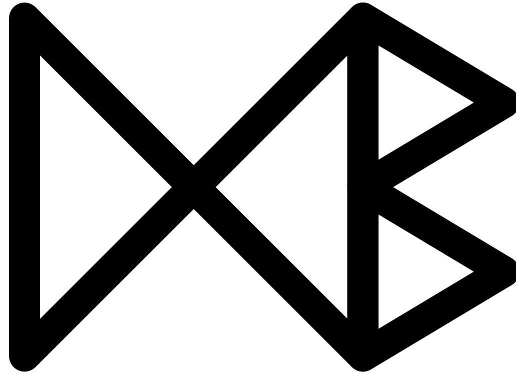

A Unit-Test Framework for Database Applications

Claus A. Christensen, Steen Gundersborg, Kristian de Linde, and Kristian Torp

May 8, 2006

TR-15

A DB Technical Report

Title	A Unit-Test Framework for Database Applications
	Copyright © 2006 Claus A. Christensen, Steen Gundersborg, Kristian de Linde, and Kristian Torp. All rights reserved.
Author(s)	Claus A. Christensen, Steen Gundersborg, Kristian de Linde, and Kristian Torp
Publication History	May 2006. A DB Technical Report

For additional information, see the DB TECH REPORTS homepage: ⟨www.cs.aau.dk/DBTR⟩.

Any software made available via DB TECH REPORTS *is provided "as is" and without any express or implied warranties, including, without limitation, the implied warranty of merchantability and fitness for a particular purpose.*

The DB TECH REPORTS icon is made from two letters in an early version of the Rune alphabet, which was used by the Vikings, among others. Runes have angular shapes and lack horizontal lines because the primary storage medium was wood, although they may also be found on jewelry, tools, and weapons. Runes were perceived as having magic, hidden powers. The first letter in the logo is "Dagaz," the rune for day or daylight and the phonetic equivalent of "d." Its meanings include happiness, activity, and satisfaction. The second letter is "Berkano," which is associated with the birch tree. Its divinatory meanings include health, new beginnings, growth, plenty, and clearance. It is associated with Idun, goddess of Spring, and with fertility. It is the phonetic equivalent of "b."

Abstract

The outcome of a test of an application that stores data in a database naturally depends on the state of the database. It is therefore important that test developers are able to set up and tear down database states in a simple and efficient manner. In existing unit-test frameworks, setting up and tearing down such *test fixtures* is labor intensive and often requires copy-and-paste of code. This paper presents an extension to existing unit-test frameworks that allows unit tests to reuse data inserted by other unit tests in a very structured fashion. With this approach, the test fixture for each unit test can be minimized. In addition, the reuse between unit tests can speed up the execution of test suites. A performance test on a medium-size project shows a 40% speed up and an estimated 25% reduction in the number of lines of test code.

1 Introduction

Unit-test frameworks are widely used by software developers to assist them in testing the correctness of software. This wide spread use of unit tests is best illustrated by the eXtreme Programming methodology [2] where a coding rule says "Code the unit test first". In the development of software, databases are very often used for storing data persistently. The code that query and update the database must naturally also be unit tested.

There exist a large number of unit-test frameworks [10, 16]. In these unit-test frameworks, each *test suite* consists of a number of *test cases* that again consists of a number of *test methods*. In an object-oriented programming language, a test case is implemented as a class, a test method is implemented as a method, and a test suite as a collection of classes.

The existing unit-test frameworks make the central assumption, that all test methods must be independent [10]. As an example, consider testing a university registrar application where data is stored in an underlying relational database. Here, there is a test method that checks that a student can enroll in a course. The test developer must ensure that this test method can be executed independently. In contrast, the table that the enrollment is inserted into can have multiple foreign keys to other tables. We assume that the enrollment table has two foreign keys to tables that stores information on the courses and the students. Unless these two tables contain reasonable values, the enrollment test method will fail due to integrity constraint violations. To insert reasonable values the test developer has to build a *test fixture* that in the registrar example inserts rows into the course and student tables that the enrollment test method use.

Unit-test frameworks use a `setUp` method for building the test fixture and a `tearDown` method for removing it again. Because test methods are independent the test developer cannot reuse or inherit code from other test cases when building the test fixture. This typically leads to copy-and-paste of code, which makes test fixtures labor intensive to build and maintain. In addition, if another test method in our registrar example queries the number of students enrolled in a course, and both the insert and the query test methods are executed as a part of testing the entire application, both test methods need to set up and tear down the test fixture. This is unnecessary since the query test method can reuse the test fixture provided by the insert test method. It is time saving to reuse test fixtures for applications that store data in a database because it is very time consuming to build test fixtures that inserts rows into a database and to remove the test fixtures again by deleting the same rows from the database.

In this paper, we argue that when testing software that stores data in a database it is an advantage to allow both test cases and test methods to be dependent in a very structured fashion. This breaks with a central assumption in existing unit-test frameworks that test methods must be independent. However, it can solve the two problems described above: 1) Make it less labor intensive to build and maintain test fixtures and 2) make it faster to execute test cases and test suites.

The paper is organized as follows. In Section 2, related work is discussed. Section 3 introduces a small test database and an application that are used as running examples. Section 4 describes the framework

constructs in details. How to reuse test fixtures is discussed in Section 5. A prototype implementation is described in Section 6. Finally, Section 7 concludes the paper and points to directions of future research.

2 Related Work

The JUnit testing framework [10] is the de-facto standard for implementing Java unit tests. The work presented in this paper extends the existing frameworks by allowing reuse (dependencies) between test methods and test cases. This makes test fixtures simpler to build and test suites faster to execute. Similarly, the utPLSQL framework [16] is a unit-test framework for Oracle's PL/SQL programming language. The framework is modeled after the JUnit framework, taking into consideration the characteristics of PL/SQL.

The DbUnit testing framework [1] is a JUnit extension that is aimed specifically at database-driven applications. Before a test method is executed, the framework puts the database in a known state, which per default is the empty state. This is done by truncating all tables. As most, if not all, frameworks build on top of the JUnit framework, DbUnit makes the assumption that test methods are independent. The framework presented in this paper allows test data and production data to coexist. In addition, we do not rely on truncating tables. An application may not have sufficient privileges in the database to do this.

Mock objects [7] are often used as stubs when unit-testing objects that participate in complex relationships. When testing database applications it is simply not possible to use mock objects because this would lead to integrity constraint violations. A stub cannot be used, real data has to be present in the database.

How to test the SQL statements executed by an application via for example JDBC is discussed in [8, 12]. Here, the actual code in the test methods is discussed. This work is orthogonal to the work presented in this paper. We look at how test methods and test cases can interact.

In [4], Chays et al. present a design of a framework for testing database applications. They discuss the role of the database state, which makes testing database applications different from testing applications that do not store data persistently. Their contribution is at a more conceptual level compared to this paper.

Testing complex database transactions is the topic in [6]. Here, a tool for checking the consistency of transactions executed in isolation is presented. In this paper, transactions are not considered and the work presented in [6] can be combined with the work presented here.

General books on software testing [2, 9, 11, 14] have no specific discussion of the testing of database applications. In [11], Lewis considers how to test if integrity constraints in a database are fulfilled something most DBMSs do automatically.

In [5], Daou et al categorize the problems that SQL inflicts on regression testing. This paper differs from [5] in that it considers unit testing, whereas Daou et al only consider regression testing and Daou et al do not discuss how to handle the test fixture for a single test method.

3 The University Example

This section introduces a database schema and a simple application that query and update the tables in the database. The database and the application are used as running examples in the paper.

3.1 The Database Schema

The UML class diagram in Figure 1 represents a university. Each class corresponds to a table. Columns are represented as attributes where primary keys are marked by (pk). Foreign keys are shown as associations between classes.

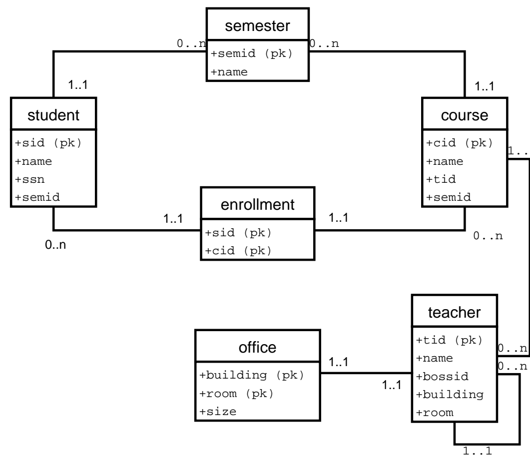

Figure 1: The University Schema

The university has a number of students. Information about these is stored in the *student* table. This table has four columns: `sid` the student id and the primary key, `name` the student's name, `ssn` the social security number, and `semid` the semester id. The *course* table stores information on courses. This table has four columns: `cid` the course id and the primary key, `name` the course name, `tid` the teacher id and foreign key to table *teacher*, and `semid` the semester id. The *enrollment* table is a relationship between the *student* and the *course* tables. It has two columns: `sid` the student id and `cid` the course id. Both columns are part of the primary key and are foreign keys to the *student* and the *course* tables, respectively. The *semester* table has two columns: `semid` the semester id, and primary key, and `name` the name of the semester. The *office* table has three columns: `building` the name of the building, `room` the room number, and `size` the size of the room. The primary key is the columns `building` and `room`. The *teacher* table has five columns: `tid` the teacher id and primary key, `name` the teacher's name, `bossid` the id of the teacher's boss and a foreign key to the *teacher* table itself, `building` the building, and `room` the room. The last two columns are a foreign key to the *office* table.

3.2 The Application

The example application is a table wrapper API that creates a class for each table in the underlying database and provides three methods. The `ins` method that inserts a row into the table. One argument for each column in the table is provided. The `del` method that deletes a single row from the underlying table. The primary key is given as argument. Finally, the `exist` method that checks if a row is stored in the table. The primary key of the row is given as argument and the method returns true or false. The method names are chosen such that they do not collide with SQL reserved words. The table wrapper API for the *office* and *teacher* tables is shown in the UML class diagram in Figure 2.

Figure 2: The Office and Teacher Table Wrapper APIs

It is important to emphasize that the unit-test framework presented in this paper is not limited to test such simple table wrapper APIs. It can be used to test all types of applications but is targeted towards applications that store data in a relational database.

4 The Framework Constructs

The framework can be divided into four main constructions that are shown in Figure 3. The first is the class `TestCase`, the second is the subclasses of `TestCase`, the third is the class `UnitTest`, and the final is the class `InvocationStack`. These constructs are described in details in the following.

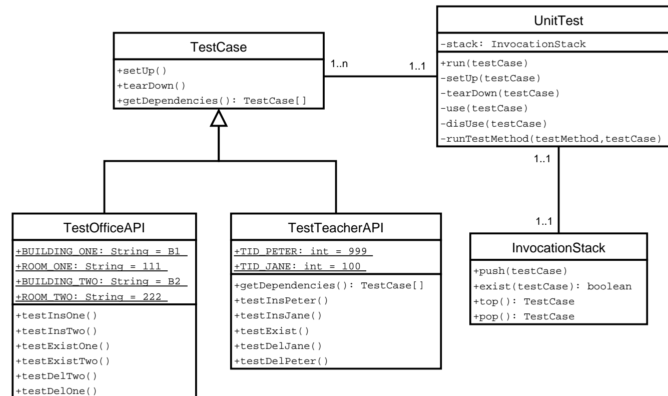

Figure 3: Test Framework Constructs

4.1 The `TestCase` Class

The class `TestCase` is supplied by the framework and is the superclass of the test cases build by the test developers, i.e., it serves the same purpose as the class `TestCase` in JUnit [10]. The class has three the public methods `setUp`, `tearDown`, and `getDependencies`. The first two methods are concrete with empty bodies. The last method returns an empty array by default.

The methods `setUp` and `tearDown` have the same purpose as in JUnit. They set up or tear down the test fixture before or after each test method is executed, respectively. The method `getDependencies` returns the test cases that a test case depends on. As an example, the table wrapper API for the *teacher* table depends on the table wrapper API for the *office* table because there is a foreign key from table *teacher* to table *office*.

All the methods on the `TestCase` class are called by the class `UnitTest` that is described next.

4.2 The `UnitTest` Class

The class `UnitTest` is used to execute the test methods on a single test case and a set of test cases. The methods and attributes defined on the `UnitTest` class are explained in this section.

4.2.1 The `stack` Variable

The private variable `stack` of type `InvocationStack` is used to control the test cases that are in use. The purpose of the variable is illustrated in Section 4.4. In the methods introduced in the following sections, the `stack` variable is simply used as a LIFO queue.

4.2.2 The `run` Method

The purpose of the public `run` method is to execute all the test methods for a single test case. Note that compared to JUnit the `run` method has been moved from the `TestCase` class to the `UnitTest` class.

The pseudo code for the `run` method is shown in Listing 1 using a Python-like [15] syntax where comments are marked by #.

```
def run(testCase):
    # set up the dependencies
    setUp(testCase)
    # execute all the test methods
    for testMethod in testCase:
        runTestMethod(testMethod, testCase)
    # tear down the dependencies
    tearDown(testCase)
```

Listing 1: The `run` Method

The `run` method is called with a `TestCase` object, e.g., an instance of the `TestOfficeAPI` class. The method then calls the private method `setUp` on the class `UnitTest` in line 3. This call builds the test fixture for the entire test case. In lines 5–6, the method loops over all the test methods in the test case and calls these methods. Finally, in line 8 the private method `tearDown` on class `UnitTest` is called to tear down the test fixture.

Note that in the implementation, discussed in Section 6, lines 5–6 of the `run` method are implemented using the reflection capabilities of the implementation language. In general, reflection is used extensively in the implementation of the unit-test framework presented here, as it is the case in for example JUnit [3]. However, it is much simpler to explain the unit-test framework methods without considering reflection.

4.2.3 The `setUp` Method

The purpose of the private method `setUp` is to ensure that the test fixture for an entire test case is set up. This can be complicated. As an example, for the table wrapper API this is complicated due to the foreign key constraints. In more details, before we can execute the test methods for the *enrollment* table API it is necessary to have data in the *course* and *student* tables. Before we can insert data in the *student* table there must be data in the *semester* table. Before we can insert data in the *course* table there needs to be data in first the *office* table, second the *teacher* table, and finally the *semester* table, see also Figure 1.

The set up of the test fixture is achieved by having the `setUp` method call the `use` method on all test cases that the current test case depends on. These dependencies are specified by the test developer and can be retrieved using the `getDependencies` method on the `TestCase` class. The `setUp` method is specified in Listing 2.

```
def setUp(testCase):
    # if already set up then skip
    if stack.exist(testCase):
        return
    # use test cases this test case depends on
    for dependent in testCase.getDependencies():
        use(dependent)
    # tell test case is in use
    stack.push(testCase)
```

Listing 2: The `setUp` Method

The `setUp` method is called with a `TestCase` object as argument. In line 3, it is checked that the test case is not already in use. This is done by calling the method `exist` on the variable `stack` (of type `InvocationStack`). If the test case is already on the stack the `setUp` method returns in line 4. Otherwise, the `use` method is called in lines 6–7 for all the test cases that the current test case depends on. To get these dependent test cases the instance method `getDependencies` on the class `TestCase`

is called. When all the dependent test cases have been set up (via the `use` method) the test case is pushed on the stack in line 9 to show that the test case is now in use.

4.2.4 The `use` Method

The purpose of the private method `use` is to make the data used by a test case available to other test cases. The `use` method is shown in Listing 3.

```
1 def use(testCase):
2     # set up myself
3     setUp(testCase)
4     # reuse insert test methods
5     for insertTestMethod in testCase:
6         runTestMethod(insertTestMethod, testCase)
```

Listing 3: The `use` Method

The `use` method is called with a `TestCase` object as argument. In line 3, the `setUp` method on the `UnitTest` class is called. This is to ensure that the test case is set up. In lines 5–6, all the *insert test methods* (the test methods with the prefix *testIns*) on the test case are called. These methods insert data into the underlying table, e.g., the insert test methods for the test case `TestOfficeAPI` inserts data into the *office* table. Insert test methods are explained in details in Section 4.3.

4.2.5 The `runTestMethod` Method

The purpose of the private method `runTestMethod` is to execute a single test method. The `runTestMethod` method is specified in Listing 4.

```
1 def runTestMethod(testMethod, testCase):
2     # call set up method
3     testCase.setUp()
4     # execute the actual test method
5     testCase.testMethod()
6     # call the teardown method
7     testCase.tearDown()
```

Listing 4: The `runTestMethod` Method

The `runTestMethod` is called with a test method and a test case as arguments. First, the `setUp` method is called in line 3. Then, the actual test method is executed in line 5. Finally, the `tearDown` method is called in line 7. Note that these three method calls are on the `TestCase` object given as argument to the `runTestMethod` method.

The `setUp`, `use`, and `runTestMethod` methods on class `UnitTest` are now defined and it can be shown how the test fixture for the table wrapper API is set up and how test methods are executed. Looking at Figure 1, the dependencies between test cases are found by looking at the associations between classes. These dependencies are listed in Table 1.

Test Case	Dependencies
TestCourseAPI	TestSemesterAPI, TestTeacherAPI
TestEnrollmentAPI	TestCourseAPI, TestStudentAPI
TestOfficeAPI	
TestSemesterAPI	
TestStudentAPI	TestSemesterAPI
TestTeacherAPI	TestOfficeAPI

Table 1: Dependencies Between Test Cases

As examples, the table shows that the `TestCourseAPI` test case depends on the `TestSemester-API` and the `TestTeacherAPI` test cases. The `TestOfficeAPI` test case has no dependencies. As shown in Table 1 each test case only specifies the test cases that it depends on directly.

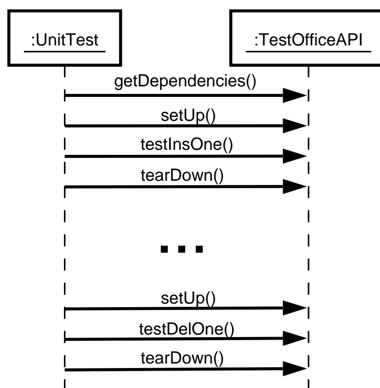

Figure 4: Set Up of Office Test Fixture

The sequence diagram in Figure 4 shows the process of setting up the test fixture for testing the table wrapper API for the *office* table. The `UnitTest` first calls the method `getDependencies` on the `Test-OfficeAPI` test case. It returns an empty array as it has no dependencies, see also Table 1. The next step is then a sequence of `setUp`, test method, and `tearDown` method calls. Except from the first call of the `getDependencies` method this corresponds to the JUnit framework.

The `TestOfficeAPI` test case has no dependencies and therefore no other test cases are involved in testing this test case. The next step is therefore to look at a test case that has dependencies. The set up of the test fixture for testing the `TestTeacherAPI` test case is shown in Figure 5. The `TestTeacherAPI` test case depends on `TestOfficeAPI` test case and the latter test case is therefore used in setting up the test fixture for the `TestTeacherAPI` test case.

As can be seen from Table 1, there needs to be data in the *office* table to test the *teacher* table. Looking at the sequence diagram in Figure 5, the method `getDependencies` is first called on the `TestTeacher-API`. This call returns that the `TestTeacherAPI` test case depends on the `TestOfficeAPI` test case. The next step is then to call the method `getDependencies` on the `TestOfficeAPI` test case. This test case has no dependencies, so the insert test methods on the `TestOfficeAPI` test case are called. These calls are all wrapped in calls to the `setUp` and `tearDown` methods. The calls of the insert test methods ensures that there is test data in the *office* table. If the test methods in the `TestTeacherAPI` test case use this data there will be no integrity constraint violations. This will be explained in details in Section 4.3.3. The next step is then to execute the test methods on the `TestTeacherAPI` test case.

It has now been shown how to set up a test fixture. The next step is to show how to tear down a test fixture. The methods `tearDown` and `disUse` methods are used for this and are introduced next.

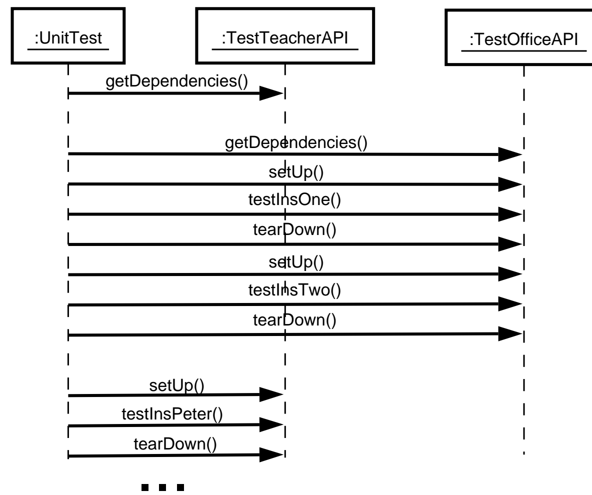

Figure 5: Set Up of Teacher Test Fixture

4.2.6 The `tearDown` Method

The purpose of the private method `tearDown` is to remove exactly the data inserted by the `setUp` method. The method is shown in Listing 5.

```
1  def tearDown(testCase):
2      # if not top of stack skip tear down for now
3      if stack.top() != testCase:
4          return
5      # tell that test case is no longer in use
6      stack.pop()
7      # tear down what test case depends on
8      for dependent in reverse
9                  (testCase.getDependencies()):
10         disUse(dependent)
```

Listing 5: The `tearDown` Method

The `tearDown` method is called with a `TestCase` object as argument. In line 3, it is first checked that the test case is the top of the stack. This is explained in details in Section 4.4. In line 6, the test case is popped of the stack to indicate that the data for this test case in no longer available. In lines 8–10, the `disUse` method of dependent test cases are called in the reverse order of which the `use` method on these test cases was called in the `setUp` method. The `disUse` method is explained next.

4.2.7 The `disUse` Method

The purpose of the private method `disUse` is to make a test case unavailable for other test cases. The method is shown in Listing 6.

```
1 def disUse(testCase):
2     # reuse all delete test methods
3     for delTestMethod in testCase:
4         runTestMethod(delTestMethod, testCase)
5     # tear down myself
6     tearDown(testCase)
```

Listing 6: The `disUse` Method

The `disUse` method is called with a `TestCase` object as argument. In lines 2–3, all delete test methods are called, i.e., test methods with the prefix *testDel*. This is explained in Section 4.3. Finally, the `tearDown` method from Listing 5 is called in line 6, this is to recursively tear down the test fixture.

Before it can be completely explained how the tear down of a test fixture works, the subclasses of the `TestCase` class need to be introduced. The `tearDown` method is then reintroduced in more details in Section 4.4.

4.3 The `TestCase` Subclasses

This section describes the subclasses of the `TestCase` class that the test developer builds. In addition, conventions that the test developer must follow are discussed. A test case consists of the following three parts.

- A list of dependent test cases
- An ordered list of test methods
- A set of public constants

4.3.1 Dependencies

The test developer must specify the test cases that a test case depends on. For the university schema in Figure 1, the dependencies are shown in Table 1. The dependencies can be queried via the `getDependencies` method that the test developer overrides. This is shown in Figure 3 where the class `TestTeacherAPI` overrides the `getDependencies` method. The `TestOfficeAPI` does not override this method because it has no dependencies.

The test developer only needs to specify the test cases that a test case depends on directly. As an example, the `TestEnrollmentAPI` test case only lists the two test cases `TestStudentAPI` and `TestCourseAPI` as dependencies even though the `TestEnrollmentAPI` test case also indirectly depends on the three test cases `TestOfficeAPI` `TestSemesterAPI`, and `TestTeacherAPI`. These indirect dependencies are found by recursively calling the `getDependencies` method. This is illustrated in Section 4.4.

There cannot be cycles in the dependencies between test cases. In addition, duplicates are not allowed in the set of dependent test cases. The unit-test framework automatically checks for both.

4.3.2 Test Methods

The test methods are the actual tests of the application, as in JUnit. However, in contrast to JUnit, the test methods are split into three distinct sets.

- Plain test methods
- Insert test methods
- Delete test methods

The plain test methods are similar to test methods in JUnit. The insert and delete test methods are used for inserting and deleting data from the underlying table, respectively. The insert test methods are the test methods that have names with the prefix *testIns*. The delete test methods are the test methods that names with the prefix *testDel*. All three sets of test methods are used as test methods. In addition, the insert and delete test methods are reused to set up and tear down the test fixture. This occurs when a test case depends on another test case. As an example, the insert test methods on the `TestOfficeAPI` test case are used to set up the test fixture for the `TestTeacherAPI` test case. In general, the insert test methods are called by the `use` method and the delete test methods are called by the `disUse` method.

As can be seen from Figure 3, the insert test methods are listed first, then plain test methods next, and finally the delete test methods. This is because the order in which the test methods are listed is also the order in which they are executed. The advantage of this is that it is simpler to build the test fixture for a test method. As an example, when executing the test method `testExistOne` on the `TestOfficeAPI` this test case can assume that data already exists in the underlying table. The `setUp` method on the `TestOfficeAPI` test case does not need to do this. The set up has been done by a previous test method, in this case the test method `testInsOne`.

That test methods depends on the execution of other test methods can lead to a ripple effect of *false negatives*, e.g., all the plain test methods fail because all the insert test methods failed, even though the error is in the insert test methods, and not in the plain test methods. This issue is discussed in Section 6.

The test developer must guarantee that all data inserted into the underlying database by the insert test cases are deleted from the database again by the delete test methods, i.e., when executing only the insert test methods followed by the delete test methods the database state should not be changed. Note that the test developer can test that the database state is unchanged by executing the insert test methods followed by the delete test methods twice. If the second execution leads to integrity constraint violations the database state is effected.

4.3.3 Public Constants

The test developer must make public the content of columns that other test cases can refer to. As an example, in Figure 3 the class `TestOfficeAPI` has four public constants `BUILDING_ONE`, `ROOM_ONE`, `BUILDING_TWO`, and `ROOM_TWO`. The values of these public constants are inserted into the primary key columns of two rows in the *office* table by the insert test methods `testInsOne` and `testInsTwo`. By making the content of the primary key public other test cases can use this data to insert foreign key to rows in the *office* table. As an example, the four public variables on the `TestOfficeAPI` test case are reused by the insert test cases on the `TestTeacherAPI` test case.

The public constants are also used to identify the row to delete from the database in the delete test method. This means that the test developer is not allowed to update the value of primary keys in the plain test methods.

The test developer must guarantee that the public constants refer to rows in the database after the insert test methods have been executed. Symmetrically, the test developer must guarantee that the public constants do not refer to rows in the database after the delete test methods have been executed.

4.4 The `InvocationStack` Class

The fourth and final construct in the unit-test framework, presented in this paper, is the `InvocationStack` class. The main purpose of the invocation stack is to keep track of which test cases have been set up and when a test case can be torn down. This is explained by the following example.

A directed graph representation of the dependencies between test cases for the university example is shown in Figure 6. The test cases are vertexes and the dependencies between them are edges. For now, please ignore the counts associated with each node. These are explained in Section 5.

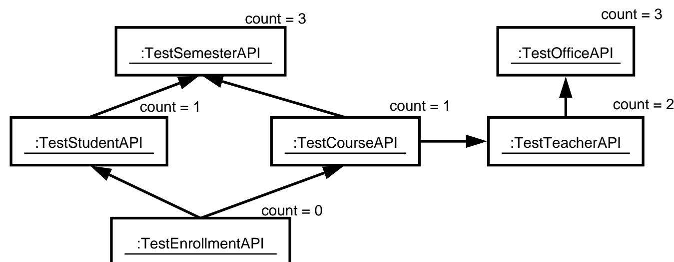

Figure 6: Graph Representing Dependencies between Test Cases

If the `TestOfficeAPI` and `TestTeacherAPI` test cases are ignored then Figure 6 shows that the `TestEnrollmentAPI` test case depends on the two test cases `TestStudentAPI` and `TestCourseAPI`. These two test cases depend on the same test case `TestSemesterAPI`. If a straight-forward approach is used for setting up test fixtures when running the `TestEnrollmentAPI` test case then the `TestStudentAPI` test case will set up the `TestSemesterAPI` and then it will set up it self. Next, the `TestCourseAPI` test case will set up the `TestSemesterAPI` test case again followed by setting up it self. The problem is that the `TestSemesterAPI` test case is set up twice. This will in the table wrapper API case lead to duplicate key insertion exceptions being thrown from the underlying database. The purpose of the invocation stack is to avoid this.

As shown in Figure 3, the `InvocationStack` class has the public methods `push`, `pop`, `top`, and `exist`. The first three methods are standard methods on a stack. The last method is used to check if an element, i.e., a `TestCase`, is already on the stack.

The `InvocationStack` is also used to ensure the robustness of the entire unit-test framework. If a failure occurs due to a system breakdown, all test cases that need to be torn down can be found on the invocation stack. The initial state of the underlying relational database can then be restored by calling the `disUse` method on the `UnitTest` class with each the test cases on the stack as argument one at a time. Note that the `InvocationStack` class is implemented as a persistent stack.

All the constructs in the unit-test framework are now introduced and Figure 7 shows how the stack is used when the test fixture for `TestEnrollmentAPI` test case is set up to be able to execute all the test methods on this test case.

The `UnitTest` class first calls the `getDependencies` method on the `TestEnrollmentAPI` test case. This returns the two dependent test cases `TestStudentAPI` and `TestCourseAPI`. Then the `getDependencies` method on the `TestStudentAPI` is called. It returns a single dependent test case `TestSemesterAPI` on which the `getDependencies` method is called again. The test case `TestSemesterAPI` has no dependencies and the insert test methods are executed. This is indicated by the generic sequence of calls `setUp(); insertTestMethod(); tearDown()`. The `TestSemesterAPI` is then pushed on the `InvocationStack` indicating that this test case is in use. With the `TestSemesterAPI` test case in use all dependencies for the `TestStudentAPI` test case are fulfilled and the insert test methods are called. The `TestStudentAPI` test case is then pushed

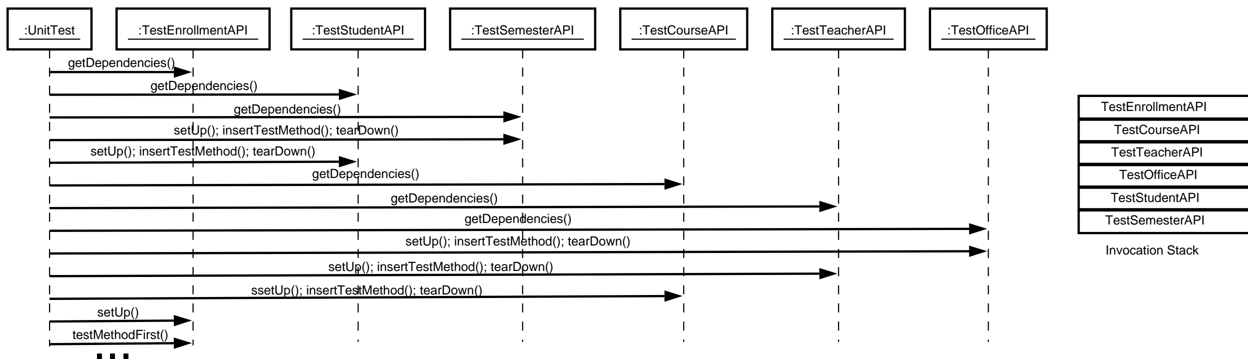

Figure 7: Set Up of Enrollment Test Fixture

on the `InvocationStack`. The `TestCourseAPI` test case still needs to be set up and the `getDependencies` on this test case is called. It returns the `TestSemesterAPI` and `TestTeacherAPI` test cases. Because the `TestSemesterAPI` is already on the stack it does not need to be set up again. Then the `getDependencies` method on the `TestTeacherAPI` test case is called it returns the `TestOfficeAPI` test case. The `getDependencies` method is called on this test case. It has no dependencies and the insert test methods are called and the `TestOfficeAPI` test case is pushed on the stack. Next, all the dependencies for the `TestCourseAPI` are now in use and the insert test methods on the `TestCourseAPI` test case are called. The `TestCourseAPI` is then pushed on the `InvocationStack`. Now all the direct and indirect dependencies for the `TestEnrollmentAPI` test case have been set up. Then this test case is pushed on the stack and all the test methods for this single test case (insert, plain, and delete test methods) can be executed.

The tear down is the reverse of the process described above and left out for brevity.

5 Reusing Test Fixtures

This section describes how the performance of testing large test suites can be improved. The idea is to minimize the number of set up and tear down of test fixtures. As an example, consider a test developer wanting to execute the `TestStudentAPI` and the `TestCourseAPI` test cases. Following the approach described in Section 4, the database is restored to its initial state after completion of the last test method for each test case. This means that the test fixture for the `TestSemesterAPI` test case is set up and torn down twice; once for testing the `TestStudentAPI` test case and once for testing the `TestCourseAPI` test case. However, it is more efficient if the `TestStudentAPI` test case sets up the test fixture for `TestSemesterAPI` and the `TestCourseAPI` tears down the test fixture for `TestSemesterAPI`.

To be able to do such optimization, i.e., reusing test fixtures between test cases, it is necessary to find an ordering of the test cases. In addition, small changes to some of the methods described in Section 4 are needed. The intuition behind the changes is that a test case needs to know how many (but not which) other test cases depends on it and postpone the tear down of itself until the last dependent test case has been tested. This is explained in details in the following.

5.1 Ordering Test Cases

To find an order in which the test cases can be executed, a directed graph representing the test cases and their dependencies are build. Figure 6 shows this graph for the university example. It is checked by the unit-test

framework that the graph is acyclic. If this is the case, the test cases can be ordered and test fixtures can be reused.

The graph in Figure 6 is acyclic. Note that the circular association from `TestTeacherAPI` to `TestTeacherAPI` is not included in the graph because a test case does not depend on itself. Hereafter, a topological sort is performed on the graph. The topological sort defines an order in which the tests cases can be executed. The task of ordering the test cases is handled by the unit-test framework and is transparent to the test developer.

5.2 The Number of Dependencies

The number of dependent test cases is found by building a graph of all the test cases and then count the total number of dependent test cases. The `count` on Figure 6 does this for the university schema example. As an example, the count for the `TestOfficeAPI` is three because the `TestTeacherAPI`, `TestCourseAPI`, and `TestEnrollmentAPI` test cases depend on it.

The unit-test framework builds the graph of test cases before the first test case is executed. It then counts the number of dependent test cases for each node (test case) and stores this in an auxiliary data structure. The next step is to make the necessary changes to the methods on the `UnitTest` class to use these counts.

5.2.1 The `tearDown` Method

The `tearDown` method presented in Section 4.2.6 needs to be changed. The altered method is shown in Listing 7.

```
1 def tearDown(testCase):
2     # Lines 2-9 from previous tearDown method
3     # clean up
4     stack.cleanUpLeftOver()
```

Listing 7: The Changed `tearDown` Method

The comment in line 2 indicates that the code from Listing 5 is reused. The only change to the `tearDown` method is in line 4 where the method `cleanUpLeftOver` on the `InvocationStack` is called. This method removes test cases on the `InvocationStack` that have a count of zero. The `cleanUpLeftOver` method is discussed in details in Section 5.2.3.

5.2.2 The `disUse` Method

The `disUse` method presented in Section 4.2.7 also needs to be changed. The altered method is shown in Listing 8.

```
1 def disUse(testCase):
2     if seen[testCase]: # look if in seen dictionary
3         return
4     seen[testCase] = True # put into dictionary
5     # reduce count
6     testCase.count -= 1
7     if testCase.count > 0:
8         return
9     # Check no left overs
10    stack.cleanUpLeftOver()
11    # Lines 2-6 from previous disUse method
```

Listing 8: The Altered `disUse` Method

In line 2, it is first checked if the test case has previously been seen by the `disUse` method, all seen methods are stored in a dictionary called `seen`. If this is the case, the method is exited in line 3. Otherwise, the test case is marked as seen in line 4. In line 6, the count on the test case is then reduced. Line 7 checks that the count is larger than zero. If this is the case, the tear down is postponed by exiting the `disUse` method in line 8. Otherwise, the stack is cleaned in line 10. Then the method body from Section 4.2.7 is executed. This is indicated by the comment in line 11. Note the dictionary `seen` is reset each time the `run` method on `UnitTest` class is executed.

5.2.3 The `cleanUpLeftOver` Method

An additional method is needed on the `InvocationStack` class to tear down test cases that are no longer in use. The method is called `cleanUpLeftOver` and is shown in Listing 9.

```
1 def cleanUpLeftOver():
2     for testCase on stack:
3       if testCase.count <= 0:
4           UnitTest.disUse(testCase)
```

Listing 9: The `cleanUpLeftOver` Method

The method loops over each of the test case entries in the stack. If the count on the test case is smaller than or equal to zero the `disUse` method is called with this test case as argument. Note that this requires that the `disUse` method on the `UnitTest` class is made public. In Figure 3, the `disUse` method is private.

5.3 Performance Analysis

This section computes how many set up and tear down of test fixtures can be avoided by reusing test fixtures between test cases.

$$noSetup = \sum_{i=1}^{\#nodes} testCase_i.count \quad (1)$$

For the approach listed in Section 4.1 the number of set up (and tear down) of test fixtures is computed by Equation 1. This number is the sum of all the children for each test case. The variable `count` shows the number of children so the number of set ups is the sum of all the `count` variables.

When using the approach in Section 5 all test cases are only set up (and torn down) once when executing a set of test cases. The number of reused set ups is directly related to the number of dependencies between the test cases. The more dependencies the more set ups can be reused. As an example, for the graph shown in Figure 6 5 out of 10 set up and teardown of test fixtures can be reused.

6 Implementation

The unit-test framework is fully implemented using the PL/SQL programming language. The `UnitTest` class contains two `run` methods. One that does reuse test fixtures between test cases and one that does not. The size of the unit-test framework is approximately 4,000 lines of code (including comments). The framework has been tested using the Oracle DBMS.

6.1 Experiences

The unit-test framework has been tested on a medium-size project consisting of approximately 38,000 lines of code and 18,000 lines of test case code. The project contains 83 test cases. None of the test cases overrides the `setUp` or `tearDown` methods from the `TestCase` class. The test cases have between 0 and 8 direct dependencies with an average of 2.3. The number of children varied between 0 and 73 with an average of 7.3 (608 children in total). Based on the number of children and the average number of lines of code in an insert test method we estimate that we save at least 6,000 lines of test case code compared to existing unit-test frameworks where test fixtures have to be build in each test case. This corresponds to a 25% reduction in the size of the test case code.

We found that it is very easy to understand and use the new unit-test framework because it builds on top of existing frameworks such as JUnit and utPLSQL. Getting the first test case up and running requires a larger effort than when using the existing unit-test frameworks because of the dependencies between test cases. However, it becomes easier and easier to add test cases because they can gradually start reusing more and more of the existing test cases.

The benefit of the public variables in the test cases is that there are no *magic values* [13] in the body of the test methods. This makes the test cases more readable and it is easier to change both the value and type of the public variables.

Ripple effects of false negatives cannot be completely avoided. However, it is not a major problem if the test cases are kept synchronized with the code, i.e., following the extreme programming paradigm very closely.

6.2 Performance

All tests are executed on a dedicated server with one Intel Pentium 4, 2.66 GHz CPU and 512 MB of RAM. Each test is executed five times. The smallest and largest numbers are discarded. The average of the last three numbers is reported here.

The test of the six test cases in the university example from Figure 1 takes 1.76 seconds for without reuse of test fixtures between test cases and 1.61 seconds with reuse. The speed up for reusing test fixtures is approximately 8.5%. This means that the reuse of test fixtures is efficient even when there are only 6 test cases in a test suite.

For the medium-size application the test cases take 365.4 seconds to complete without reuse and 222.2 seconds with reuse. This is a 39.2% speed up. A closer inspection reveals that the speed up is partly due to that the test fixtures of a few test cases are quite time consuming and that these test cases are listed as dependencies in several test cases.

7 Conclusion

This paper presents a unit-test framework targeted towards testing database applications. The main idea in the new framework is to allow both test methods and test cases to be dependent. This is in contrast to existing frameworks where each test method must be independent. The benefit of the dependencies is that test fixtures can be reused, which makes it is faster to build and maintain complex test fixtures compared to the existing unit-test framework. In addition, the dependencies between test cases allows for reuse of test fixture between test cases, which is shown to speed up the execution of test suites.

The unit-test framework has been implemented and tested on a medium-size project. A performance study shows that the dependencies between test cases can speed up the execution of large test suites by 40%. In addition, it is estimated that 25% less test code needs to be written because of reuse of test fixtures between test cases.

An interesting direction of future work is to allow multiple users to use the same test cases concurrently. This is not possible in the current design and implementation.

Acknowledgments

We thank Logimatic Software A/S for allowing us to use experiences gained from working for this company in this paper. We thank Brian of CISS, Aalborg University for providing references.

References

[1] Dbunit. `dbunit.sf.net`. As of 2005.09.08.

[2] K. Beck and C. Andres. *Extreme Programming Explained, 2nd Ed.*. Addison-Wesley, ISBN 0321278658, 2004.

[3] K. Beck and E. Gamma. JUnit Cookbook. `junit.sf.net`. As of 2005.09.08.

[4] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weber. A Framework for Testing Database Applications. In *Proceedings of the Int. Symposium on Software Testing and Analysis*, pp. 147–157, 2000.

[5] B. Daou, R. A. Haraty, and N. Mansour. Regression Testing of Database Applications. In *Proceedings of SAC*, pp. 285–289, 2001.

[6] Y. Deng and D. Chays. Testing Database Transactions with AGENDA In *Proceedings of ICSE*, pp. 78–87, 2005.

[7] S. Freeman, T. Mackinnon, and J. Walnes. Mock Roles, Not Objects. In *Proceedings of OOPSLA*, pp. 236–246, 2004.

[8] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proceedings of ICSE*, pp. 645–654, 2004.

[9] P. C. Jorgensen. *Software Testing - A Craftsman's Approach, 2nd Ed.*. CRC Press LLC, ISBN 0849308097, 2002.

[10] JUnit. `www.junit.org`. As of 2005.09.08.

[11] W. E. Lewis and G. Veerapillai. *Software Testing and Continuous Quality Improvement, 2nd Ed.*. Auerbach, ISBN 0849325242, 2004.

[12] R. A. McClure and I. H. Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements In *Proceedings of ICSE*, pp. 88–96, 2005.

[13] S. McConnell. *Code Complete, 2nd Ed.*. Microsoft Press, ISBN 0735619670, 2004

[14] W. Perry. *Effective Methods for Software Testing, 2nd Edition*. Wiley, ISBN 047135418X, 2000.

[15] Python. `www.python.org`. As of 2005.09.08.

[16] utPLSQL. `utplsql.sf.net`. As of 2005.09.08.