Datensätzen bedeuten sie, dass die Werte `null` sind. Diese Situation führt schnell zu Problemen, wenn in dem ersten Datensatz eine Spalte einen `null`-Wert enthält, der geprüft werden soll. Diese Situation tritt sehr häufig ein und kann ohne DTD nicht gelöst werden.

Aus der Erfahrung heraus kann DbUnit aus den oben beschriebenen Gründen sinnvoll nur mit einer DTD verwendet werden. Anderenfalls ergeben sich große Fehlerpotentiale oder wilde Work-Arounds, die der Verständlichkeit der Tests und der Testdaten schaden. In der DbUnit-Dokumentation wird im Übrigen ebenfalls die Verwendung einer DTD empfohlen.

Darüber hinaus ergeben sich durch die Verwendung einer DTD weitere Vorteile. Die Testdaten können bereits während der Entwicklung auf Gültigkeit validiert werden. Des Weiteren bieten die gängigen Entwicklungsumgebungen Auto-Vervollständigungen bei der Erstellung der XML-Testdaten an, sodass die Erstellung der Testdaten deutlich beschleunigt wird.

6.2.10. Warum bekomme ich die Fehlermeldung trotz korrekter Testdaten?

Bei der Verwendung von Testdaten kann es zu folgender Fehlermeldung kommen:

java.lang.RuntimeException: org.dbunit.dataset.DataSetException: Line XX: The content of element type "dataset" must match "(INCLUDE*,TABLE1*, TABLE2*, ..., TABLEn*)"

Diese Fehlermeldung erscheint, wenn die Reihenfolge der Tabellen innerhalb der Testdaten nicht der vorgegebenen Reihenfolge aus der DTD entspricht. Die Reihenfolge ist jedoch wichtig, da z.B. beim Einspielen von Testdaten die Fremdschlüsselbeziehungen berücksichtigt werden müssen. Anderenfalls kann es passieren, dass ein Testdatensatz in die Datenbank eingespielt wird, der auf einen noch nicht vorhandenen Datensatz verweist. In diesem Fall würde die Datenbank eine Fehlermeldung wegen einer Constraint-Verletzung werfen. In der DTD berücksichtigt die Reihenfolge der Tabellen diese Abhängigkeiten, sodass diese Reihenfolge als bindend für alle Testdaten verwendet wird.

Das Auftreten dieses Laufzeitfehlers kann bereits während der Entwicklung verhindert werden, wenn die Testdaten bereits in der Entwicklungsumgebung gegen die DTD validiert werden.

6.2.11. Warum ändert sich die Reihenfolge der Tabellen in der DTD?

In der DTD wird die Struktur der Datenbank beschrieben. Dies umfasst u.a. die Liste der Tabellennamen, die in der Datenbank vorhanden sind. Die Reihenfolge ist dabei festgelegt. Bei der Generierung einer neuen DTD aus der Datenbank kann es sein, dass sich die Reihenfolge der Tabellen ändert. Um diesen Sachverhalt zu verstehen, muss man sich anschauen, wie die Reihenfolge ermittelt wird.

Zunächst werden die Namen der Tabellen alphabetisch sortiert. Wenn die Tabellen keine Fremdschlüsselbeziehungen haben, ist das die Reihenfolge der Tabellen in der DTD. Existieren Fremdschlüsselbeziehungen zwischen den Tabellen, werden diese aufgelöst, indem die Reihenfolge der Tabellen so angepasst wird, dass bei dem Einspielen der Testdaten in der festgelegten Reihenfolge keine Fremdschlüsselverletzungen auftreten können. Die Reihenfolge der Tabellen ist somit abhängig von den vorhandenen Fremdschlüsselbeziehungen. Ändert sich etwas an diesen Beziehungen, kann sich dadurch ebenfalls eine neue Reihenfolge der Tabellen ergeben.

6.2.12. Welche Fehlermeldungen gibt es in checkerberry db?

Tabelle 6.2. Fehlermeldungen von checkerberry db

CB-DB-1000	Der Ausführungskontext ist nicht definiert.
Diese Fehlermeldung erscheint, wenn innerhalb von checkerberry db auf den Ausführungskontext zugegriffen wird, dieser aber nicht vorhanden sind. Der Ausführungskontext ist dabei ein Container, der verschiedene konkrete Kontexte z.B. den ParameterContext enthält.	
CB-DB-1001	Die Tabellenbeschreibung für die Tabelle 'XXX' ist nicht definiert.
Die Ermittlung der Lookup-Keys für die Tabelle `XXX` konnte nicht durchgeführt werden, da für die Tabelle `XXX` keine Tabellenbeschreibung definiert ist. Um dieses Problem zu beheben, muss in der Implementierung des Interfaces `DatabaseDescriptionCallback` eine Tabellenbeschreibung für die Tabelle `XXX` angelegt werden.	
CB-DB-1002	Die Lookup-Keys für die Tabelle 'XXX' identifizieren nicht eindeutig die Zeilen der Tabelle.
Diese Fehlermeldung tritt bei der Erzeugung eines Diff-Reports auf, wenn die definierten Lookup-Keys der Tabelle `XXX` die Zeilen der Tabelle nicht eindeutig identifizieren. Das Ergebnis des Diff-Reports wäre dann verfälscht, da die Zuordnung von erwarteten und vorhandenen Daten nicht eindeutig möglich ist.	

CB-DB-1003	Der Kontext für die Testklasse 'XXX' und die Testmethode 'YYY' konnte nicht ermittelt werden.
In der Setup-Phase wird für die zu startende Testmethode ein Kontext aufgebaut, der den Namen der Testklasse und –methode beinhaltet und vorhandene Annotationen auswertet. Diese Fehlermeldung wird geworfen, wenn bei der Erstellung dieses Kontextes ein Fehler auftritt.	
CB-DB-1004	Für die Konfiguration der Datenbankbeschreibung ist kein DatabaseDescriptionCallback definiert.
Diese Fehlermeldung erscheint, wenn innerhalb der checkerberry db-Bridge kein Callback für die Angabe der Datenbankbeschreibung definiert wurde.	
CB-DB-1005	Die initialen Testdaten konnten nicht gelesen werden.
Beim Einlesen der initialen Testdaten ist ein Fehler aufgetreten.	
CB-DB-1006	Die Testdaten mit dem Suffix 'XXX' konnten nicht gelesen werden.
Beim Einlesen der Testdaten mit dem Suffix `XXX` ist ein Fehler aufgetreten.	
CB-DB-1007	Die Daten aus der Datenbank konnten nicht gedumpt werden.
Bei der Erstellung eines Datenbank-Dumps ist ein Fehler aufgetreten, sodass die Daten nicht in einer XML-Datei gespeichert werden konnten.	
CB-DB-1008	Für das Include 'XXX' ('YYY') wurde keine Datei gefunden.
Diese Meldung erscheint, wenn eine Testdatendatei eine andere Datei XXX einbindet, für die keine Testdaten unter der Location `YYY` (`RELATIVE`, `ABSOLUTE`, `CLASSPATH`) gefunden werden konnten.	
CB-DB-1009	Für das Include 'XXX' ('YYY') konnte die Datei 'ZZZ' nicht eingelesen werden.
Diese Meldung erscheint, wenn eine Testdatendatei eine andere Datei XXX einbindet, für die unter der Location `YYY` (`RELATIVE`, `ABSOLUTE`, `CLASSPATH`) die Datei `ZZZ` gefunden wurde, die aber nicht eingelesen werden konnte.	
CB-DB-1010	Die Konfiguration der DTD-Informationen ist nicht vorhanden.
Diese Meldung wird ausgegeben, wenn die DTD-Informationen in der checkerberry db-Konfiguration nicht angegeben wurden. Zur Behebung dieses Fehlers müssen die DTD-Informationen (`public identifier` und relativer Dateiname zum Klassenpfad) in der Konfiguration eingetragen werden (siehe Abschnitt 2.5.2.3, „Konfiguration von checkerberry db mit dem DbConfigurationCallback“).	
CB-DB-1011	Die DTD-Datei 'XXX.dtd' ist nicht vorhanden.

Diese Meldung wird ausgegeben, wenn die konfigurierte DTD mit dem Namen `XXX.dtd` nicht gefunden werden konnte.	
CB-DB-1012	Der Zugriff auf die Datenbank-URL 'jdbc:XXX' wurde nicht erlaubt. Bitte überprüfen Sie die konfigurierten White- und Blacklist-Einstellungen.
Diese Meldung erscheint, wenn die aktuelle JDBC-URL (`jdbc:XXX`) nicht freigeschaltet wurde. Wenn der Zugriff für die angegebene JDBC-URL erlaubt werden soll, muss die Whitelist-Konfiguration angepasst werden.	
CB-DB-1015	Die checkerberry db-Umgebung (CheckerberryDbEnvironment) fehlt. Bitte erstellen Sie die Umgebung in der Setup-Phase.
Diese Meldung erscheint bei dem Zugriff auf die checkerberry db-Umgebung, wenn diese nicht erzeugt wurde. Das bedeutet, dass die Umgebung in der Setup-Phase nicht erzeugt wurde, da ggf. die Testklasse nicht von der korrekten Oberklasse erbt. Bitte achten Sie auch darauf, dass sie die Umgebung in der Teardown-Phase durch den Aufruf der Methode tearDown wieder freigeben.	
CB-DB-1016	Ungültiger Index 'XXX' beim Zugriff auf die Liste der Tabellen mit der Größe 'YYY'.
Innerhalb von checkerberry db wird häufig über Listen von Tabellen iteriert. Diese Fehlermeldung erscheint, wenn der Iterator auf einen ungültigen Index verweist.	
CB-DB-1017	In der Spalte 'XXX' ist ein leerer Parameter enthalten.
Diese Meldung erscheint, wenn ein leerer Parameter `${}` in der Spalte `XXX` in den erwarteten Testdaten definiert wurde. Parameternamen müssen mindestens ein Zeichen beinhalten.	
CB-DB-1018	Für den Parameter 'XXX' in Tabelle A, Zeile B, Spalte C wurde keine Ersetzung angegeben.
Diese Meldung erscheint, wenn in den erwarteten Testdaten der Parameter `${XXX}` definiert wurde und in dem dazugehörigen Test kein Wert für diesen Parameter definiert wurde. Das Problem lässt sich dadurch beheben, dass ein Wert für den Parameter definiert wird. Alternativ kann der Parameter auch aus den Testdaten entfernt werden oder die Spalte zu der Liste der auszuschließenden Spalten hinzugefügt werden. Es ist eine fachliche Entscheidung, welcher Lösungsweg in dieser Situation eingeschlagen wird.	
CB-DB-1020	Es wurden keine initialen Testdaten gefunden.
Checkerberry db ermöglicht die Überprüfung des aktuellen Datenbankinhalts gegen die initialen Testdaten. Diese Fehlermeldung wird geworfen, wenn bei der Überprüfung der Datenbank gegen die initialen Testdaten keine initialen Testdaten vorhanden sind.	
CB-DB-1022	Der Report konnte nicht erstellt werden.

Diese Meldung wird verwendet, wenn während der Erstellung eines Reports ein Fehler auftritt. Die Ursache des Fehlers wird dieser Meldung angehängt.	
CB-DB-1023	Das Template 'XXX' wurde nicht gefunden.
Die Reports werden intern durch die Verwendung von Templates erzeugt. Diese Meldung tritt auf, wenn das benötigte Template `XXX` zur Erstellung eines Reports nicht vorhanden ist.	
CB-DB-1025	Uninterpretierbarer Funktionsaufruf in Tabelle A, Zeile B, Spalte C bei Indizes D
Ein (nicht-maskierter) Funktionsaufruf in dem angegebenen Feld konnte nicht interpretiert werden. Die häufigsten Ursachen hierfür sind Syntaxfehler, das Maskieren nur eines Teils des Funktionsaufrufes oder Fehler beim Schachteln von Funktionsaufrufen. Die Fehlermeldung gibt eine Liste mit den Indizes der einzelnen, nicht interpretierbaren Funktionsaufrufs-Teile an.	
CB-DB-1026	Unbekannte Funktion XXX in Tabelle A, Zeile B, Spalte C bei Index D
Es wurde versucht, eine für checkerberry db unbekannte Funktion aufzurufen. Entweder der Name der Funktion ist falsch geschrieben oder es handelt sich um eine nutzerdefinierte Funktion, die noch nicht registriert wurde (siehe Beispiel 2.31, „Registrieren der AddFunction“).	
CB-DB-1027	Ungültige Konfiguration: XXX darf kein Prefix von YYY enthalten.
Bei der ersten Verwendung einer neuen Syntax für Parameter oder Funktionsaufrufe wird diese auf Konsistenz hin überprüft. Diese Fehlermeldung besagt, dass dabei ein Fehler festgestellt wurde und gibt die beiden kollidierenden Syntax-Elemente an, um eine Rekonfiguration zu ermöglichen.	
CB-DB-1030	Validierung des Trennzeichens für Funktionsargumente fehlgeschlagen (X): muss ein einzelnes Zeichen sein.
Das Trennzeichen für Funktionsargumente muss ein einzelnes Zeichen sein.	
CB-DB-1031	Validierung des Maskierungszeichens fehlgeschlagen (X): muss ein einzelnes Zeichen sein.
Das Maskierungszeichen muss ein einzelnes Zeichen sein.	
CB-DB-1032	Validierung der Funktion fehlgeschlagen: Darf nicht ‚null‘ sein.
Die zu registrierende Funktion darf nicht ‚null‘ sein.	
CB-DB-1033	Validierung des Parameter-Prefix fehlgeschlagen: Darf nicht ‚null‘ und nicht leer sein.
Der Parameter-Prefix darf nicht ‚null‘ und nicht leer sein.	

CB-DB-1034	Validierung des Parameter-Suffix fehlgeschlagen: Darf nicht ‚null' und nicht leer sein.
Der Parameter-Suffix darf nicht ‚null' und nicht leer sein.	
CB-DB-1035	Validierung des Funktionsaufruf-Prefixes fehlgeschlagen: Darf nicht ‚null' und nicht leer sein.
Der Funktionsaufruf-Prefix darf nicht ‚null' und nicht leer sein.	
CB-DB-1036	Validierung der öffnenden Funktionsaufruf-Klammer fehlgeschlagen: Darf nicht ‚null' und nicht leer sein.
Die öffnende Klammer für Funktionsaufrufe darf nicht ‚null' und nicht leer sein.	
CB-DB-1037	Validierung der schließenden Funktionsaufruf-Klammer fehlgeschlagen: Darf nicht ‚null' und nicht leer sein.
Die schließende Klammer für Funktionsaufrufe darf nicht ‚null' und nicht leer sein.	
CB-DB-1038	Argumente der Funktion sollten dem Format „+3 days" oder „-2 years" entsprechen. XXX ist kein gültiges Argument.
Die Funktion `->now()` oder `->today()` wurde mit einem ungültigen Argument aufgerufen. Ein häufiger Fehler ist das Einfügen von Leerzeichen vor oder nach den Argumenten.	
CB-DB-1039	Die Funktion XXX in Tabelle A, Zeile B, Spalte C benötigt Argumente.
Die Funktion XXX muss mit mindestens einem Argument aufgerufen werden.	
CB-DB-1040	DbUnit hat eine Exception geworfen.
Kapselt eine Exception des DbUnit Frameworks.	
CB-DB-1041	Der Bezeichner XXX kann nicht für die Tabelle YYY verwendet werden.
Es wurde ein ungültiger Bezeichner für eine konfigurierbare Tabelle gewählt. Dies muss in der DbConfigurationCallback geändert werden.	
CB-DB-1042	Wenn für eine Klasse Initialdaten definiert sind, darf die Annotation ClearTables nicht an dieser Klasse gesetzt werden.
Wenn Initialdaten für eine Klasse vorliegen, darf diese Klasse nicht mit ClearTables annotiert werden. Es ist hingegen zulässig, eine Methode mit ClearTables zu annotieren, wenn nur für die Klasse Initialdaten vorliegen und umgekehrt.	

CB-DB-1043	Wenn für eine Methode Initialdaten definiert sind, darf die Annotation ClearTables nicht an die Methode gesetzt werden.
Wenn Initialdaten für eine Methode vorliegen, darf diese Methode nicht mit ClearTables annotiert werden. Es ist hingegen zulässig, eine Methode mit ClearTables zu annotieren, wenn nur für die Klasse Initialdaten vorliegen und umgekehrt.	
CB-DB-1044	Für die Tabelle XXX sind sowohl Daten als auch ein EMPTY_TABLE Tag definiert.
Das EMPTY_TABLE Tag soll genutzt werden, um explizit anzugeben, dass eine Tabelle leer ist. Wenn zusätzlich Daten für diese Tabelle definiert wurden, deutet das auf einen fachlichen Fehler hin. Wenn Dateien inkludiert werden, ist es möglich, dass in der einen Datei ein Datensatz für eine Tabelle definiert wurde, die in einer anderen Datei als leer markiert ist.	
CB-DB-1048	Die DTD Datei 'XXX' konnte nicht gelesen werden.
Diese Meldung erscheint, wenn beim Lesen der angegebenen DTD Datei ein Fehler aufgetreten ist.	
CB-DB-1049	Der Auto-Parameter "XXX" hat bereits den Wert "YYY". Der neue Wert "ZZZ" kann nicht zugeordnet werden.
Diese Meldung erscheint, wenn der Wert eines Autoparameters nicht eindeutig ermittelt werden kann.	
CB-DB-1051	Für die Tabelle "XXX" und die Spalte "YYY" wurde kein aktiver Validator gefunden.
Diese Meldung erscheint, wenn für eine Spalte kein aktiver Validator gefunden wurde.	
CB-DB-1052	Für die Tabelle "XXX" und die Spalte "YYY" wurde ein unbekannter Validator definiert: "ZZZ".
Diese Meldung erscheint, wenn für die Spalte YYY der Tabelle XXX ein Validator mit der Id ZZZ verwendet werden soll, der an der Konfiguration nicht registriert wurde.	
CB-DB-1053	Die Reihenfolge der Tabellen innerhalb der XML-Testdaten entspricht nicht der Reihenfolge aus der DTD. Die Reihenfolge muss lauten [XXX-1, XXX-2, ..., XXX-n].
Diese Fehlermeldung erscheint, wenn die Reihenfolge der Tabellen in den Testdaten nicht der Reihenfolge der Tabellen in der DTD entspricht. Die Tabellenreihenfolge in der DTD legt fest, in welcher Reihenfolge die Tabellen in die Datenbank eingespielt werden. Diese Reihenfolge ist wichtig, da sie Fremdschlüsselbeziehungen berücksichtigt und somit verhindert, dass Constraint-Verletzungen beim Einspielen der Testdaten auftreten. Diese Fehlermeldung listet die Tabellen in der Reihenfolge auf, in der sie angegeben werden müssen. Die Liste enthält dabei nur Tabellen, die tatsächlich in den fehlerhaften Testdaten verwendet wurden.	

CB-DB-1054	Es wurde versucht innerhalb einer Read-Only-Verbindung schreibend auf die Datenbank zuzugreifen.
Diese Fehlermeldung erscheint, wenn schreibend auf eine JDBC-URL zugegriffen wurde, die als Read-Only konfiguriert ist.	
CB-DB-1055	Bitte geben Sie in der Konfiguration eine JDBC-URL (z.B. "jdbc:hsqldb:mem:sample-test") ein, wenn sie den Standard JDBC Database-Connector verwenden.
Diese Fehlermeldung erscheint, wenn die Standard-Implementation des `DatabaseConnectors` verwendet wird, ohne dass in der Konfiguration die JDBC-URL der zu verwendenden Datenbank angegeben wurde. Ohne diese URL ist checkerberry db nicht in der Lage, eine Verbindung zu der gewünschten Datenbank herzustellen.	
CB-DB-1056	Bitte geben Sie in der Konfiguration einen JDBC-Treibernamen (z.B. "org.hsqldb.jdbcDriver") ein, wenn sie den Standard JDBC Database-Connector verwenden.
Diese Fehlermeldung erscheint, wenn die Standard-Implementation des `DatabaseConnectors` verwendet wird, ohne dass in der Konfiguration der zu verwendenden Datenbanktreiber angegeben wurde. Ohne den Treiber ist checkerberry db nicht in der Lage, eine Verbindung zu der gewünschten Datenbank herzustellen.	
CB-DB-1057	Der Datenbankwert "XXX" konnte nicht in einen String konvertiert werden.
Dieser Fehler tritt auf, wenn die Konvertierung eines Datenbankinhalt in einen String fehlschlägt, z.B. bei der Konvertierung von Binärdaten zur Erstellung eines Datenbank-Dumps.	
CB-DB-1058	Die folgenden Autoparameter konnten nicht aufgelöst werden: XXX.
Dieser Fehler tritt auf, wenn eine Reihe von Autoparametern nicht ermittelt werden konnten. Die nicht ermittelbaren Autoparameter werden als Liste XXX angegeben.	
CB-DB-1059	Die Reihenfolge der Datenbanktabellen konnte aufgrund ungültiger Abhängigkeiten nicht berechnet werden (siehe LOG(error)).
Dieser Fehler tritt auf, wenn eine neue DTD berechnet werden soll und die Datenbanktabellen ungültige, in der Regel zyklische, Abhängigkeiten haben. In der Log-Datei finden sich weitere Informationen zu den Tabellen und ihren Abhängigkeiten.	
CB-DB-1060	Die Anzahl der Spalten ('TABLE') für den Primärschlüssel ist unterschiedlich OLD! =NEW
Der Fehler tritt auf, wenn beim Zusammenfügen von Tabellendaten unterschiedliche Metadaten für eine Tabelle ermittelt wurden. Die Anzahl der Spalten (OLD und NEW) für den Primärschlüssel unterscheidet sich bei der Tabelle 'TABLE'.	

CB-DB-1061	Die Spalten ('TABLE') passen nicht zueinander 'OLD'!='NEW'
Der Fehler tritt auf, wenn beim Zusammenfügen von Tabellendaten unterschiedliche Metadaten für eine Tabelle ermittelt wurden. Die Informationen der vorhanden Spalten der Tabelle 'TABLE' passen nicht zueinander. Die Liste der Spaltennamen (OLD und NEW) werden als Liste ausgegeben.	
CB-DB-1062	Die Spalten ('TABLE') passen nicht zueinander. Das Attribut 'COLUMN.ATTR' hat unterschiedliche Werte: 'OLD'!='NEW'
Der Fehler tritt auf, wenn beim Zusammenfügen von Tabellendaten unterschiedliche Metadaten für eine Tabelle ermittelt wurden. Dabei ist es zu Abweichungen des Attributes ATTR der Spalte COLUMN in der Tabelle TABLE gekommen.	
CB-DB-1063	Der Vorgang wurde nach XXX Durchläufen abgebrochen.
Der Fehler tritt auf, wenn die Anzahl der Durchläufe den eingestellten Maximalwert (XXX) überschritten haben. In diesem Fall sollte man entweder die Auswahl einschränken oder die erlaubte Anzahl erhöhen (siehe Beispiel 2.43, „Erhöhen der maximalen Anzahl der Durchläufe“).	

6.3. Checkerberry web

Dieses Kapitel listet eine Reihe von häufig gestellten Fragen und deren Antworten auf.

6.3.1. Wie kann ich checkerberry web in Hudson integrieren?

Checkerberry web benötigt für die Ausführung der Tests einen Browser und somit auch eine grafische Oberfläche. Diese Vorbedingung ist bei einer Hudson-Installation auf Windows-Systemen automatisch gegeben. Bei der Installation auf Linux-Systemen sieht dies anders aus. Für dieses Problem gibt es unterschiedliche Lösungen. Eine Möglichkeit besteht in der Registrierung verschiedener Windows-Rechner als Hudson-Slaves, die die Tests durchführen. Dadurch wird zusätzlich die Last verteilt. Des Weiteren können verschiedene Windows-Betriebssysteme verwendet werden.

Eine andere Möglichkeit besteht in der Bereitstellung eines X-Servers. Für den Betrieb auf Linux-Servern empfiehlt sich der Einsatz von `xvfb` (siehe auch [Blog alittlemadness.com, 2010]) oder `vncserver`.

Das beschriebene Vorgehen, lässt sich entsprechend auf den Jenkins übertragen.

6.3.2. Wie kann ich Portal-Anwendungen testen?

In Portal-Anwendungen ist es möglich, mehrere Portlets auf einer Webseite anzuzeigen. Da insbesondere auch das gleiche Portlet mehrfach auf der Webseite angezeigt werden kann, muss die Portal-Anwendung die Eindeutigkeit der HTML-IDs sicherstellen. Zu diesem Zweck wird jeder Portlet-Instanz in der Regel eine dynamische ID zugeordnet, die als Präfix für die HTML-IDs verwendet wird. Die Vergabe der Portlet-ID kann während des Deployments oder während des Starts des Application-Servers erfolgen. Insbesondere ist diese Portlet-ID zum Zeitpunkt der Testerstellung nicht verfügbar. Wie kann man in diesem Umfeld GUI-Tests programmieren?

Checkerberry web bietet den großen Vorteil des Modellansatzes, über den die erforderliche Implementierung realisiert werden kann. Die Lösung ist dabei von der konkreten Anwendung und Portal-Implementierung abhängig. Dennoch lässt sich ein allgemeiner Lösungsweg beschreiben.

Die wesentliche Aufgabe bei dem Test von Portal-Anwendung besteht darin, die Zuordnung zwischen Portlet und Portlet-ID herzustellen. Zu diesem Zweck bietet checkerberry web die abstrakte Klasse `AbstractRemoteControlPortlet`. Für jedes Portlet der Anwendung wird eine eigene Portlet-Klasse implementiert z.B. `LoginPortlet`. In dieser Klasse implementiert der Entwickler Getter-Methode für alle Webseiten, die zu diesem Portlet gehören z.B. `public LoginPage getLoginPage()`. Die abstrakte Portlet-Klasse verfügt über das Property `portletId`. Dieses Property kann über die enthaltenen Page-Modellklassen an die Komponenten weitergereicht werden.

Das folgende Beispiel zeigt eine Implementierung der `LoginPage` im Portal-Umfeld. Im Gegensatz zu dem vorherigen Beispiel enthält die `LoginPage`-Klasse nun eine Referenz auf das Portlet, in dem die Seite angezeigt wird. Über die Portlet-ID kann dann die HTML-ID der enthaltenen Komponenten dynamisch erzeugt werden.

```
public class LoginPage extends AbstractRemoteControlPage {

  private AbstractRemoteControlPortlet portlet;

  // Konstruktor mit Contextprovider und Portlet aufrufen
  public LoginPage(ContextProvider contextProvider, AbstractRemoteControlPortlet portlet) {
      super(contextProvider);
      this.portlet = portlet;
  }

  public RemoteControlComponent getUserTextField() {
    // Portlet Präfix bei der HTML-ID berücksichtigen.
    return createComponentProxy(portlet.getPortletId() + ":userId");
  }
}
```

Beispiel 6.5. Login-Page-Modell im Portal-Umfeld

Da das LoginPage-Modell über das Portlet erzeugt wird, kann die Referenz auf das Portlet leicht übergeben werden. Das folgende Beispiel zeigt eine mögliche Implementierung.

```
public class LoginPortlet extends AbstractRemoteControlPortlet {

  // Konstruktor mit Fernsteuerung und ContextProvider aufrufen
  public LoginPortlet(ContextProvider contextProvider) {
    super(contextProvider);
  }

  public LoginPage getLoginPage() {
    // LoginPage mit Referenz auf dieses Portlet aufrufen. (Das
    // Portlet ist auch ein ContextProvider...)
    return new LoginPage(this);
  }
}
```

Beispiel 6.6. Beispiel-Implementierung Login-Portlet-Modell

Das Portlet-Modell wird mit dem aktuellen ContextProvider erzeugt. Über Getter-Methoden werden die Modelle der Webseiten zur Verfügung gestellt, die innerhalb des Portlets verwendet werden. Bei der Erzeugung dieser Modelle wird die Referenz auf das entsprechende Portlet einfach übergeben.

Das vorgestellte Verfahren bietet eine gute Möglichkeit, Modelle in Portal-Umgebungen zu definieren. Die entscheidende Frage ist bisher jedoch unbeantwortet geblieben: Wie kann die Portlet-ID für ein Portlet ermittelt werden?

Eine Möglichkeit für die Ermittlung der Zuordnung von Portlet und Portlet-ID besteht in der Erweiterung der HTML-Datei. Über ein zusätzliches oder bestehendes HTML-Tag kann die Information in die Webseite eingebunden werden. Es ist z.B. möglich, ein `DIV`-Element um jedes Portlet

zu definieren, das einen Portlet-Namen und die Portlet-ID in der HTML-ID enthält z.B. `<DIV ID="PORTLET:NAME=Login;ID=XYZ4711">`. Der vollständige Inhalt der HTML-Seite kann über die Methode `RemoteControl.getHtmlSource()` ermittelt werden, sodass die Zuordnung ausgelesen und interpretiert werden kann.

Es ist abhängig von der Portal-Anwendung, wann die Zuordnung von Portlet und Portlet-ID ausgelesen und verarbeitet wird. Aus Performance-Gründen kann es sinnvoll sein, die HTML-Seite einmal einzulesen und in die ermittelten Zuordnungen in der Klasse `WebContext` zu speichern. In anderen Portal-Anwendungen kann es sinnvoller sein, die Zuordnung direkt bei der Erzeugung des Portal-Modells vorzunehmen.

6.3.3. Warum erscheint bei clickAndWaitForPage ein Timeout?

Warum tritt bei dem Aufruf von `clickAndWaitForPage` immer ein Timeout auf, obwohl die Seite angezeigt wird? Die Methode `clickAndWaitForPage` wird verwendet, wenn nach dem Klicken einer Komponente eine komplette Seite neu geladen wird. Der Browser versendet einen HTTP-Request und empfängt eine HTTP-Response. Nachdem die Seite vollständig geladen wurde, wird der Aufruf der Methode `clickAndWaitForPage` beendet. Die Methode erkennt jedoch nicht, wenn die Seite durch einen Ajax-Request geladen wurde. Dynamische Änderungen des DOM-Baums durch JavaScript werden ebenfalls nicht als Neuladen der Seite erkannt. In diesen Situationen muss die Methode `clickAndWait` verwendet werden.

Alternativ kann auch explizit auf die Existenz konkreter Felder gewartet werden z.B. `userField.waitUntilAvailable()`. Dieses Vorgehen hat zum einen den Vorteil, dass die Tests schneller durchlaufen, da keine unnötigen Wartezeiten verwendet werden. Zum anderen sind die Tests auch robuster. Dies liegt daran, dass die Verwendung von Timeouts auf unterschiedlichen Plattformen zu Problemen führt. Die Timeouts müssen so klein wie möglich sein, um die Testlaufzeiten nicht zu stark zu verzögern. Auf der anderen Seite müssen die Timeouts groß genug sein, damit die Tests auf allen Umgebungen korrekt ausgeführt werden. Es kommt daher häufig vor, dass die Timeouts zu gering gewählt werden und Tests fehlschlagen, obwohl die Ausführung langsamer ist. Das passiert bei dem expliziten Warten auf Komponenten nicht.

6.3.4. Warum laufen die Tests im Internet Explorer nicht?

Wenn GUI-Tests im Firefox Browser erfolgreich verlaufen und im Internet Explorer nicht, kann das verschiedene Ursachen haben. In der Regel hat Selenium beim Internet Explorer Schwierigkeiten mit Funktionen, die das native Key-Handling betreffen. Wenn also der Aufruf `typeKeys()` an einer Komponente des Page Models fehlschlägt, ersetzen Sie ihn durch `type()`. Selenium arbeitet mit JavaScript (siehe Abschnitt 3.2, „Funktionsweise"). Der Internet Explorer blockiert diese in der Standard Einstellung. Wenn Sie den geschützen Modus deaktivieren, kann Selenium per JavaScript auf die Application under Test zugreifen. Alternativ ist es auch ausreichend, die zu testende URL als vertrauenswürdige Seite in den Einstellungen des Internet Explorers aufzunehmen.

6.3.5. Welche Fehlermeldungen gibt es in checkerberry web?

Tabelle 6.3. Fehlermeldungen in checkerberry web

CB-WEB-1000	Die Durchführung der Aktion ist nicht möglich, da der Bildschirmschoner aktiv ist.
Diese Fehlermeldung erscheint, wenn eine Aktion durchgeführt werden soll, die bei aktivem Bildschirmschoner nicht ausgeführt werden kann. Dies ist z. B. der Fall, wenn ein nativer Tastendruck ausgeführt werden soll. Bei einem aktiven Bildschirmschoner erreicht der Tastendruck nie die zu testende Webseite, sodass diese Fehlermeldung die Ursache für den fehlschlagenden Test angibt.	

CB-WEB-1001	Die Testklasse 'XXX' ist unbekannt.
Diese Fehlermeldung erscheint, wenn bei der Erzeugung der checkerberry web-Umgebung auf einen Testklassenname `XXX` zugegriffen wird, der unbekannt ist.	
CB-WEB-1002	Die Testmethode 'XXX' der Testklasse 'YYY' ist unbekannt.
Diese Fehlermeldung erscheint, wenn auf eine Testmethode `YYY` der Testklasse `XXX` zugegriffen wird und die Testklasse diese Methode nicht enthält oder der Zugriff auf diese Methode nicht möglich ist.	
CB-WEB-1004	Selenium hat eine Exception geworfen.
Kapselt eine Exception des Selenium Frameworks.	
CB-WEB-1007	Das checkerberry web-Environment ist bereits aktiv. Bitte prüfen Sie, ob die `setUp`-Methode zweimal aufgerufen wurde oder ein `tearDown`-Aufruf fehlt.
Ein checkerberry web-Environment muss vor jeder Testmethode über den Aufruf der Methode `setUp` initialisiert werden und nach jeder Testmethode über den Aufruf `tearDown` runtergefahren werden. Anderenfalls erscheint die angegebene Fehlermeldung.	
CB-WEB-1008	Für den Browsertyp XXX wurde kein WebDriver-Creator registriert.
Für die Verwendung von WebDriver in checkerberry web ist es erforderlich, dass ein Creator zur Erzeugung der entsprechenden WebDriver-Instanz konfiguriert ist. Für Firefox, Internet Explorer und Google Chrome sind die entsprechenden Creator-Instanzen schon registriert. Wenn ein Browser mit WebDriver verwendet werden soll, der über keine Creator-Instanz verfügt, wird diese Fehlermeldung ausgegeben.	

6.3.6. Warum Selenium RC als Default-Fernsteuerung?

Bei der Verwendung von Selenium gibt es zwei Möglichkeiten, die Kommunikation (Fernsteuerung) mit dem Browser umzusetzen: Selenium RC (Remote Control) und WebDriver. Während Selenium RC die ursprüngliche Implementierung von Selenium ist, wurde WebDriver über Google in das Projekt eingebracht. Es wird vermutet, dass WebDriver die einzige zukünftige Lösung sein wird. Aktuell ist es jedoch so, dass Selenium RC und WebDriver sich sehr gut ergänzen. Dies sieht man gerade auch bei der Browser-Unterstützung (siehe [Selenium Platforms, 2012]). Da Selenium RC weitere Vorteile hat (mehrere Browser-Instanzen pro Test, umfangreichere API, Unterstützung älterer Browser) wurde die Umstellung der internen Fernsteuerung nicht vorgenommen.

6.3.7. Kann ich mein Modelle für Selenium RC und WebDriver verwenden?

Ja. Theoretisch ist es problemlos möglich, die verwendeten Modellklassen sowohl mit Selenium RC und WebDriver zu verwenden. In der Praxis ist es jedoch so, dass WebDriver eine eingeschränktere API als Selenium RC hat. Aus diesem Grund kann es passieren, dass Funktionen verwendet werden, die in Selenium RC zu dem gewünschten Ergebnis führen, während sie unter WebDriver nicht oder anders funktionieren. In der Konfiguration von checkerberry web kann eingestellt werden, dass Selenium RC und WebDriver sich möglichst gleich verhalten sollen (Kompatibilitätsmodus), damit das Umschalten der Fernsteuerung möglichst einfach ist. Per Default ist der Kompatibilitätsmodus aktiviert. Wenn das ursprüngliche Verhalten

der ausgewählten Implementierung gewünscht ist, muss dieser Kompatibilitätsmodus explizit deaktiviert werden.

Kapitel 7. Best Practices

7.1. Checkerberry test center

7.1.1. Namenskonvention

Bei der Verwendung des checkerberry test centers hat sich gezeigt, dass die Einhaltung einer Namenskonvention sinnvoll ist. Die Namenskonvention dient zur Markierung und Gruppierung verschiedener Tests. Dies ist vor allem dann sinnvoll, wenn die Tests zu unterschiedlichen Zeitpunkten ausgeführt werden sollen. Dies kann aus unterschiedlichen Gründen sinnvoll sein. Zum einen benötigen die Tests des checkerberry test centers bestimmte Voraussetzungen (Datenbank oder laufende Web-Anwendung), die nicht zu jedem Zeitpunkt erfüllt sind. Zum anderen kann die Laufzeit der Tests zu lang sein, sodass diese Tests z.B. nur nachts ausgeführt werden.

Für die Tests mit checkerberry db wird die Namenskonvention `XXXIntegrationTest` verwendet, während die Tests mit checkerberry web der Namenskonvention `XXXGuiTest` folgen.

7.2. Checkerberry db

Universelle Werkzeuge zeichnen sich dadurch aus, dass sie hilfreich in vielen Umgebungen eingesetzt werden können. Als Nachteil dieser Flexibilität stellt sich dann die Frage: Wie setze ich das Werkzeug in der aktuellen Umgebung am besten ein? Es gibt eine Reihe von Empfehlungen, die sich aus den Erfahrungen der letzten Jahre ergeben haben. Ein Teil dieser Empfehlungen kommt bereits aus dem DbUnit-Projekt [DbUnit Best Practices, 2010], wobei wir die Gelegenheit nutzen, sie hier noch einmal aufzugreifen und zu erläutern.

7.2.1. Good setup don't need cleanup

Fangen wir also gleich mit dem Klassiker an: Good setup don’t need cleanup [DbUnit Best Practices, 2010]. Hinter dieser griffigen Phrase verbirgt sich die Idee, dass Tests nach ihrer Ausführung nicht für das Aufräumen verantwortlich sind. Stattdessen ist jeder Test in der Setup-Phase dafür verantwortlich, alle benötigen Rahmenbedingungen herzustellen. In der Setup-Phase wird somit implizit für den zuvor ausgeführten Test aufgeräumt.

Das klingt erst einmal sinnvoll und logisch. Allerdings sollte man etwas tiefer einsteigen, um die verschiedenen Facetten dieser Regel zu verstehen.

Innerhalb von Unit-Tests wird die Setup-Phase genutzt, um die Rahmenbedingungen des Tests herzustellen. Danach wird der Test durchgeführt, bevor in der Teardown-Phase alle Ressourcen wieder freigegeben werden. Die Grundidee der Teardown-Phase bestand darin, dass sich das System nach dem Aufräumen in dem gleichen Zustand wie vor der Setup-Phase befinden sollte.

In der Praxis stellten sich dann jedoch einige Entwickler die Frage: Warum Aufräumen, wenn jeder Test ohnehin sein benötigtes Umfeld in der Setup-Phase wieder herstellt?

Es gibt viele Situationen, in denen ein Test Aufräumarbeiten leisten muss. Jeder Test muss nach seiner Ausführung dafür sorgen, dass die im Test erzeugten Objekte wieder freigegeben werden. Dies ist insbesondere im JUnit-Umfeld wichtig, da anderenfalls der Speicher unnötig verschwendet wird. Bei großen JUnit-Test-Suites kommt es in diesen Situationen regelmäßig zu `OutOfMemory`-Fehlern. Des Weiteren sollte jeder Test Änderungen rückgängig machen, die nur diesen Test betreffen. Ersetzt ein Test beispielsweise einen speziellen Service in der allgemeinen Umgebung durch eine eigene Implementierung, so muss dies nach der Testdurchführung wieder korrigiert werden. Es ist nicht pragmatisch, wenn hunderte Tests einen Service initialisieren müssen, nur weil ein einzelner Test eine andere Ausprägung des Services benötigt.

Stellen Sie sich den Aufwand vor, wenn alle Tests die Besonderheiten einzelner Tests berücksichtigen müssten.

Auf der anderen Seite ist das Aufräumen der Datenbank nach der Testdurchführung überflüssig. Wenn ein Test die Datenbank benötigt, muss er in der Setup-Phase den gewünschten Zustand herstellen. Es wäre daher sinnlos, in der Teardown-Phase die Datenbank erst in den einen und in der Setup-Phase dann in einen anderen Zustand zu versetzen. Das kostet unnötig Zeit. Des Weiteren vereinfacht das unaufgeräumte Hinterlassen der Datenbank die Fehleranalyse. Führt man einen einzelnen fehlerhaften Test aus, so bleibt nach der Testdurchführung der letzte Zustand der Datenbank erhalten und kann somit direkt für die Analyse verwendet werden.

7.2.2. Mindestens ein Datenbankschema pro Entwickler

Die Empfehlung, dass jeder Entwickler ein eigenes Datenbankschema zur Verfügung haben sollte, kommt ebenfalls aus dem DbUnit-Projekt [DbUnit Best Practices, 2010]. Die Idee dahinter ist klar: Für die Durchführung von integrativen Datenbanktests kann nur ein Test zurzeit laufen. Wenn sich mehrere Entwickler ein Datenbankschema teilen, erfordert das einiges an Abstimmungsbedarf und führt im schlimmsten Fall zu Wartezeiten. Das erinnert ein wenig an alte Zeiten, in denen ein zwei Stunden Zeitfenster ausreichen musste, um einen Stapel Lochkarten auszuführen und die auftretenden Fehler zu entdecken. Diese Situation passt nicht in die heutige Entwicklerwelt, in der dem Team alle Beschränkungen aus dem Weg geräumt werden sollen.

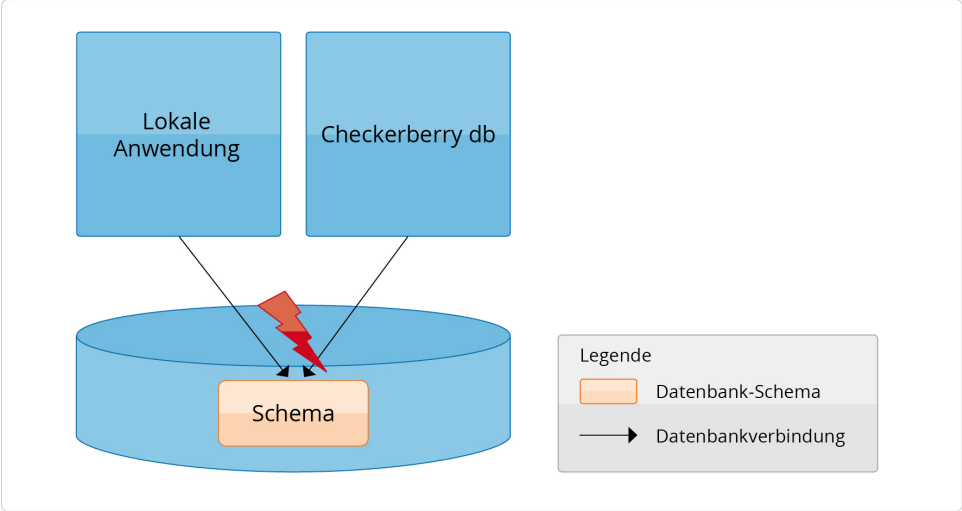


Abbildung 7.1. Eine Datenbank pro Entwickler

Abbildung 7.1, „Eine Datenbank pro Entwickler" skizziert das angesprochene Problem. Der Software-Entwickler verwendet für die lokale Anwendung und für die checkerberry db-Tests das gleiche Datenbankschema. Zum einen ergibt sich das Problem, dass die lokale Anwendung und die Tests nicht parallel ausgeführt werden können. Zum anderen löscht checkerberry db regelmäßig die Daten in dem Schema, sodass sich der Entwickler keine Testdaten für die lokale Anwendung aufbauen kann.

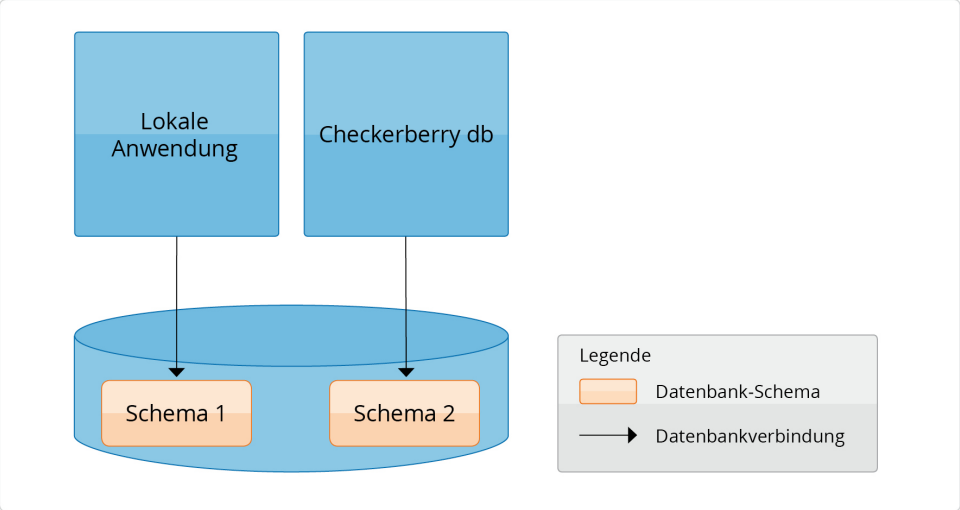


Abbildung 7.2. Zwei Datenbanken pro Entwickler

Abbildung 7.2, „Zwei Datenbanken pro Entwickler“ zeigt die Lösung des Problems. Sowohl die lokale Anwendung als auch die checkerberry db-Tests verwenden ein eigenes Schema und können so unabhängig voneinander ausgeführt werden.

Aus unserer Sicht geht die Forderung ein Datenbankschema pro Entwickler bereitzustellen in die richtige Richtung, wobei es konkreter „mindestens ein Schema“ heißen müsste. In den meisten Projekten haben Entwickler lokale Installationen der Kundenanwendungen. Diese Anwendungen werden zum Entwickeln, Testen oder zur Fehleranalyse verwendet. Die Anwendungen benötigen häufig umfangreiche Daten, die sequentiell durch den Entwickler aufgebaut werden. Es ist nicht sehr förderlich, wenn diese Datenbestände jedes Mal gelöscht werden, wenn lokal integrative Datenbanktests durchgeführt werden. Natürlich kann der Entwickler die Daten sichern und neu einspielen. Dieser Schritt ist jedoch fehleranfällig, insbesondere da er einfach vergessen werden kann. Die Korrektur dieses Datenverlustes kostet Zeit und Nerven. Durch die Einrichtung von mehreren Datenbankschemata pro Entwickler kann man diesen Problemen aus dem Weg gehen.

Die Forderung eines eigenen Datenbankschemas für jeden Entwickler ist effizient nur durch eine entsprechende Infrastruktur zu gewährleisten. Die Installation eines neuen Datenbankschemas muss so einfach wie möglich sein, damit neue Entwickler schnell ein eigenes Schema bekommen können und damit Änderungen der Datenbankstruktur schnell für alle Entwickler verfügbar sind. In vielen Umgebungen, wie z.B. Maven, kann die Anlage oder Änderung eines Datenbankschemas durch die Ausführung eines parametrisierbaren Skripts erfolgen.

Wenn diese Infrastruktur aufgebaut ist, macht es keinen großen Unterschied, ob jeder Entwickler ein oder zwei Datenbankschemata zur Verfügung hat. Es bleibt abhängig von den fachlichen Anforderungen, wie viele Datenbankschemata für die effiziente Software-Entwicklung erforderlich sind. Es sollte jedoch darauf geachtet werden, dass einheitliche Namenskonventionen verwendet werden z.B. "USR4711" und "USR4711_TEST" o.ä.

7.2.3. Vermeide das Testdaten-Labyrinth

Checkerberry db bietet die Möglichkeit allgemeine Testdaten in spezielle Testdaten einzubinden. Diese Möglichkeit erinnert an das DRY-Prinzip. Die Abkürzung DRY steht für „Don’t repeat yourself“ und beschreibt ein Prinzip aus der Software-Entwicklung, dass die Duplizierung von Source-Code untersagt. Der Source-Code ist so zu strukturieren, dass die Duplizierung von Code-Fragmenten nicht erforderlich ist. Diese Forderung ist ein wichtiger Baustein für die Entwicklung von wartbarem Source-Code.

Checkerberry db bietet technisch die Möglichkeit das DRY-Prinzip auch in den Testdaten umzusetzen. Davon sollte man jedoch maßvoll Gebrauch machen.

Java-Source-Code unterliegt klaren Regeln und lässt sich gut strukturieren. Die meisten Entwicklungsumgebungen bieten viele Möglichkeiten, den Source-Code zu analysieren, strukturiert darzustellen und unterstützen den Benutzer bei der Navigation zwischen den verschiedenen Klassen. Insbesondere lassen sich Referenzen der Klassen untereinander schnell ermitteln.

Bei der Bearbeitung von XML-Testdaten ist diese Tool-Unterstützung nicht vorhanden. Selbst die einfache Frage, in welchen Testdaten bestimmte Include-Dateien verwendet werden, ist nicht so leicht zu beantworten. Über eine einfache Suche findet man nur die Testdaten, die die Include-Datei direkt verwenden. Indirekte Verwendungen (Includes von Includes) findet man so nicht.

Das größere Problem stellt jedoch nicht die mangelnde Tool-Unterstützung dar, sondern der hohe Wiederverwendungsgrad. In den Testdaten gibt es immer ausgezeichnete Objekte, die in sehr vielen Tests verwendet werden können. Bei konsequenter Anwendung des DRY-Prinzips würden diese Daten ausgelagert und von vielen Tests referenziert werden. Die folgende Abbildung stellt das Problem dar.

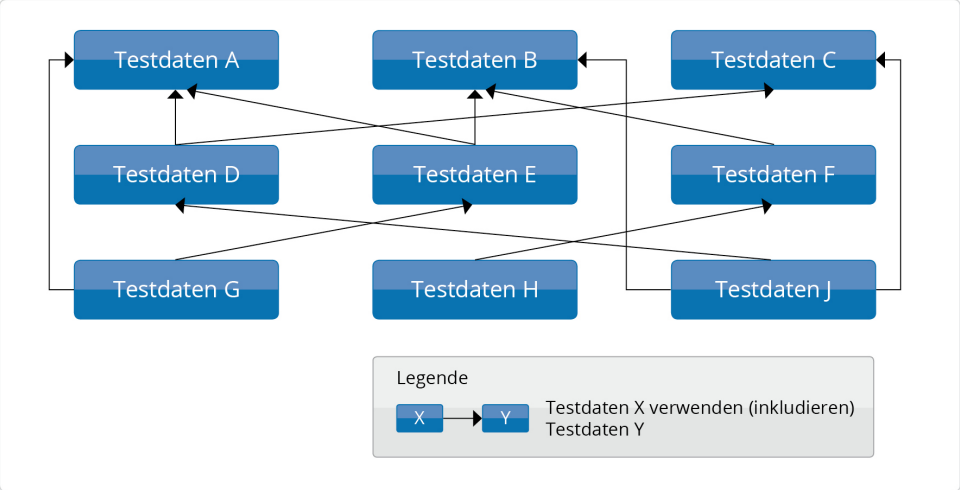


Abbildung 7.3. Testdaten-Labyrinth

Die Abbildung zeigt mögliche Abhängigkeiten von Testdaten untereiander. Die Referenzen der Testdaten untereinander werden unüberschaubar. Dies führt zum einen dazu, dass auf den ersten Blick nicht ersichtlich ist, welche Testdaten für einen Test verwendet werden. Betrachtet man zum Beispiel die „Testdaten G“ aus der Abbildung, so werden dort die „Testdaten E“, „Testdaten A“ und „Testdaten B“ verwendet. Das Beispiel aus der Abbildung ist jedoch sehr klein. Bei einer konsequenten Umsetzung des DRY-Prinzips in einer realen Anwendung kann die Anzahl der inkludierten Testdaten für eine einzelne Testdatendatei schnell den zweistelligen Bereich erreichen.

Zum anderen ist die Wartbarkeit der Testdaten stark eingeschränkt, da der Entwickler nicht überblicken kann, was für Konsequenzen die Änderung von Testdaten hat. In der Abbildung werden die „Testdaten A“ von den „Testdaten G“ und „Testdaten J“ verwendet. Selbst in dieser einfachen Situation müssen bei einer Änderung der „Testdaten A“ die Auswirkungen auf die anderen Tests überprüft werden. Bei einer konsequenten Umsetzung des DRY-Prinzips würde die Wiederverwendung der Testdaten erheblich zunehmen. In dieser Situation können zentrale Testdaten dann im zwei- bis dreistelligen Bereich referenziert werden. Es könnte somit sein, dass die „Testdaten A“ nicht zweimal sondern 200 Mal referenziert werden. Das verhindert eine Änderung an den „Testdaten A“, da die Auswirkungen der Änderungen nicht überschaubar sind. Das Entstehen eines solchen Testdaten-Labyrinths sollte unbedingt vermieden werden.

Bei der Erstellung von Testdaten ist das DRY-Prinzip nicht geeignet. Es gibt eine Reihe von Daten, bei denen eine Auslagerung in separate Testdatendateien sinnvoll ist. Dies umfasst Stamm- oder Konfigurationsdaten sowie generelle fachliche Daten. Im Zweifel ist es besser Testdaten für den Preis der Wartbarkeit zu duplizieren.

7.2.4. Tests müssen unabhängig voneinander sein

Mit dieser Empfehlung streifen wir wieder ein Thema, das kontrovers diskutiert wird: Müssen Tests unabhängig voneinander sein?

Was bedeutet „Unabhängigkeit" an dieser Stelle? Es gibt Stimmen, die propagieren, dass Tests auf den Ergebnissen anderer Tests aufsetzen sollen bzw. dürfen. Die Unabhängigkeit ist an dieser Stelle nicht gegeben, da der nachfolgende Test ohne den vorherigen nicht korrekt ausgeführt werden kann. Wenn der erste Test also fehlerhaft ist, wird der zweite Test auch fehlschlagen. Änderungen an dem ersten Test erfordern Änderungen an dem zweiten Test, wobei diese Abhängigkeit maximal im Javadoc sichtbar ist. Verwendet der Test ggf. weitere Bedingungen von anderen Tests und wie erkennt der Entwickler diese Abhängigkeiten?

Durch diese Abhängigkeiten sind nicht mehr alle Tests einzeln ausführbar. Tritt ein Fehler in einem Regressionstest auf, muss der Entwickler herausfinden, auf welche Tests der fehlschlagende Test aufsetzt, damit das Problem lokal reproduziert werden kann. Im schlimmsten Fall setzt sich die Abhängigkeit bis zum ersten Test fort. Dann wäre die gesamte Test-Suite nur als Ganzes ausführbar.

Die Intransparenz der Abhängigkeiten führt langfristig zu einem sehr aufwändigen Prozess zur Testerstellung und –pflege. Dieser Preis ist zu hoch für die Zeit- und Ressourcen-Einsparungen, die durch abhängige Tests ermöglicht werden. Daher sollten Tests immer unabhängig voneinander sein.

7.2.5. Testdaten so groß wie nötig, so klein wie möglich

Bei der Erstellung der Testdaten sollte man stets darauf achten, minimale Testdaten zu verwenden, d.h. die Testdaten sollten nur Einträge enthalten, die für den Test wirklich relevant sind. Dies hilft die Testdaten übersichtlich zu halten. Anderenfalls kommt es häufig dazu, dass man den Überblick verliert. Je mehr Einträge in den Testdaten vorhanden sind, desto schwieriger ist es herauszufinden, welche davon tatsächlich benötigt werden. Insbesondere bei der Fehleranalyse ist dies problematisch. Wenn während eines Regressionstests ein Test fehlschlägt, muss die Ursache für den Fehler gefunden werden. Der Fehler kann in der fachlichen Logik, im Test, in den Testdaten oder in einer beliebigen Kombination der vorherigen Punkte liegen. In diesem Fall ist es einfacher, eine XML-Datei mit 30 Zeilen als eine XML-Datei mit 800 Zeilen zu analysieren.

Ein weiterer Aspekt ist die Performance bei integrativen Datenbanktests. Je mehr Testdaten einzuspielen sind, desto länger dauern die Tests und desto später bekommt der Entwickler ein Feedback.

Bei der manuellen Erstellung der Testdaten ist diese Empfehlung meist erfüllt. Wenn die Testdaten jedoch durch einen Datenbank-Dump erstellt wurden, enthalten sie meist mehr Informationen als nötig. In diesen Situationen empfehlen wir, die irrelevanten Testdaten zu löschen.

7.2.6. Lagere Caching-Daten in Includes aus

Checkerberry db unterstützt das Caching von Tabellen. Checkerberry db merkt sich dabei lediglich, ob eine cacheable Tabelle bereits durch einen Test eingespielt wurde. Ist dies der Fall, so wird die Tabelle nicht erneut eingespielt. Insbesondere kommt es nicht zu einem Vergleich der Inhalte dieser Tabelle.

Dieses Verfahren geht von der Annahme aus, dass alle Tests, die den Cache nutzen, die gleichen Werte in den cacheable Tabellen erwarten. Aus diesem Grund sollten die Daten der cacheable Tabellen in eigene Testdaten ausgelagert werden. Die Daten können dann in die entsprechenden Testdaten eingebunden

werden, sodass jeder Test tatsächlich die gleichen Daten der cacheable Tabellen verwendet. Auf diese Art und Weise werden potentielle Fehler vermieden.

7.2.7. Verwende fachliche Lookup-Keys

Die Lookup-Keys werden von checkerberry db verwendet, um eine Zuordnung zwischen erwartetem und tatsächlichem Dateninhalt herzustellen. Sie sind für checkerberry db somit das, was die Primary-Keys für die Datenbank sind. Generell können die Primary-Keys auch als Lookup-Keys verwendet werden. Wir empfehlen jedoch, sofern möglich, fachliche Schlüssel als Lookup-Keys zu verwenden.

Fachliche Schlüssel haben den Vorteil, dass sie greifbarer und leichter verständlich sind. Im Gegensatz zu technischen Schlüsseln haben fachliche Schlüssel eine Bedeutung, die dem Entwickler den Umgang mit den Testdaten vereinfacht. Das gilt insbesondere bei der Fehleranalyse.

Des Weiteren sind technische Schlüssel in der Regel generiert, sodass diese zum Zeitpunkt der Testerstellung noch nicht feststehen. Um den Vergleich dennoch zu ermöglichen, müssten für jeden Schlüssel Parameter definiert werden, was unnötig aufwendig ist.

7.2.8. Datenbank-Sequences mit Puffer verwenden

Für die Erzeugung von Primärschlüsseln werden in der Datenbank häufig Sequences verwendet. Bei einer Sequence handelt es sich im Wesentlichen um einen Zähler, der einen Anfangswert hat und der nach jeder Abfrage um einen Offset erhöht wird. Über die Sequence können so eindeutige technische Schlüssel erzeugt werden.

Bei der Verwendung von Testdaten kann es dazu kommen, dass Konflikte in den Primärschlüsseln auftreten. Folgendes Beispiel zeigt ein mögliches Szenario: In den Testdaten wird für die Tabelle `USERS` ein Eintrag mit der ID 42 in der Datenbank angelegt. Danach wird ein neuer User in der Datenbank erzeugt. Die ID des neuen Datensatzes wird über eine Sequence erzeugt, die zufällig den Wert 42 zurückliefert. Da diese ID bereits vorhanden ist, kommt es zu einem Datenbankfehler und der Test schlägt fehl.

Eine grundlegende Eigenschaft von Tests besteht in der Robustheit, d.h. Tests müssen immer deterministische Ergebnisse liefern. Dies ist dem oben beschriebenen Beispiel nicht der Fall, da das Ergebnis des Tests abhängig von dem Wert der verwendeten Sequence ist. Das Problem lässt sich dadurch lösen, dass für die Testdaten ein eigener Wertebereich für die Primärschlüssel verwendet wird. Dies erreicht man leicht durch die Verwendung eines erhöhten Anfangswerts für die Sequences. Wenn jede Sequence nicht bei 0 sondern bei 1.000 startet, können die Werte von 0 bis 999 von den Testdaten verwendet werden, ohne dass es zu Konflikten kommt. Die Größe des Anfangswertes sollte dabei nicht zu klein gewählt werden

7.3. Checkerberry web

Dieser Abschnitt beschreibt Vorgehensweisen, die sich bei der Verwendung von checkerberry web als sinnvoll erwiesen haben.

7.3.1. Anwendungsspezifische Modellschicht

Bevor die Modellklassen für eine konkrete Anwendung erstellt werden, sollte eine weitere abstrakte Schicht eingezogen werden. Ein Ziel des Modellansatzes besteht darin, generelle Funktionalität zu kapseln und für alle Tests zur Verfügung zu stellen. Aus diesem Grund sollten für die Klassen `RemoteControlComponent`, `AbstractRemoteControlPage` und `AbstractRemoteControlPortlet` eigene abstrakte Klassen definiert werden, von denen alle konkreten Ausprägungen erben z.B. `WikipediaPage` oder `LeoPortlet`.

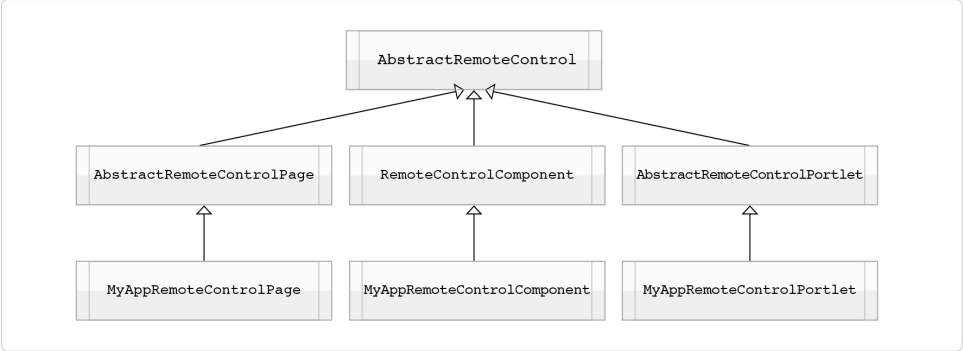


Abbildung 7.4. UML-Klassenhierarchie der Modellklassen

Die Abbildung zeigt eine exemplarische Klassenhierarchie für eine anwendungsspezifische Modellschicht. In den abstrakten Klassen `MyAppRemoteControlPage`, `MyAppRemoteControlComponent` und `MyAppRemoteControlPortlet` können allgemeine Funktionalitäten der entsprechenden Anwendung gekapselt werden.

Die Methode `AbstractRemoteControlPage.createComponentProxy` sollte dann wie folgt überschrieben werden:

```
protected MyAppRemoteControlComponent createComponentProxy(
    String elementLocator) {
  return new MyAppRemoteControlComponent(this, elementLocator);
}
```

Beispiel 7.1. AbstractRemoteControlPage createComponentProxy

7.3.2. Bevorzugter Locator: HTML-ID

Selenium unterstützt eine Reihe von Locatoren um HTML-Komponenten zu identifizieren. Die Lesbarkeit der Locatoren ist dabei nicht zwingend erforderlich, da sie innerhalb der Modellklassen gekapselt sind. Dennoch empfehlen wir HTML-IDs als Locatoren zu verwenden. Dies hat im Wesentlichen zwei Gründe. Zum einen sind HTML-IDs robuster im Hinblick auf Änderungen an der Web-Anwendung. Ein neues Layout der Webseite hat keine Auswirkung auf die HTML-IDs, sodass keine Anpassungen an den Modellklassen erforderlich sind. Bei der Verwendung von XPath-Ausdrücken als Locatoren wäre die Änderung massiv. Zum anderen bietet die Verwendung von HTML-IDs in einigen Browsern Performance-Vorteile. In allen Versionen des Internet Explorers sind die vorhandene XPath-Engines sehr langsam, sodass die Testausführung bei intensiver Verwendung von XPath-Locatoren deutlich länger dauert. Hinzu kommt, dass bei der Verwendung von WebDriver als Fernsteuerung, im Vergleich zu Selenium RC nur eine Teilmenge der Locatoren verwendet werden kann. Die Verwendung von HTML-IDs unterstützen beide Ansätze, sodass auch die Wiederverwendbarkeit der Modellklassen für Selenium RC und WebDriver gegeben ist.

Diese Empfehlung bedeutet natürlich auch, dass bei der Entwicklung der Webseiten entsprechende HTML-IDs zu vergeben sind. Gerade bei bestehenden Anwendungen kann die nachträgliche Einführung von HTML-IDs jedoch sehr aufwändig sein.

7.3.3. Ermittlung der Locatoren

Die Kernaufgabe bei der Erstellung der Modellklassen besteht in der Ermittlung der Locatoren der einzelnen Komponenten. Bei einfachen Webseiten können die Locatoren durch die Anzeige des Seitenquelltextes der HTML-Seite bestimmt werden. Es geht jedoch noch einfacher.

Die bereits vorgestellte Selenium IDE [Selenium Download, 2010] kann als Firefox-Plug-in installiert werden und zeichnet die Interaktion des Benutzers mit einer Webseite auf. Diese Aufzeichnung beinhaltet auch den

Locator der entsprechenden Komponenten, sodass diese direkt abgelesen werden können. Dieses Vorgehen ist in den meisten Fällen schnell und ausreichend.

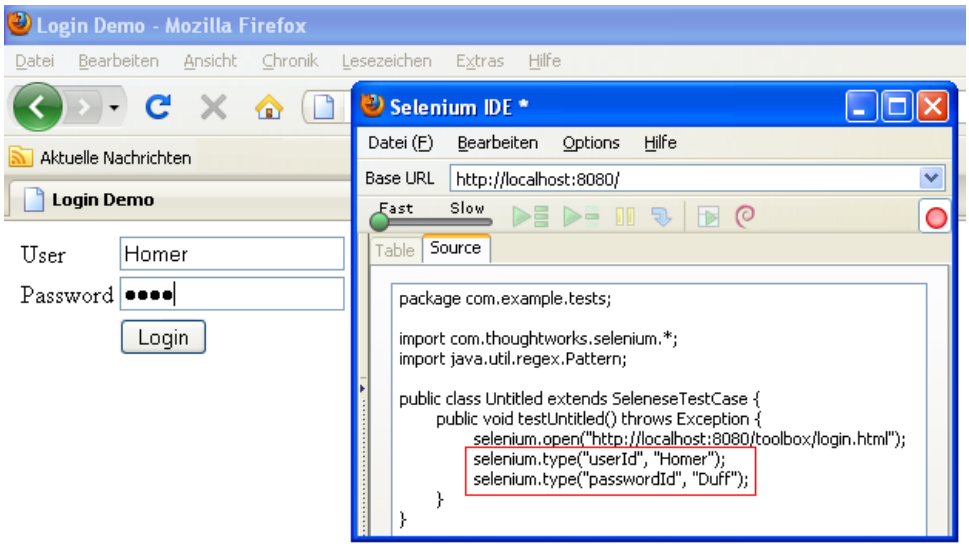


Abbildung 7.5. Ermittlung der Locatoren mit der Selenium IDE

Abbildung 7.5, „Ermittlung der Locatoren mit der Selenium IDE“zeigt ein Beispiel für die Aufzeichnung der Locatoren durch die Selenium IDE. Die zu testende Webseite enthält ein Eingabefeld für den Usernamen, ein Eingabefeld für das Passwort und einen Login-Button. Durch das Starten der Selenium IDE über das Firefox-Menü `Extras->Selenium IDE` werden die Benutzeraktionen aufgezeichnet. In dem dargestellten Beispiel wird der Benutzername „Homer“ und das Passwort „Duff“ eingetragen. Die Selenium IDE zeichnet diese Aktionen auf und bietet somit eine Möglichkeit die Locatoren der beiden Eingabefelder zu bestimmen. In dem Beispiel verwendet das Eingabefeld für den Usernamen den Locator `userId`, während das Eingabefeld für das Passwort den Locator `passwordId` verwendet.

In komplexeren Webseiten, in denen u.a. über JavaScript der DOM-Baum der HTML-Seite manipuliert wird, kann die Bestimmung der Locatoren komplizierter sein. In diesem Fall kann man das Firefox Plug-in DOM Inspector [DOM Inspector, 2010] verwenden. Über den DOM Inspector kann der aktuelle DOM-Baum der HTML-Seite durchlaufen werden, sodass HTML-IDs oder auch sinnvolle XPath-Locatoren ermittelt werden können.

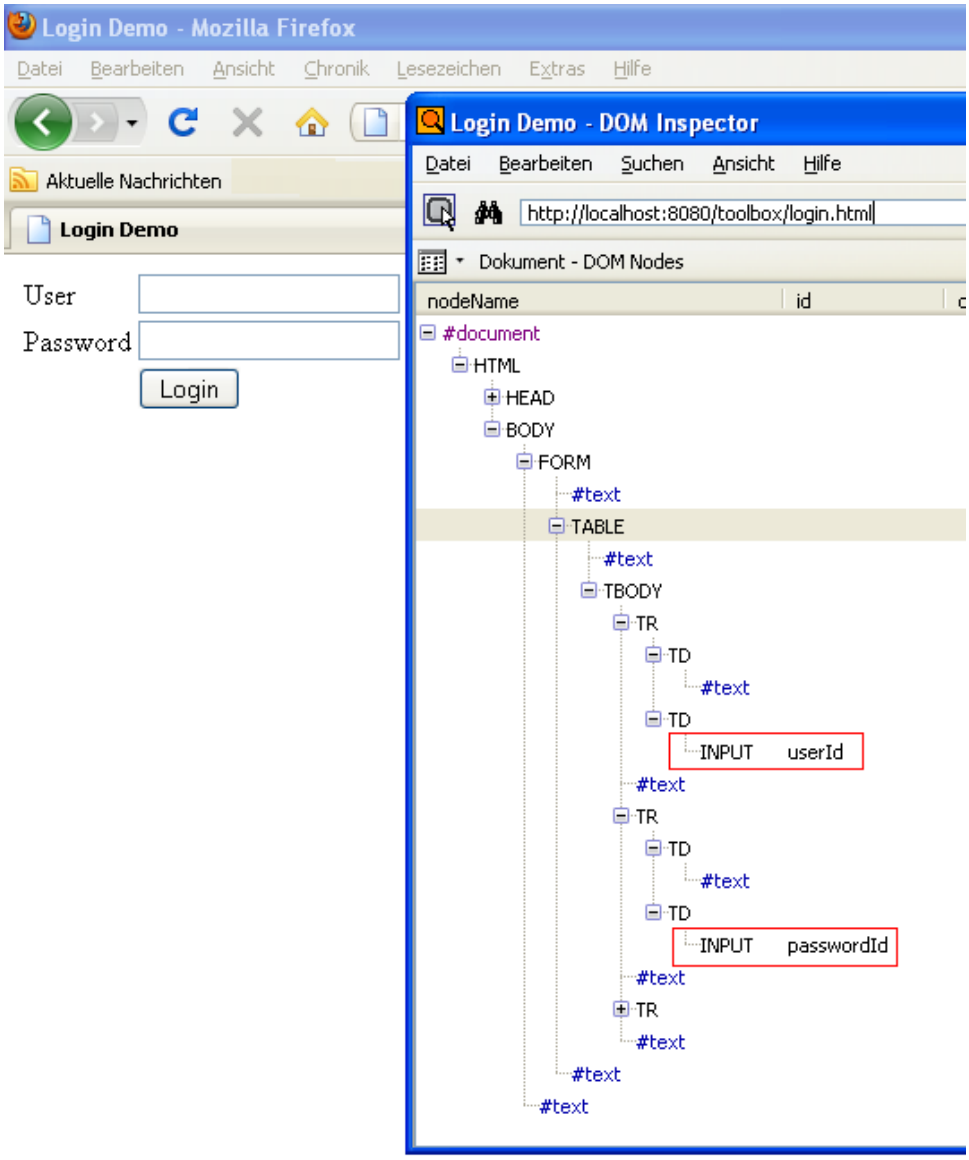


Abbildung 7.6. Ermittlung der Locatoren mit dem DOM Inspector

Abbildung 7.6, „Ermittlung der Locatoren mit dem DOM Inspector" zeigt die Verwendung des DOM Inspectors am Beispiel der Login-Seite, die bereits als Beispiel für die Selenium IDE verwendet wurde. Der DOM-Inspector wird über das Firefox-Menü `Extras->DOM Inspector` gestartet. Daraufhin wird die aktuell angezeigte Webseite analysiert und als Baumstruktur dargestellt. Die Login-Seite verfügt im BODY über ein FORM-Element, das wiederum eine Tabelle beinhaltet. In zwei Zellen der Tabelle sind die Eingabefelder für den Usernamen und das Passwort zu finden.

Die Verwendung des DOM Inspectors kann bei Webseiten mit vielen Komponenten schnell unübersichtlich werden, sodass in der Regel die Selenium IDE das einfachere Werkzeug ist, um Locatoren zu bestimmen. Die Selenium IDE stößt jedoch an Ihre Grenzen, wenn die Webseite dynamisch durch JavaScript manipuliert wird, sodass in diesen Fällen der DOM Inspector zum Einsatz kommt.

Es gibt eine Reihe weiterer Tools, die bei der Ermittlung der Lokatoren unterstützen können. Firebug ist ein weiteres Tool, das gerade im Bereich der Web-Entwicklung häufig zum Einsatz kommt [Firebug, 2011].

7.4. Checkerberry cockpit

7.4.1. Erstellung von Sicherungskopien

Bevor das checkerberry cockpit verwendet wird, sollten Sicherungskopien der zu bearbeitenden Testdaten oder Hibernate-Mappings angelegt werden. Gerade bei umfangreichen Mapping-Informationen können sich kleine Fehler wie z.B. Tippfehler einschleichen. Diese Fehler fallen nach der Konvertierung schnell auf, müssen dann allerdings behoben werden. In der Regel ist dies am einfachsten, indem man den Fehler in den Mapping-Informationen korrigiert, die gesicherten Daten wieder herstellt und die Konvertierung erneut startet.

7.4.2. Angabe von Mapping-Informationen

Das checkerberry cockpit bietet zwei Möglichkeiten, um Mapping-Informationen anzugeben. Wann sollte welche Variante verwendet werden? Die Angabe der Mapping-Informationen in Form einer Excel-Datei sollte bei umfangreichen Änderungen vorgenommen werden. Der Vorteil der Excel-Datei besteht im Wesentlichen darin, dass die Informationen gespeichert und gesichert werden können. Bei kleinen Änderungen ist jedoch die direkte Eingabe über das checkerberry cockpit schneller.

Literaturverzeichnis

[Apache Archiva Homepage, 2010] Abgerufen 2010 von Apache Archiva Homepage: http://archiva.apache.org/

[Blog alittlemadness.com, 2010] Abgerufen 2010 von Blog alittlemadness.com: http://www.alittlemadness.com/2008/03/05/running-selenium-headless/

[DbUnit Best Practices, 2010] Abgerufen 2010 von DbUnit Best Practices: http://www.dbunit.org/bestpractices.html

[DbUnit Homepage, 2010] Abgerufen 2010 von DbUnit Homepage: http://www.dbunit.org

[DbUnit-Properties, 2010] Abgerufen 2010 von DbUnit-Properties: http://www.dbunit.org/properties.html

[dnsjava, 2012] Abgerufen 2012 von dnsjava: http://www.dnsjava.org/

[DOM Inspector, 2010] Abgerufen 2010 von DOM Inspector: https://addons.mozilla.org/de/firefox/addon/6622/

[EclipseLink, 2011] Abgerufen 2011 von EclipseLink Homepage: http://www.eclipse.org/eclipselink/

[Firebug, 2011] Abgerufen 2011 von Firefox Add-on Homepage: https://addons.mozilla.org/de/firefox/addon/firebug/

[GlassFish Homepage, 2011] Abgerufen 2011 von GlassFish Homepage: http://glassfish.java.net/downloads/3.1-final.html

[GlassFish Persistence, 2011] Abgerufen 2011 von Glassfish Persistence Homepage: https://blogs.oracle.com/GlassFishPersistence/entry/use_hibernate_as_a_persistence

[GlassFish Server, 2011] Abgerufen 2011 von Oracle Glassfish Server Homepage: http://download.oracle.com/docs/cd/E19798-01/

[Google Page Objects, 2010] Abgerufen 2010 von Google Page Objects: http://code.google.com/p/selenium/wiki/PageObjects

[Hudson Homepage, 2010] Abgerufen 2010 von Hudson Homepage: http://www.hudson-ci.org/

[Java Community Homepage, JSR 318, 2011] Abgerufen 2011 von Java Community Process Homepage: http://jcp.org/en/jsr/detail?id=318

[JUnit Homepage, 2010] Abgerufen 2010 von JUnit Homepage: http://www.junit.org

[Log4j Homepage, 2010] Abgerufen 2010 von Log4j Homepage: http://logging.apache.org/log4j/

[Nexus Homepage, 2010] Abgerufen 2010 von Nexus Homepage: http://nexus.sonatype.org/

[P6Spy Dokumentation, 2010] Abgerufen 2010 von P6Spy Dokumentation: http://www.p6spy.com/documentation/other.htm

[Proxy Injection, 2010] Abgerufen 2010 von Proxy Injection: http://seleniumhq.org/docs/05_selenium_rc.html#proxy-injection

[Same Origin Policy, 2010] Abgerufen 2010 von Same Origin Policy: http://de.wikipedia.org/wiki/Same_Origin_Policy

[Selenium Download, 2010] Abgerufen 2010 von Selenium Download: http://seleniumhq.org/download/

[Selenium Homepage, 2010] Abgerufen 2010 von Selenium Homepage: http://seleniumhq.org/

[Selenium Platforms, 2012] Abgerufen 2012 von Selenium Platforms: http://seleniumhq.org/about/platforms.html

[SeleniumRC FAQ, 2010] Abgerufen 2010 von SeleniumRC FAQ: http://wiki.openqa.org/display/SRC/Selenium+RC+FAQ

[Service Locator Pattern, 2010] Abgerufen 2010 von Service Locator Pattern: http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.htm [http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html]

[slf4j Homepage, 2013] Abgerufen 2013 von slf4j Homepage: http://www.slf4j.org

[TestNG Homepage, 2010] Abgerufen 2010 von http://testng.org/doc/index.html

[TheServerSide.COM, 2011] Abgerufen 2011 von TheServerSide.COM Homepage: http://www.theserverside.com/