Testen von Datenbankcode

Testen von Softwaresystemen 2004

Kerscher Robert, 0056221, 880
Meng Johann, 0055684, 921

1 Überblick

Beim Testen von Datenbanksystemen und darauf arbeitendem Code ist es immer wichtig, dass man jederzeit den Überblick darüber behält in welchem Zustand die Datenbank ist. Selbst wenn der eigene Code absolut fehlerfrei funktioniert kann es sein, dass die Datenbank selbst Fehler aufweist (was aber bei heutigen Kaufversionen doch relativ sicher ausgeschlossen werden kann) oder dass anderer Code auf dieselbe Datenbank zugreift und sie während der Abarbeitung der eigenen Befehle in einen inkonsistenten Zustand versetzt.

Wir werden uns in dieser Arbeit aber nicht auf das Testen von Datenbanken konzentrieren, sondern gehen von Datenbanksystemen aus die für unsere Zwecke optimal und somit fehlerfrei sind.

1.1 Probleme beim Testen von Datenbankcode

Es gibt wie immer in der Softwareentwicklung eine Vielfalt von Möglichkeiten um „guten" Code von außen zu beeinflussen oder die Operationsbedingungen so zu setzen dass Code Fehler produziert.

Da bei der Anwendung von Datenbanken außer dem eigenen Code noch externe Software (die Datenbank) benutzt wird, ist es hier noch wichtiger eine einwandfreie Funktionalität der eigenen Arbeit zu garantieren, denn eines ist sicher – der Kunde findet den Fehler, auch wenn es der Tester nicht geschafft hat.

1.2 Probleme im Umgang mit Datenbanken

1. Problem: Datenmüll

   Beim Arbeiten mit Datenbanken muss immer damit gerechnet werden, dass die DB mit ungültigen oder sogar korrupten Daten gefüllt ist und somit den Ablauf des eigenen Codes entschieden beeinflussen kann. Natürlich kann auch die eigene Anwendung solchen Code in eine Datenbank schreiben.
2. Problem: Beeinflussung der Tests von außen

   Das schlimmste das beim Testen passieren kann ist, wenn sich zwei Tests gegenseitig beeinflussen.
   Beispiel: Test 1 entfernt Datensatz A aus einer Tabelle
   Test 2 basiert aber auf der Annahme dass Datensatz A in der Tabelle vorhanden ist und verändert diesen Datensatz in beliebiger Weise
   Um dieses Szenario auszuschließen muss man sich beim Design der Testfälle nur davon überzeugen ob alle Testfälle in beliebiger Reihenfolge, aber auch ob jeder Test für sich alleine ausgeführt werden kann.

3. Problem: Unbekannter Zustand der Datenbank

   Speziell wenn man nicht auf einer selbst aufgesetzten Datenbank arbeitet weiß man in der Regel nichts über den Zustand der Datenbank beim Start des Tests. Zusätzlich wünscht sich der Tester noch, dass die Datenbank nach dem Test, unabhängig von der Anzahl und Komplexität der ausgeführten Testfälle, wieder im gleichen Zustand verfügbar ist wie vor dem Test.

Zusammenfassend kann man sagen, dass es also notwendig ist, sicher sein zu können, dass die Datenbank ihren Zustand nach dem Test nicht geändert hat und auch keine ungültigen Daten in der Datenbank sind bzw. zurückbleiben.

Das lässt sich grundsätzlich recht leicht realisieren indem man nach Abschluss der Tests die genau entgegengesetzten Operationen wie im Test erneut ausführt. Es geht hauptsächlich um die Operationen INSERT, DELETE und UPDATE, denn

dabei werden unter Umständen Datensätze verändert. Bei einer reinen Suchfunktion ändert sich ja nichts an der Datenbank.

Man könnte also von Hand immer die jeweils entgegengesetzten Befehle ausführen und somit alle Anweisungen Schritt für Schritt wieder rückgängig machen.

Bei dieser Vorgehensweise eröffnen sich aber erneut Fragen und Probleme:

- Wenn ich ein INSERT rückgängig machen will, benutze ich dazu das von mir selbst programmierte DELETE oder eine Löschfunktion aus der Datenbank-Bibliothek?
- Wenn ich ein gelöschtes Objekt wieder in die Datenbank zurückschreiben will, erfordert auch das wieder Code der natürlich auch fehlerhaft sein kann
- Wenn ich meine eigenen INSERT und DELETE-Funktionen testen will, welche davon führe ich zuerst aus ?
- Ich kann nach dem Test die Datenbank wieder komplett löschen in dem ich eine TearDown-Methode aufrufe. Auch hier können wieder Fehler versteckt sein
- Ich kann den Datenbankzustand nach den Testen vernachlässigen und einfach ein Backup der Datenbank drüberspielen. Einerseits eine vielleicht akzeptable Lösung um den gültigen DB-Zustand zu garantieren, andererseits aber auch ein große Gefahrenquelle

Eine einzige, zusammenfassende Lösung für dieses Problem wäre es, alle Befehle des Tests in nur einer einzigen Transaktion ablaufen zu lassen. Nach dem Test würde ein einziges Rollback genügen. Doch auch hierbei können wieder Probleme auftreten, da die Objekte einerseits mit inneren Transaktionen, andererseits mit äußeren Transaktionen arbeiten müssten.

2 Testen von Datenbankcode

2.1 Die adHoc-Methode

In der Regel läuft ein Datenbankcode-Test ohne adäquate Testumgebung so ab:

- Die Datenbank wird aufgesetzt und die Applikation meldet sich an
- Daten werden in die Datenbank geschrieben
- Ein SELECT-Statement wird abgesetzt um anzuzeigen was in der Datenbank steht

Das Problem bei dieser Vorgehensweise ist zuerst die Beschränktheit in Bezug auf die Datenmenge. Niemand wird von Hand 100 Datensätze schreiben, löschen und aktualisieren und nach jedem Befehl prüfen ob die Datenbank noch in einem gültigen Zustand ist – diese Tests sollten also automatisiert werden.
Auf der anderen Seite wird zumeist nur sporadisch getestet und nicht nach einem festgelegten Zeitplan.

Es kann somit passieren dass eine Funktionalität, die nur selten genutzt wird beim Test zum Zeitpunkt x mitgetestet wird und einwandfrei funktioniert. In der Zwischenzeit wird aber der Code an einer anderen Stelle soweit verändert dass auch diese Funktionalität nur mehr eingeschränkt leistungsfähig ist. Der Programmierer testet aber zum Zeitpunkt x+1 dieses Feature nicht mehr, da es ja beim letzten Mal funktioniert hat.
Selbst wenn dieses Codestück nur in einem von 1000 Fällen ausgeführt wird, entdeckt spätestens der Kunde diesen Fehler mit ziemlicher Sicherheit.

Eine automatisierte Testumgebung verlangt nur einmaliges Nachdenken über bestimmte Test- und Ausnahmefälle. Diese werden dann in den Testablauf integriert und der Programmierer oder Tester muss sich nicht mehr darum kümmern, weil die Testumgebung den Test automatisch bei jedem Testlauf erneut überprüft.

2.2 Automatisierung von Testläufen – 2 Ansatzpunkte

Händisch durchgeführte Tests werden normalerweise der Einfachheit halber auf der gleichen Datenbank ausgeführt auf der auch die Applikation später arbeiten wird. Das Aufsetzen einer zweiten Datenbank nur für Testzwecke ist meistens den Aufwand nicht wert.

Falls man sich aber dafür entscheidet eine spezielle Testumgebung zu installieren und zu benutzen stellt sich die Frage wie sollte man dabei vorgehen:

2.2.1 Eigene Testdatenbank anlegen

Um für die Testumgebung einen relevanten Datenbankzugriff zu ermöglichen wird in diesem Fall eine eigene Datenbank angelegt die ausschließlich für JUnit-Tests benutzt wird.
In einer professionellen Entwicklungsumgebung ist es sogar notwendig vier verschiedene Datenbanken anzulegen:

Production database
Local development
Shared development
Deployment/Integration

Es gibt dazu einige Regeln die befolgt werden sollten um keine übermächtigen und lang dauernde Testläufe zu produzieren:
Die Datenmenge in einer Testdatenbank sollte z.B. alle Möglichkeiten von Eintragungen überdecken, aber nicht aus einer riesigen Ansammlung gleichwertiger Datensätze bestehen, nur um dem realen Einsatz so ähnlich wie möglich zu sein.

Gegenstimmen behaupten, dass der Ansatz mit Testdatenbanken viel zu aufwändig ist, da gerade in JUnit sehr viel Arbeit mit Skripten erledigt werden kann die die gleiche Funktionalität simulieren. Abgesehen davon dauert der Test mit einer echten Datenbank viel länger.

Zusammenfassend gibt es folgende Betrachtungsweisen:

Argumente für Testdatenbanken:

- Man kann reale Objekte in einer realen Umgebung testen und ist so nahe wie möglich an einer realen Anwendung
- Es lassen sich auch gleich JDBC/RDBMS-Fehler erkennen und dementsprechend behandeln
- Man muss für den Fall, dass man mit einer langsamen Datenbank arbeitet, nicht alle Tests immer und immer wieder durchführen. Es reicht, wenn man sie nur einmal bis zweimal täglich ablaufen lässt.

Argumente gegen Testdatenbanken:

- Man muss drei bis vier Datenbanken anlegen und einsetzen
- Diese Datenbanken müssen durch den ganzen Projektverlauf mitgeschleppt werden
- Dadurch ergeben sich andere, neue Probleme
- Durch Skripts kann man die Arbeit von Datenbank leicht nachbilden
- Tests auf einer richtigen Datenbank sind viel langsamer
- Wenn man irgendwelche Besonderheiten simulieren will, muss man zuvor die Datenbank aufwändig in den gewünschten Zustand bringen.

2.2.2 Einsatz von Mock-Objekten

Ein anderer Ansatzpunkt beim Testen von Datenbankcode ist der Einsatz von Mock-Objekten.

Was ist ein Mock-Objekt ?

Ein Mock-Objekt ist ein kleine Stück Software das die Eigenschaften eines großen Stücks Software (z.B. eines Programms oder einer Datenbank) nachbildet. Es kann ganz genau definiert werden wie ein Mock-Objekt auf welche Eingabe reagiert und welche Ausgabe man dafür erwartet.

Ein **Mock-Objekt** ……

……ist schnell
……ist schnell gebaut
……ist schnell konfiguriert
……ist deterministisch
……hat voraussagbares Verhalten
……hat kein Benutzerinterface
……ist direkt ansprechbar (im Bezug auf seine Eigenschaften)

Argumente für Mock-Objekte:

- Sind viel schneller als reale Datenbanken
- Simulieren auch exzeptions von Datenbanken
- Erlauben durch einfache Konfigurierbarkeit eine größe Menge an Tests. Viele unwahrscheinliche Ausnahmefälle die sonst umständlich konstruiert werden müssten, können leicht simuliert werden.

Argumente gegen Mock-Objekte:

- Die Benutzung von Mock-Objekten setzt die Erstellung einer künstlichen Umgebung für die Testabläufe voraus
- Falls sich die Datenbank ändert schlagen die Tests trotzdem nicht fehl, weil man ja keine Datenbank benutzt.

3. Professionelle Testumgebungen und Werkzeuge

Es gibt viele verschiedene Softwaretools die das Testen und die Analyse von Code unterstützen. Wie beschränken uns hier nur auf Java-Tools und JDBC-Datenbanken, weil diese Produkte zumeist frei erhältlich sind und Informationen zur Software nicht von kommerzieller Werbung beeinflusst werden.
DBUnit ist frei erhältlich und steht seit Mai 2003 in der Version 2.1 zum Download bereit.

3.1 DBUnit

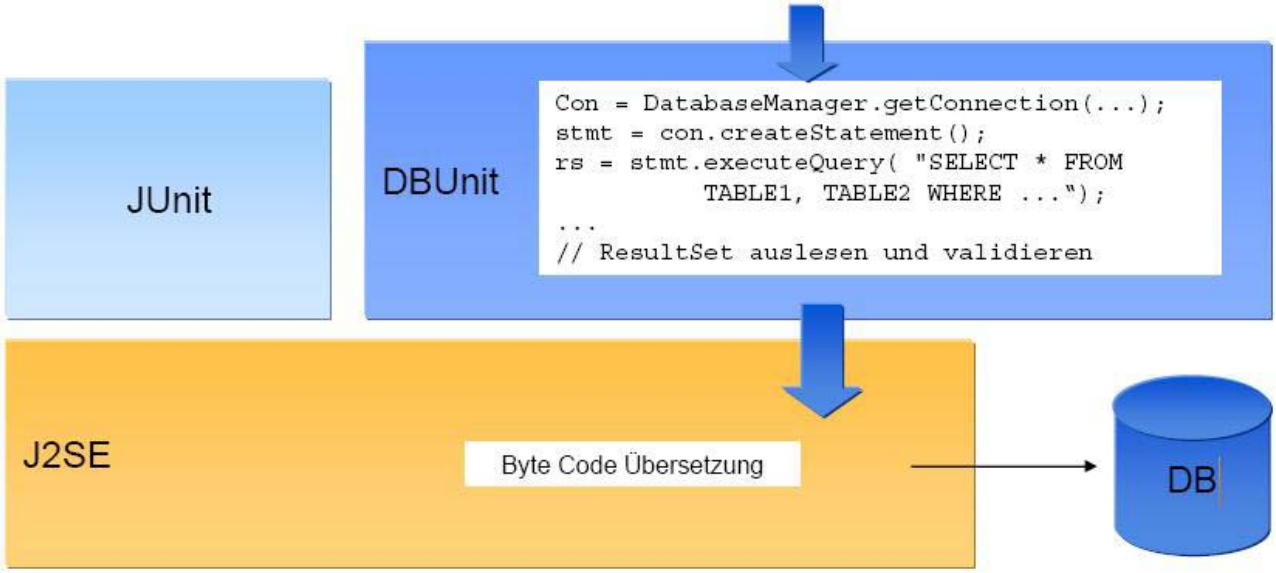
Quelle: http://dbunit.sourceforge.net/

Eine gute Hilfe auf dem Gebiet der automatischen Tests ist DBUnit.
DBUnit ist Bestandteil des JUnit-Frameworks und ist speziell für den Test von Datenbankcode gedacht.

DBUnit arbeitet in der aktuellsten Version mit folgenden Datenbanken zusammen:

- OracleDatabase
- MsSQL
- MySQL
- IbmDB2
- IbmInformix
- HypersonicSQL
- PostgreSQL
- SybaseSQL
- InterBase
- OpenBase8
- SapDB
- MaxDB

DBUnit stellt eine Schnittstelle zwischen der zu testenden Applikation und der Java Virtual Machine her und bietet in dieser Position vielfältige Möglichkeiten im Rahmen von JUnit.



DBUnit kann die Datenbank während den Tests immer wieder in einen gültigen Zustand zurückversetzen. Dadurch können zumindest alle Fehler die durch äußeren Einfluss auftreten können zur Gänze ausgeschlossen werden.

Es arbeitet hauptsächlich mit Mock-Objekten und Stubs auf Basis von XML um dem zu testenden Code während der Testdurchläufe eine gültige Umgebung bieten zu können. Der Tester hat zwischen den Tests jederzeit die Möglichkeit den Datenbankzustand zu ändern.

Beispiel:

Wir wollen jetzt zeigen wie ein Datenbankcodetest mit DBUnit aussehen könnte und was einem das Framework dabei an Werkzeugunterstützung anbietet.

Als Ausgangsbasis haben wir eine Datenbank nach folgendem Schema:

```
mysql> desc EMPLOYEE;
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| employeeUid |  bigint(20)    |      | PRI | 0       |       |
| startDate    | timestamp(14) | YES  |     | NULL    |       |
| firstName    | varchar(50)   | YES  |     | NULL    |       |
| ssn          | varchar(50)   | YES  |     | NULL    |       |
| lastName     | varchar(50)   | YES  |     | NULL    |       |
+--------------+--------------+------+-----+---------+-------+
6 rows in set (0.00 sec)
```

Eine mögliche Ausprägung der Datenbank könnte dann so aussehen:

```
mysql> select * from employee;
+--------------+----------------+-----------+--------------+-------------+
| employeeUid  | startDate      | firstName |   ssn        | lastName    |
+--------------+----------------+-----------+--------------+-------------+
|            1 | 20010101010101 |  Andrew   | 000-29-2030  | Glover      |
+--------------+----------------+-----------+--------------+-------------+
1 row in set (0.00 sec)
```

Diese Datenbank wird in ein XML-Schema umgelegt und sieht im konkreten Fall so aus:

```
<EMPLOYEE employee_uid='1'
        start_date='2001-11-01'
        first_name='Andrew'
        ssn='xxx-xx-xxxx'
        last_name='Glover' />
```

Mit diesen so genannten Seed-Files ist es jetzt relativ einfach möglich ein Datenbankkonstrukt mit beliebig vielen Benutzern zu füllen ohne dabei wirklich die Datenbank selbst zu benutzen.

```
<dataset>
  <EMPLOYEE employee_uid='1'
        start_date='2001-01-01'
        first_name='Drew' ssn='000-29-2030'
        last_name='Smith' />

  <EMPLOYEE employee_uid='2'
        start_date='2002-04-04'
        first_name='Nick' ssn='000-90-0000'
        last_name='Marquiss' />

  <EMPLOYEE employee_uid='3'
        start_date='2003-06-03'
        first_name='Jose' ssn='000-67-0000'
        last_name='Whitson' />
</dataset>
```

DBUnit kann in der Folge mit diesem, in XML vorliegenden Datenbankschema arbeiten. DBUnit unterstützt außer der „herkömmlichen" Programmierung von Testfällen auch den Einsatz von Build-Files. Auf beide Möglichkeiten wollen wir hier näher eingehen:

3.1.1 Testen ohne Ant-Builds:

DBUnit bietet eine abstrakte Klasse mit dem Namen *DatabaseTestCase* die direkt von der JUnit-Klasse *TestCase* erbt.
Der Entwickler muss beim Einsatz von *DatabaseTestCase* 2 Methoden selbst schreiben und zwar *getConnection()* und *getDataSet()*.

getConnection()
Wie eine Verbindung zu einem beliebigen Datenbankkonstrukt in Java geschaffen wird ist natürlich vom Einsatz der jeweiligen Datenbank abhängig und erfordert je nach Produkt mehr oder weniger Zusatzsoftware und wissen.
Es ist erforderlich dass der Entwickler oder Tester die Datenbankverbindung erfolgreich anlegt und *getConnection()* ein *IDatabaseConnection* Objekt zurückliefert.

getDataSet()
Diese Methode erwartet ein *IDataSet* Objekt das den Inhalt eines Files nach der Vorlage der zuvor generierten XML-Beschreibung liefert.

Darüber hinaus bietet DBUnit noch zwei Methoden um den Zustand der Datenbank vor und nach dem Test direkt zu verändern, *getSetUpOperation()* und *getTearDownOperation()*

Es gibt verschiedene Vorgehensweisen wie diese Methoden zu füllen sind. Eine beliebte Strategie ist es getSetUpOperation() also ein REFRESH durchführen zu lassen bei dem die Datenbank mit den jeweiligen Inputdaten aktualisiert wird.
Eine andere Möglichkeit – und unsere Meinung nach die sicherere – ist es in der Datenbank alle angesprochenen Tabelle vollständig zu löschen und von vorne mit den in XML generierten Daten zu füllen. Damit ist man in jedem Fall sicher, dass die Datenbank beim nächsten Test mit den richtigen Datensätzen arbeitet.

Konkretes Beispiel:

An einem einfachen Beispiel wollen wir jetzt zeigen wie sich DBUnit recht leicht dafür einsetzen lässt um alle möglichen Methoden der Datenbankmanipulation in einer Datenbankapplikation durchzutesten.

Eine dem obigen Datensatz entsprechende Anwendung könnte in etwa folgende Methoden anbieten:

public void createEmployee(EmployeeValueObject emplVo)
public EmployeeValueObject getEmployeeBySocialSecNum(String ssn)
public void updateEmployee(EmployeeValueObject emplVo)
public void deleteEmployee(EmployeeValueObject emplVo)

Diese Methoden benötigen eine befüllte Datenbank. Es ist aber nicht möglich zu garantieren dass *createEmployee(EmployeeValueObject emplVo)* die Datenbank auch in einem konsistenten Zustand zurücklässt um alle anderen Methoden zu testen. Deshalb wird DBUnit hier eingesetzt um die Datenbank über ein SetUP in einen gültigen Zustand zu bringen

Um auch *createEmployee(EmployeeValueObject emplVo)* testen zu können, generiert DBUnit Einträge mit sinnlosem Inhalt um eventuelle nicht behandelte Exceptions zu provozieren.

Um die oben aufgezählten Methoden zu testen verwenden wir das zuvor angelegte XML-File:

```
<dataset>
  <EMPLOYEE employee_uid='1'
        start_date='2001-01-01'
        first_name='Drew' ssn='333-29-9999'
        last_name='Smith' />
  <EMPLOYEE employee_uid='2'
        start_date='2002-04-04'
        first_name='Nick' ssn='222-90-1111'
        last_name='Marquiss' />
  <EMPLOYEE employee_uid='3'
        start_date='2003-06-03'
        first_name='Jose' ssn='111-67-2222'
        last_name='Whitson' />
</dataset>
```

DBUnit übernimmt die gesamte Verwaltung der Datenbank – der Tester kann sich jetzt auf aussagekräftige Testfälle konzentrieren ohne auch noch die Datenbank im Auge behalten zu müssen.

Der Test der Methode *getEmployeeBysocielSecNum()* könnte dann so aussehen:

```
public void testFindBySSN() throws Exception{
        EmployeeFacade facade = //Datenbankkonstrukt
        EmployeeValueObject vo=
        facade.getEmployeeBySocialSecNum("333-29-9999");
        TestCase.assertNotNull("vo shouldn't be null", vo);
        TestCase.assertEquals("should be Drew", "Drew", vo.getFirstName());
        TestCase.assertEquals("should be Smith", "Smith", vo.getLastName());
}
```

Der Tester muss jetzt angeben bei welchen Testfällen er welche Rückgaben aus der Datenbank erwartet und kann somit natürlich den Test dieser Methode sehr leicht automatisieren.
Jetzt wird die Create-Methode getestet. Das geht fast genauso einfach wie zuvor:

```
public void testEmployeeCreate() throws Exception{
        EmployeeValueObject empVo =  new EmployeeValueObject();
         empVo.setFirstName("Noah");
         empVo.setLastName("Awan");
         empVo.setSSN("564-55-5555");
         EmployeeFacade empFacade = //Datenbankkonstrukt
         empFacade.createEmployee(empVo);
}
```

Der neue Benutzer wird jetzt angelegt. Diese Methode kann genau wie oben einfach automatisiert und auch mit ausgefallenen Daten angewandt werden.

Um die Methode *updateEmployee(EmployeeValueObject emplVo)* zu testen ist etwas mehr an Aufwand durchzuführen. Ein beliebiger Datensatz muss zuvor gefunden werden. Danach muss er irgendwie verändert (upgedatet) werden und sollte bei einer erneuten Suche natürlich wieder gefunden werden. Abschließend muss noch der gewünschte update-Wert mit dem tatsächlich gefundenen neuen Wert abgeglichen werden.

```
public void testUpdateEmployee() throws Exception{
        EmployeeFacade facade = //Datenbankkonstrukt
         EmployeeValueObject vo =
        facade.getEmployeeBySocialSecNum("111-67-2222");
         TestCase.assertNotNull("vo was null", vo);
         TestCase.assertEquals("first name should be Jose",
        "Jose", vo.getFirstName());
         vo.setFirstName("Ramon");
         facade.updateEmployee(vo);
         EmployeeValueObject newVo=
        facade.getEmployeeBySocialSecNum("111-67-2222");
         TestCase.assertNotNull("vo was null", newVo);

         TestCase.assertEquals("name should be Ramon",  "Ramon",
newVo.getFirstName());
}
```

Die letzte nicht getestete Methode, deleteEmployee, ist wieder etwas einfacher zu überprüfen. Ein Datensatz muss zuvor gesucht und gefunden, gelöscht und erneut gesucht werden. Der Datensatz darf danach natürlich nicht mehr gefunden werden, sonst ist von einer Fehlimplementierung auszugehen.

```
public void testDeleteEmployee() throws Exception{
        EmployeeFacade facade = //obtain façade
        EmployeeValueObject vo =
        facade.getEmployeeBySocialSecNum("222-90-1111");
        TestCase.assertNotNull("vo was null", vo);
        facade.deleteEmployee(vo);
        try{
                EmployeeValueObject newVo =
                facade.getEmployeeBySocialSecNum("222-90-1111");
                TestCase.fail("returned removed employee");
        }catch(Exception e){}
}
```

So wie diese 4 Methoden können natürlich beliebige andere Operationen auf die Datenbank sehr einfach überprüft werden. Einmal geschriebene Testfälle bleiben in der Testumgebung und werden bei jedem Test mit ausgeführt bis zu einem SetUp oder TearDown.
Eine Automatisierung der betreffenden Tests ist auch dementsprechend einfach zu realisieren.

3.1.2 Testen mit Ant-Builds:

Abgesehen von der oben gezeigten Programmierung von Testfällen bietet DBUnit noch ein „Ant"-Modul das die Benutzung von so genannten Ant-Build-Files ermöglicht.

Ant ist ein java-basiertes Tool das ähnlich wie „make" arbeitet, aber - nach Herstellerangaben ohne dessen negative Eigenschaften.
Ant ist speziell dafür entwickelt worden um plattformunabhängige Builds ohne die Restriktionen der anderen auf dem Markt erhältlichen Tools zu ermöglichen.
Ant ist gerade deswegen java-basiert und arbeitet mit Konfigurationsfiles auf XML-Basis um nicht mehr von der Shell und den zugrunde liegenden Betriebssystem-Einschränkungen betroffen zu sein.

Um beim Ant-Build eine Datenbank zu erreichen und ein vordefiniertes Seed-File einzutragen sieht so aus:

```
<taskdef name="dbunit" classname="org.dbunit.ant.DbUnitTask"/>
        <dbunit driver=" org.gjt.mm.mysql.Driver "
                url=" jdbc:mysql://127.0.0.1/hr "
                userid="hr"
                password="hr">
        <operation type="INSERT"  src="seedFile.xml"/>
</dbunit>
```

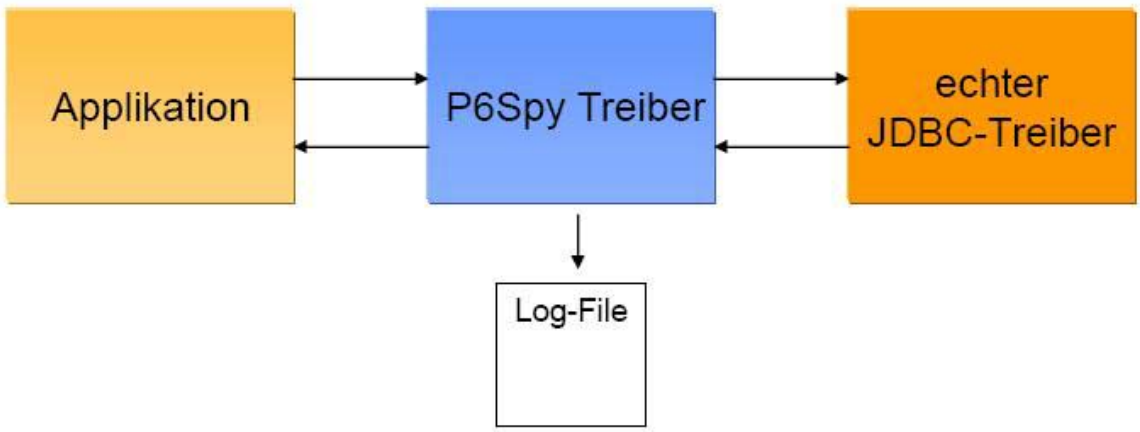
Das Löschen der Tabellen aus der Datenbank verläuft nach demselben Muster als Ant-Build:

```
<dbunit driver=" org.gjt.mm.mysql.Driver " url=" jdbc:mysql://127.0.0.1/hr "
        userid="hr"
        password="hr">
        <operation type="DELETE" src="seedFile.xml"/>
</dbunit>
```

Man sieht dass diese Art der Datenbankmanipulation sehr leicht lesbar und auch sehr schnell ausführbar ist. Mit Hilfe dieser Build-Files kann der Datenbank-Zustand effizient und einfach im Testverlauf gesetzt und verändert werden.

3.2 P6Spy

P6Spy ist ein Werkzeug mit dem der Datentransfer zwischen einer Anwendung und dem JDBC-Treiber der Datenbank abgefangen, analysiert und gegebenenfalls auch verändert werden kann. Es ist nicht gleichwertig mit DBUnit zu betrachten, da es sich nicht um eine automatische Testumgebung handelt. P6Spy kann aber gerade bei Datenbankanwendungen sehr große Hilfe leisten und die Fehleranalyse und –behebung sehr stark erleichtern.
P6Spy lässt den Code von Applikation und Datenbank völlig unbeeinflusst in jedes Projekt integrieren und installiert sich zwischen den beiden Schnittstellen.
Für JBoss, ATG, Orion, JOnAS, iPlanet, WebLogic, WebSphere, Resin and Tomcat sind einfache Tutorials zur Installation verfügbar



P6Spy enthält 2 Module:

- P6Log
  Damit können alle Operationen die die Zielapplikation an die Datenbank über den JDBC-Treiber absetzt mitgeloggt und analysiert werden. Die geloggten SQL-Statements können im Anschluss analysiert werden und zur Verbesserung von Fehlverhalten aber auch zur Optimierung der Systemperformance benutzt werden.

- P6Outage
  Mit P6Outage lässt sich der Logging Performance Verlust minimieren indem nur lange laufende SQL-Statements herausgefiltert werden.
  In P6Outage kann per Configuration-File eine minimale Laufzeit von Statements eingestellt werden. Wenn der Wert z.B. auf 10 gesetzt ist, filtert P6Outage nur Befehle die länger als diese 10 Sekunden laufen.