A Multi User Unit Test Framework for Testing Database Applications

Project Period: Dat6, February 1st, 2004 – June 1st, 2004

Supervisor: Kristian Torp

Claus Abildgaard Christensen

Steen Gundersborg

Kristian de Linde

Jacob Richard Thomber

A Multi User Unit Test Framework for Testing Database Applications

Claus Abildgaard Christensen, Steen Gundersborg
Kristian de Linde, Jacob Richard Thornber

Department of Computer Science, Aalborg University, Denmark
{cac,eraser,kdl,jrt}@cs.auc.dk

31st May 2004

Abstract

Unit testing is the foundation for building correct and robust applications. Unit testing of database applications differs from traditional unit testing because of persistency and foreign key constraints. Before executing a test case, data associated via foreign keys must exist. Furthermore, after each test case it is necessary to clean up any modifications made to the database. Failure to do so will cause the second unit test to fail.

Existing test frameworks assume independency between test cases. Retaining this assumption when testing database applications, results in a high test fixture. Therefore, a new approach is preferable. We propose that test cases and unit tests should be allowed to depend on each other. This enables us to reuse test fixture and avoid redundant code. Furthermore, as test fixture is only set up and torn down once per unit test, as opposed to once per test case, our framework offers a significant performance improvement compared to existing frameworks.

We have designed and implemented the proposed test framework, minimizing test fixtures and providing automatic cleanup of the database, also in the case of a system crash. We conclude that the test framework is simpler to use than existing frameworks. It allows for faster execution of unit tests, requires less work from developers, and it can be integrated with an existing framework.

1 Introduction

The eXtreme Programming (XP) methodology [2] is becoming increasingly popular among developers [17]. To ensure the correctness of the produced software, an important aspect of XP is the use of automated unit testing. Moreover, to guarantee the correctness of software, even after the addition of new functionality or test cases, regression testing is recommended.

Furthermore, development moves more and more towards using databases, e.g., in ERP systems and in e-commerce. As competition in the software market increases, it is important to ensure correctness of applications. To ensure this, it is preferable to introduce unit testing in this domain.

Integrity constraints bind database tables together. Therefore, it is, in some cases, necessary to insert data in one table before being able to test methods implemented on another table. These integrity constraints make some unit tests dependent on each other.

Current unit test frameworks as JUnit [12] and utPLSQL [9] all share the common assumption, that test cases are independent, as described in the XP methodology. This article settles with this assumption. We argue that, when testing database applications or programs using associations and aggregations, it is beneficial to allow both unit tests and test cases to be dependent.

Testing of database applications differs from testing traditionally software in that most stored procedures have persistent side effects where as testing traditional applications has data stored in main memory and is cleaned up by the test framework at program termination. For example an insert method cannot typically insert the same row twice in a table, without violating the integrity constraints in the database. Therefore, it is typically not possible to execute the same test methods twice, without cleanup between the executions of the tests. This makes regression testing of database applications more difficult as clean up of the database also have to be considered.

The contributions of this article are in the field of testing database applications. First, it presents a new approach for testing database applications that can be combined with existing frameworks. Secondly, it explains the ideas behind a partially automatic test fixture setup and teardown methods. Furthermore, our framework allows for reuse of the text fixture and automatic clean up of the database in case of malfunctioning test cases or a system crash. Thus, the time spent before and after testing is minimized. Hereby, we avoid the repetitive test code that is necessary in existing unit test frameworks. The article gives design and implementation details of the proposed test framework. Finally, we present an extension to the test framework that allows for multi-user support.

The paper is structured as follows. Section 2 looks at the related work done in the field of database testing. In Section 3, we present the test setup environment, i.e., a sample database schema used throughout the article as basis for examples. Section 4 presents the ideas behind the test framework. Section 5 describes the multiple user extension to the framework. It presents how the single user test framework has to be extended in order to support for multiple users testing simultaneously. Section 6 describes how the test fixture is reduced by ordering the unit test, and allowing for concepts to remain set up during several unit tests. In Section 7, the utPLSQL framework is examined. Section 8 explains how the test framework could exploit Oracle specific functionality. In Section 9, implementation details of the single user framework are given. Furthermore, an approach for combining our framework with utPLSQL is given. Section 10 gives a performance analysis of three test approaches. Finally, in 11 the conclusion and future work are given.

2 Related Work

The JUnit testing framework [7, 12] has become the de facto standard for implementing Java unit tests. JUnit supports for the selective retesting, also know as regression testing [8], of a software system that has been modified as a result of a bug. Moreover, no previously working functions is failing as a result of the reparations made. Also, to ensure that newly added functions does not cause previously defined functions to fail. Bottom line, JUnit is a testing framework that defines classes for testing. The main class is known as the *TestCase* class. A developer uses JUnit by extending this class and defines assertions and test methods denoted by the prefix test. These test methods are responsible for the calls to the application that the developer wants to test. If any of the assertions fails, it is displayed. JUnit's test fixture is a common set of test data and objects, which is shared by all tests. It is instantiated by the means of two special methods, the setUp and the tearDown methods. Once the test fixture has been setup, JUnit allows for easily testing the application. The test case, or a collection of test cases also known as a test suite, can easily be run.

The DbUnit testing framework [1] is a JUnit [12] extension that is aimed specifically at database-driven applications. Before a test case 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. Therefore, test cases that fail cannot corrupt the database. Several best practices are proposed, e.g., all testing should be conducted on a dedicated test server. As with all frameworks build on the JUnit framework, DbUnit also make the assumption that test cases are independent.

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 application different from testing applications that do not store data persistently. They propose a tool-set architecture consisting of four tools, including a prototype implementation of the first tool. This tool populates a given database with data satisfying integrity constraints.

Books on traditional software testing [2, 11, 16] provide no specific handling of testing of database applications. In [13], Lewis does consider testing of database applications. However, Lewis only considers how to test if integrity constraints are fulfilled, e.g., how to see if all primary keys are valid, something most RDBMSs do automatically.

In [2], Beck advocate for automatic testing, however, as with JUnit [12] an important assumption is that all test cases are independent. In this article we claim that when testing database applications it is beneficial to let test cases depend on each other.

The utPLSQL framework described in [9] is an Oracle PL/SQL unit-testing framework. The framework is modeled on the JUnit [12] framework. The utPLSQL framework enables testing of each package automatically. However, utPLSQL is developed for testing PL/SQL applications, where our framework focuses on testing the interaction between the application and the database.

In [10], Daou et. al categorize the problems that SQL inflicts on regression testing into three groups: control dependencies, data flow dependencies, and component dependencies. They suggest a method for performing regression testing on database applications. According to Daou et. al, programming with SQL and database triggers often involves exception handling, that is many exceptions have to be caught and handled. As traditional control flow modeling does not contain a way of modeling exception handling, Daou et. al propose a way of doing this. Each statement is represented as a node, each node has two exits one for successful endings the other for unhandled exceptions. If no exception handling is available, all exception links will be linked to an unhandled exception node. A specific exception is modeled by a predicate node, which has two exits, the first links to the start node of the exception block, the other to the next handled exception. The regression test method is split into two parts, part one consists of modification detection, and impact analysis, i.e., estimating what will be affected if a change in the software is made. As a change in one component often affects other components, because the state of the database is altered, it is necessary to find all dependencies between components of the application in order to determine which components should be tested. A component is marked as modified and placed in a list of modified components, called the *component firewall*, if either itself has been modified or deleted, or if it is dependent on a modified or deleted component. All test cases that involves components in the firewall are selected for regression testing. This article differs from [10] by Daou et. al in that we consider unit testing, whereas Daou et. al strictly considers regression testing, i.e., it is not discussed how a single test case should be set up.

3 The Test Setup

This section introduces a small database schema that will be used as an example throughout the article. Hereafter, the used terminology is explained. The schema depicted in Figure 1 is used as an example schema when describing the test framework.

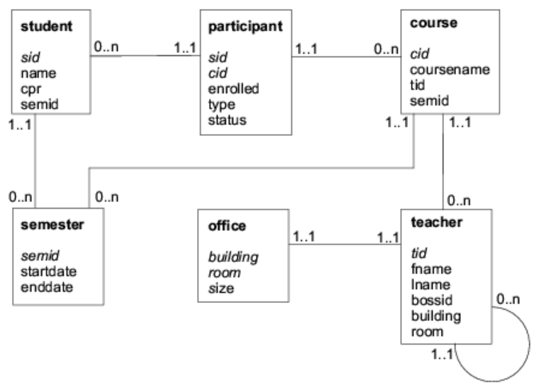

Figure 1: The University Example Schema. Attributes in italics denotes the primary key of a table

The example schema represents a small university. The university has a number of students, information on these is stored in the *student* table. The *course* table stores the courseidentifier *(cid)*, the course name and the foreign key teacher identifier *(tid)*. The *participant* table is a relationship between the *student* and the *course* tables. The primary key of the *participant* table are the *sid* and *cid* attributes, which are also foreign keys referencing the *student* and the *course* tables, respectively. In addition, the *participant* has three additional attributes. First, *enrolled* that describes when the student is enrolled to the course. Secondly, a *type* attribute that describes if the student is following the course in the traditional manner or if he is distant learning. Finally, the *status* attribute describes if the student is active or inactive in the course. As depicted in the ER-diagram in Figure 1, a student can follow zero or many courses and a course can have zero or many students. Recall the foreign key *tid* in the *course* table, this key references the *teacher* table. Each course has exactly one teacher associated. A teacher can teach zero or many courses. A teacher can act as a supervisor for other teachers. This is modeled by the self-reference and the *bossid* attribute on the *teacher* table. Finally, each teacher is situated in exactly one office. This is modelled through the composite foreign key consisting of *building* and *room*, which references the *office* table. The *semester* table is used to connect students and courses to specific semesters, e.g., an introductory semester. The primary key of the table is the *semid* attribute, which is used as a foreign key in the *student* and *course* tables. Moreover, the *semester* table has two date attributes indicating the start and end date of a semester.

The terminology used in [5] is also used in this article. Meaning that a *concept* is equivalent to a class in an object-oriented language or a table in the database. A *record* corresponds to an instance of a *concept* or a row of a database table. Finally, a *record id* is a unique identifier of a *record*, corresponding to the primary key of a database table.

4 Test Framework

In the following section a framework for unit testing database applications is designed. It aims at correctness testing and not performance testing. Furthermore, the framework aims at minimizing test fixture [3], also known as the amount of work needed before after doing actual testing. The framework consists of a set of methods associated with each concept and a persistent stack containing the concepts active during each step of the testing. These will be described individually. Hereafter, a set of conventions are described. These conventions are necessary for the developer to follow for the framework to work. Finally, examples of the test framework are presented.

4.1 The idea of the framework

Traditionally, when testing code, test cases are run, errors found are corrected and the test cases are rerun. Moreover, they might be executed every night to ensure any modifications made to the code during the day do not violate any test cases. Several tools like JUnit [12] and utPLSQL [9] exist that are well suited for this task – when persistency is not considered.

Generally, test cases are grouped into unit tests, each testing a specific part of the given application. The entire collection of unit tests is again grouped together, forming a test suite for testing the entire application. Using a part of the schema depicted in Figure 1, Figure 2(a) shows an example of a test suite comprised of three unit tests. The example shows an execution of the test suite, running the three unit tests. It is important to note that the unit tests can be run independently, and furthermore, that each test case can be run independent from other test cases. This is due to the fact that each test case only uses main memory data structures. When the test is completed the data is simply discarded because the program exits. At a higher level of abstraction this can be viewed as testing functions, i.e., on a given input, compare the output with the expected output.

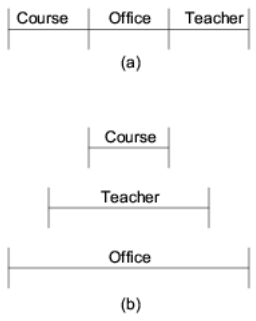

Figure 2: Difference from standard unit testing to unit testing of database applications. Part A is the standard way of testing. Part B test of database applications, where test cases are dependent of each other, i.e., to test *course* relies on tests on *teacher* which requires *office* to be tested first.

When testing code that uses data stored persistently in a database things are more difficult because the outcome of a test case depends on the state of the database. Furthermore, each test case may have persistent side effects, altering the state of the database. This can be viewed as testing procedures. Since there is no output it is testing for side effects, in contrast to existing test frameworks.

Testing database applications poses two main problems: Persistency and foreign keys. The persistent side effect of a unit test is already an issue when testing an independent concept. Since connected concepts require several concepts to be set up in order to test a single concept, foreign keys pose a problem when testing these. In the following, solutions to both problems are proposed.

4.1.1 Persistency

When testing database applications data used in the test cases are stored persistently in the database. Hence, without proper cleanup after running a test case, a second execution might not yield the same result or fail. Thus, to preserve the property of test cases being independent, proper cleanup must be performed after each test case. However, since this imposes a high performance overhead we suggest another approach, where we allow test cases within a unit test to depend on each other, making room for reuse of code. This implies that test cases no longer are independent since any modifications made by one test case affects all succeeding test cases. Hence, in general, single test cases cannot be run individually but only as a part of executing a unit test. This implies that cleanup is only necessary after running the unit test.

To be able to do the cleanup of the database after running a unit test, it is necessary to setup some rules for what is allowed when modifying data and keep track of what needs to be undone. These rules are the following.

- Primary keys or any other candidate keys used in the unit test must be public.
- It is only allowed to change the data that the test itself has inserted. This includes primary keys and candidate keys.

If these rules are followed then the framework ensures that it can clean up, even if the test crashes.

4.1.2 Foreign keys

In database schemas there can be dependencies among data. Concepts may rely on data from other concepts, posing a problem when testing. Proper setup is required such that all needed concepts valid data.

Traditional testing methods dictate that unit tests must be self containing. Using this idea, imagine a unit test for the *teacher* concept. Figure 3 shows that concepts are associated to each other via foreign keys between tables, and unit test for these concepts are therefore also associated. Since the concept associated with the unit test for *teacher* depends on the *office* concept. Thus, the unit test must supply valid data for the table associated with the *office* concept to remain independent. Now imagine a unit test for the *course* concept. The *course* concept depends on the *teacher* concept and on the *office* concept via transitivity. Therefore, the unit test for the *course* concept must supply valid data, not only for the table associated with the *teacher* concept, but also for the table connected with the *office* concept. Hence, repetitive code for creating valid data for table connected to the *office* concept exists in the unit tests for both the *teacher* – and the *course* concept.

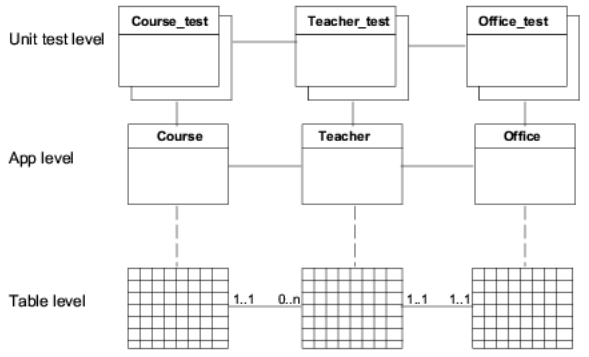

Figure 3: The coherence between tables, concepts and unit tests.

This is only a simple example schema. Real schemas are much more complex, thus the amount of redundant code becomes very high. Similarly, the problem with repetitive occurs when cleaning up. Figure 2(b) shows a test suite comprising of three unit tests. This figure illustrates how the unit tests in Figure 3 depends on each other. Using our approach, reconsider the unit test for the *teacher* concept. Instead of supplying valid data for the *office* concept, we reuse the code already in a unit test for the *office* concept. When testing the *course* concept, the code from a unit test for *teacher* is reused, which again reuses code from a unit test for *office*. This way, a unit test must only supply valid data for the concept it is testing, reusing code from unit tests belonging to the concepts, the current concept is directly related to. Again, the same approach is applied when cleaning up. Compared to the traditional approach this gives a significantly lower test fixture.

4.2 Extending the idea

The idea of the test framework can be utilized in other contexts than database applications. Testing programs with associations or aggregations, e.g., in the Java programming language, pose some of the same problems. Thinking of references between concepts as foreign-key dependencies, there is a major difference. Associations and aggregations result in main-memory structures only. Hence, cleanup is not necessary between test runs since data placed in main memory is removed at unit test termination. In other words, persistency is not an issue. However, populating main memory with references to valid concepts prior to running a given unit test is conceptually the same as supplying valid data for tables related to concepts needed for a given unit test. Thus, the example given in Section 4.1.2 is just as relevant to testing associations and aggregations as testing database applications. However, since they only pose a subset of the problems associated with testing database applications, we choose to focus only on the latter.

4.3 The Framework

The framework consists of a set of methods, public constants and a persistent stack: A setup method, a teardown method, a run method, a group of test methods, public constants and finally a table containing the active concepts, i.e., the concepts needed for the current test. This is depicted in Figure 4. The constructs will now be described in further detail.

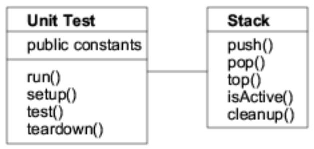

Figure 4: An UML class diagram of the framework.

4.4 Framework Constructs

In the following, each of the methods supplied by the framework will be described.

Setup: The setup method is used to ensure that there is a valid foundation for testing, i.e., the database is put in a known state. Recall Figure 3, in order to test the *course* concept it is necessary to have data in the *teacher* concept. The purpose of the setup method is to ensure that for the given concept there is valid data in the table related to it. Since tables can have foreign keys to other tables these also need to contain valid data. This is achieved by having the setup method call the setup method on all concepts the current concept is related to via foreign key constraints.

If the setup method is on the concept currently being tested then it only calls the setup on other connected concepts. Thus, it does not insert any data. However, if the setup is not on the concept being tested then other needed setup methods are called and data is inserted. This is done because the inserts is a part of the tests themselves. The setup method is schema dependent.

Teardown: The teardown method is the reverse of the setup method. It ensures that modifications made to the table related to a given concept during setup and testing is undone, leaving the table in the state it was in before testing begun. As the setup method, the teardown method reuses teardown code from other unit tests. When several concepts have been setup for a given test, teardown must be called in the reverse order, e.g., the last concept that is setup is the first to be torn down. This is also illustrated in Figure 2b. Here, *course* was the last to be setup, thus it is the first to be torn down. The teardown method is schema dependent.

Run: The run method is the method called for executing the unit test itself. The run method ensures that there are valid data in all required tables by calling the setup method. The test itself is done by calling all the test methods, in the correct order. Finally, the cleaning of the database is done by calling the teardown method. The run method is schema independent.

4.4.1 Active concepts

A persistent stack of the concepts currently active is maintained. This information ensures the robustness of the framework, since in case of a system breakdown any modifications made to the database can be undone using the cleanup method, when the system again is up and running. Whenever the setup method have been called on a concept, this concept is pushed onto the stack using the push method. As soon as the teardown method is called on a concept, this concept is popped off the stack, via the pop method. However, the concept is only popped if it is on top of the stack, and this is checked by the top method. To determine if a concept already is active we allow looking at all the elements of the stack, not being able to modify them, however. The isActive method offers this functionality. The methods using the persistent stack are executed in an autonomous transaction. The stack is schema independent.

4.5 Conventions

This section describes the conventions that the developer has to follow in order for the test framework to work.

4.5.1 Public constants

The public constants are used by the test framework to refer to primary keys. These are not only on the concept which is currently being tested, but also on the concepts that are connected via foreign keys. There are three types of these constants.

1. The first type is mandatory and used to insert data. Thereby, they are available as foreign keys when inserting in other tables. These primary keys are valid and used in the database.
2. The second type is optional and used to enable modifications of primary keys and insertion of additional data in the actual test methods between the setup and the teardown phases of the test.
3. The third type is optional and consists of primary keys that are valid and does not exist in the database. These are not to be used for insertion, but for testing for non-existing rows.

The public constants are schema dependent and are created by the developer.

4.5.2 Ordering of Tests

It is necessary to enforce a bottom-up approach when testing concepts. The developer has to start by testing concepts that other concept does not reference. In the university example, this means that the developer should start by testing either the *semester* or the *office* concepts. When the *semester* concept has been tested, the developer can continue on to the *student* concept and so forth.

As test cases are dependent, it is also important that the developer consider this, when designing a unit test. For example, it would be natural to insert data in a concept, before testing, e.g., an exist method.

4.5.3 Naming conventions

Packages and methods that are to be tested, are by default prefixed with test_. Then, the test framework will automatically execute these tests. Naturally, the default prefix can be redefined by the user. Without a prefix, the framework would execute every method, which often would not be desirable.

4.6 Reuse of code

The framework reuses code for setup and teardown to minimize the test fixture. The following example sequence diagram in Figure 5 shows the process of setting up the database for a test and the following sequence diagram in Figure 6 shows the process of cleaning up the database after the test. The example diagrams use a subsection of the schema shown in Figure 1. The subsection of the schema only contains the tables *student*, *participants*, *course* and *semester*.

4.6.1 Setting up the database

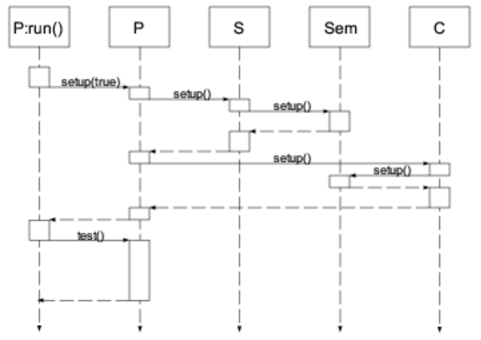

Figure 5: Sequence diagram showing the process of setting up the database for a test run. The abbreviations P: participants unit test, S: student unit test, Sem: semester unit test and C: course unit test.

The example sequence diagram in Figure 5 shows the process of setting up the database for a unit test on the *participants* concept. The test is initiated by calling the run method on the *participants* unit test. The run method ensures that all the needed methods are called in the correct order. Hereafter, the run method calls the setup method on the *participants* unit test. The setup method is called with the boolean value true that tells the setup method that it is this unit test being executed now. The setup knows from the data dictionary [14] that data is needed in both the *student* table and the *course* table. Therefore, setup calls the setup method on the student unit test. However, setup on *student* needs data from the *semester* table. Therefore, it calls the setup method on the *semester* unit test and the first data is inserted. The setup method from the *semester* unit test returns after inserting the data and setup on *student* is now able to insert data and return. As the *participants* table dependens on the *course* table, setup is still not ready to insert data, so setup on the *course* unit test is called. The *course* setup needs data from the *semester* table and, therefore, calls the setup method on *semester*. The *semester* setup returns without doing anything since data is already inserted. The *course* setup now inserts data and returns. The *participants* setup method is now finished and returns to the run method which initiates the unit test.

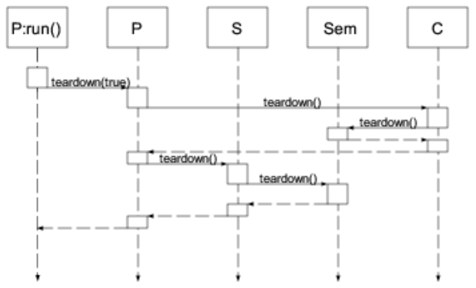

Figure 6: Sequence diagram showing the process of cleaning up the database after a test run. The abbreviations P: participants unit test, S: student unit test, Sem: semester unit test and C: course unit test.

4.6.2 Cleaning up the database

The example sequence diagram in Figure 6 shows the process of cleaning up the database after running the unit test on the *participants* concept. The cleanup is initiated by the run method calling the teardown method on the *participants* unit test. The teardown method removes all test data from the *participants* table and calls the teardown method on the *course* unit test. The *course* teardown removes the test data from the *course* table and calls the teardown method on the *semester* unit test. The teardown on *semester* returns without removing any data, because its data is still in use and the teardown on *course* returns. The *participants* teardown now calls the teardown method on the *student* unit test. The *student* teardown removes the test data for the *student* table and calls the teardown method on the *semester* unit test. The *semester* teardown removes test data and returns to the *student* teardown which returns to the *participants* teardown method which returns. This concludes the cleanup of the database.

4.7 Examples

In this section we give two examples of how the test framework functions. Initially, we consider a test involving only one concept. Hereafter, we consider a case involving two concepts.

4.7.1 Testing an independent concept

In this section, we consider a scenario where we wish to test a single concept, in this case the *office* concept from the small university scheme depicted in Figure 1. The test is initiated by calling the run method. The run method starts by calling the setup method. The run method is shown in Listing 1.

```
procedure run is
begin
```

```
  setup(true);
  test ();
  teardown();
end;
```

Listing 1: The run method when testing on a single concept.

As the *office* concept has no dependencies to other concepts, no setup of any other concepts are necessary. When only one concept is tested, the setup method does not insert data, this is left to the developer. Instead, the setup method initializes the public constants that are available to the developer. These constants include those that are to be used as primary keys and primary keys that the developer are allowed to update, as described in Section 4.5.1. Hereafter, the setup method returns to the run method, which calls the test methods. The setup method is shown in Listing 2.

```
1  -- package header
2    building constant          := 'E4';
3    room constant              := '110';
4    building_update constant := 'E3';
5    room_update constant     := '210';
6
7  -- package body
8  procedure setup(self_test boolean) is
9  begin
10   if ( not stack.is_active(''office '')) then
11     stack.push(''office '');
12
13     if ( not self_test ) then
14       insert (building, room);
15     end if;
16   end if;
17 end;
```

Listing 2: The setup method when testing on a single concept.

In lines 2-5 of Listing 2 the public constants are set up. Note that in PL/SQL this is done the header of a package. In line 10, we check if the *office* concept is already on the persistent stack. If this is not the case, the concept is set up. In line 11, we push the office concept onto the persistent stack. This enables cleanup of the database, should an error occur. As this is a test of the *office* concept, the setup method does not insert any records into the *office* table. The developer will have to do this manually in the test methods. Here it is important to note that the developer should use the defined constants.

An example of a test method written in pseudo code is shown in Listing 3.

```
1  procedure test
2  begin
3    insert (building, room);
4    assert.eq('Building not found!',
5                exist (building,room), TRUE);
6
7    update(building, room
8           building_update, room_update);
9    assert.eq('Building_update not found!',
10               exist (building_update,
11                        room_update), TRUE);
12 end;
```

Listing 3: The test method for testing the *office* concept.

In line 3 of Listing 3 a record is inserted into the *office* table, and the public constants defined in the setup method is used. Immediately after the insertion, in lines 4 and 5, an exist method is used to check if the new record in fact is inserted. Should this not be the case, the assert method will return the error string 'Bulding not found'. In line 7 we update the primary key of the record inserted in line 3. Hereafter, the exist method and the assert.eq method is used to check if the record has been updated. Again an error string is returned if the record does not exist.

As shown in Listing 2, the setup method calls the teardown method immediately after the test method completes its execution. The teardown method is shown in Listing 4.

```
1 procedure teardown
2 begin
3   if ( stack.top('office ')) then
4     delete (building_update, room_update);
5     delete (building, room);
6     commit();
7     stack.pop();
8   end if;
9 end;
```

Listing 4: The teardown method for testing the *office* concept.

First, the teardown method checks that the *office* concept is on top of the stack, in line 3. If not, other concepts still depend on it, and the teardown method does nothing. If the *office* concept is on top of the stack, all records inserted into the *office* concept are removed, in lines 4 and 5. Please note that the transaction is commited in line 6. This is done to ensure that any concepts that have not been cleaned up are removed from the stack. Finally, line 7 pops the *office* concept of the persistent stack as it is no longer in use, and the *office* concept has been properly restored to it's default state.

In the next section we consider unit test, where we are to test the *teacher* concept. The interesting thing here is that in order to test the *teacher* concept we need data from the *office* concept.

4.7.2 Testing connected concepts

In this section we show an example of a unit test on the *teacher* concept. As mentioned previously, the interesting thing here is that the *teacher* concept is connected through foreign keys to the *office* concept. The unit test initiates in the same way as described in the previous section, by calling the run method. The difference lies in the setup method and the teardown method. The setup method is depicted in Listing 5.

```
1  -- package header
2    tid  constant          := 1;
3    tid_update constant := 10;
4
5  -- package body
6  procedure setup(self_test boolean) is
7  begin
8    if ( not stack.is_active('teacher')) then
9      office .setup(false );
10     stack.push('teacher');
11
12     if ( not self_test ) then
13       insert (tid,office .building,office .room);
14     end if;
15   end if;
16 end;
```

Listing 5: The setup method used to test the *teacher* concept.

As in the previous section the public constants are defined in the package header. The first things that happen in the setup method is that we check if the *teacher* concept is active. If this is not the case, the setup method on the *office* concept is called with the boolean value false. The false value indicates that the setup method on *office* should load data into the office table, as another concept is using it for testing purposes. As described in the previous section, the *office* concept is also pushed onto the persistent stack, indicating that the concept is now active and should be restored to its default state upon completion of the unit test. Hereafter, in line 10, the *teacher* interface is pushed onto the persistent stack. All concepts are now properly set up and, we are ready to run the test methods, that is shown in Listing 6.

```
1  procedure test
2  begin
3    insert (tid, office .building, office .room);
4    assert.eq('Teacher not found!',
5                exist (tid, office .building,
6                office .room), TRUE);
7  end;
```

Listing 6: An example test method on the *teacher* concept.

In line 3, a record is inserted into the *teacher* concept. Note here that some attributes have been omitted for brevity – naturally they should be included in a real test case. The values used in the insertion are the publically defined tid, and the public constant building and room from the *office* concept. In lines 4 and 5, the exist and the assert.eq methods are used to check if the record is in fact inserted. Should this not be the case an error string is printed. After completion of the test, control is returned to the run method that calls the teardown method on the *teacher* concept. The teardown method is shown in Listing 7.

```
1  procedure teardown
2  begin
3    if (stack.top('teacher')) then
4      delete(tid_update);
5      delete(tid);
6      commit();
7      stack.pop();
8      office .teardown(false);
9    end if;
10 end;
```

Listing 7: The teardown method of the *teacher* concept.

First, the teardown method checks that the *teacher* concept is on top of the stack, in line 3. If not, the teardown method does nothing. If the *teacher* concept is on top of the stack, all records inserted into the *teacher* concept are removed, in lines 4 and 5. Note that the transaction is committed in line 6. This is done to ensure that any concepts that have not been cleaned up are removed from the stack. In line 7, the *teacher* concept is popped of the persistent stack as it is no longer in use, and has been properly restored to the default state. In line 8, the teardown method on the *office* concept is called. When that method has finished executing, all affected concepts have been restored and all elements have been popped off the stack.

4.7.3 Summary

The above examples show how to use the test framework. There are some advantages of using the test framework. The framework provides a systematic way of setting up the database for a unit test and an automatic cleanup after testing. Furthermore, the framework is able to cleanup test data when exposed to system crashes. Additionally, the framework has a little test fixture. However, these advantages come at a cost that comes in the form of some requirements to the user of the test framework. The user must publish primary keys and call methods in the correct order. Furthermore, the user may only change data that the test itself had inserted and only modify specific primary keys. Furthermore, the framework requires that the schema must exist.

5 Multi User Framework

The framework is extended to support multiple users. In the following section, the challenges on the multiple user extension are discussed. First, an overview of the proposed solution is given. Then, limitations of that solution are discussed. Hereafter, details of the extension are presented. Finally, an example of two users testing simultaneously is given.

5.1 Challenges

When extending the framework to support multiple users the following challenges arise.

Concurrency Several users could try to test the same concept at the same time. This is a problem because both users would try to insert data using the same primary keys, which would lead to primary key violations.

Dependent concepts Users could try to test concepts that depend on each other. For example, the *student* and the *course* concepts can be tested simultaneously. However, the *student* and the *semester* cannot be tested simultaneously, because the test on the *semester* concept might change the data of the *semester* table and thereby corrupting the *student* test.

Clean up Cleaning up after a unit test is more difficult, since several concepts being tested can share data inserted into other concepts. Hence, when one unit test has finished, data inserted into other concepts might still be used by other unit tests. Therefore, it cannot be removed yet. As an example, consider unit testing both the *student* and the *course* concepts. If the unit test of *student* is completed, data inserted into the *semester* concept cannot be removed until after the unit test of the *course* concept also is completed.

As a solution to the challenges above, we propose three extensions. The solution to the first challenge is not to allow several users to test the same concept at the same time. To solve the second challenge, we propose that users cannot test concepts that depend on each other, at the same time. As a solution to the third challenge, we propose that users must supply the maximum running time of a unit test.

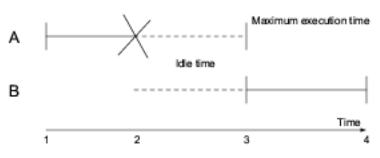

Figure 7: Illustration of the worst case idle time scenario with two testers.

Figure 7 shows a possible scenario with two testers, A and B, testing two dependent concepts. At time 1, A initiates a unit test. A expects his unit test to terminate no later than at time 3. At time 2, A's unit test crashes. Since A's unit test has crashed and A does not clean up the database afterwards, B cannot start testing before time 3 even though A's unit test is no longer executing. B has to wait unnecessarily as the maximum execution time deviates from the actual test time. The larger this deviation is the longer B has to wait unnecessarily. Thus, it is important not to over-estimate the maximum execution time. At time 4, B's unit test is finished.

The following section describes the proposed extensions needed to the test framework and their consequences in detail.

5.2 The Extensions

The underlying table for the stack is extended with four columns. The first is the user name of the user that is testing a concept. This enables us to clean up after a unit test from one user, while leaving unit test from other users unaffected. This implies that if the data inserted by a unit test is shared between several unit tests, the data inserted is removed when the last unit test is finished. The second column is a time stamp describing when data inserted by the setup method for the given concept is invalidated. This ensures that if a test crashes and the user does not initiate a cleanup afterwards, other tests must not wait indefinitely for the crashed test to finish. This implies that users must provide reasonable time stamps. This is the only additional convention needed when going from the single-user framework to the multi-user framework. The third column describes if the concept itself is being tested, in the following this is denoted as a self test. This is necessary because if it is the concept itself that is being tested, it cannot be used by any other tests, since data in the underlying table may be modified. The last column describes if the concept is in the process of setting up, or it has been setup already. This ensures that a test setting up will not be aborted on account of one of the concepts which are being setup. For example, when testing the *participant* concept the first row inserted on the stack will indicate that the *participant* concept is being setup for a self test by this user. The time stamp will indicate the maximum time boundary of the test. When this test reaches the setup of the *student* concept another row is put on the stack. This row indicates that the *student* concept is being setup for a test and will prevent other users from starting testing the *student* concept itself, and thereby aborting the *participant* test.

The cleanup method is extended to remove all data inserted by the current user, by calling the teardown method on all concepts the user has on the stack. Furthermore, any data invalidated by the time stamp, for any user, is removed as well.

The top method is extended to consider only the current user.

The isActive method is modified to return the status of a concept, instead of a boolean value. This makes it possible to differentiate between several states of a concept. The first state is that the concept is not on the stack. The second state is that the concept currently is being setup. The third state is that the concept is setup already. The last state is that the concept is undergoing test.

The push method is extended such that it takes the user name of the given user, a time stamp, if the concept is being used or tested itself, and if the concept is being setup or has been already. These parameters correspond to the new attributes of the stack table.

The only extension needed to the pop method is the addition of the user name, such that the stack not necessarily pops the top element, but the top element for that user.

The setup method needs to be modified, to avoid problems with two tests wanting to setup the same concept for a self test, and one test trying to setup a concept another wants to test, concurrently. When accessing the stack to check the current state of the given concept, a lock on the stack table must be established beforehand. This is because, depending on the state of the concept, a new row is pushed on the stack, or another row is altered. This must happen without any other users accessing the stack. Otherwise, it could be possible for a concept to be tested itself and be used by other unit tests at the same time. If the concept depends on other concepts, it is marked as being setup on the stack, and the actual insertion of data is deferred until the dependant concepts have been setup. If the concept does not depend on other concepts, data is inserted immediately. It should be noted that the lock on the stack table is held while data is inserted in the concept.

The teardown method needs modification as well. When tearing down, inserted data should not be removed if other unit tests are using the concept. As with the setup method, the stack is locked before accessing it and until after the inserted data has been removed.

The run method is only extended by calling the cleanup method before starting the test.

5.3 Testing Concepts Simultaneously

The example sequence diagram in Figure 8 shows the process of two users testing the *student* concept and the *course* concept simultaneously. The example diagram uses a subsection of the schema shown in Figure 1. The subsection of the schema only contains the concepts *student*, *course*, and *semester*.

The two tests are initiated by two different users that independently execute the run methods of the *student* unit test and the *course* unit test. First, the *student* unit test is initiated by when the run method is executed. Then, the run method of the *student* unit test cleans up the database by removing any data owned by the user and any data containing

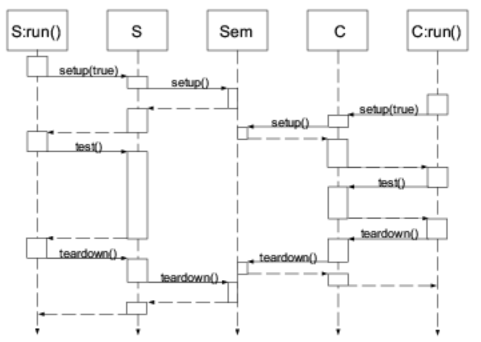

Figure 8: Sequence diagram showing the process of testing the *student* concept and the *course* concept simultaneously. The abbreviations S: *student* unit test, Sem: *semester* unit test and C: *course* unit test.

an invalid time stamp. After the clean up, the setup method of the *student* unit test is called. The setup method checks that the concept is not in use. Then, a row is pushed onto the stack, declaring that the concept is being setup for a self test. Hereafter, the setup of the *semester* unit test is called. The *semester* setup checks that it is not in use and inserts the test data into the concept. Then, a row is pushed onto the stack. This row indicates that the first user has setup the *semester* concept, and that it is not for a self test. Afterwards, the *semester* setup returns. Simultaneously, the second unit test is initiated, and the run method of the *course* unit test cleans up the database and calls the setup of the *course* unit test. The setup method of the *course* unit test puts a row onto the stack, indicating that the concept is being setup for a self test. Then, the setup method of the *semester* unit test is called. The setup method of the *semester* unit test checks the stack. Because the *semester* concept is already setup, the setup method just puts a row onto the stack indicating that the concept is also used in the second unit test, and the setup method returns. At the same time, the first unit test is at the setup of the *student* unit test. Here, the row on the stack is updated so it indicates that it is setup for a self test. Then, the row is moved to the top of the stack and the setup method returns to the run method. The run method executes the test cases from the *student* unit test. Meanwhile, the second test is in the setup method of the *course* unit test. Here, the row on the stack is updated such that it is moved to the top of the stack, and the setup method returns. The run method of the *course* unit test executes the test cases of the *course* unit test. After they have finished, the teardown method of the *course* unit test is called. The teardown methods checks the stack and removes the test data from the concept.

Then, the teardown method of the *semester* unit test is called. The teardown method of the *semester* unit test checks the stack. Since another unit test is using the concept, only the row on the stack used in this unit test is removed. Hereafter, the teardown method returns. The teardown of the *course* unit test returns, and the second unit test is over. Simultaneously, the first unit test has finished its test cases and calls the teardown method of the *student* unit test. The teardown method removes all the test data from the concept and the row on the stack. Then, the teardown method of the *semester* unit test is called. Now, no other unit tests are using the *semester* concept, so both the row on the stack and the test data is removed, and the teardown method returns. The teardown method of the *student* unit test returns, and the first unit test is finished.

5.4 Generalization

The stack is locked during all access to it and all insertion of test data. This ensures that only one change can be made to the system at the time. Therefore, it is impossible to set up more than one concept at the time. Thus, two users cannot test the same concept at the same time, since one user will always be the first to lock the stack and hereby be able to set up a concept. The same argument applies when multiple users tries to test the same concept. Furthermore, this applies for dependent concepts as well. Hence, the idea of the framework holds when scaling from two to several users.

5.5 Limitations

This section describes the limitations of the multi-user framework. First, users cannot test the same concept at the same time. Furthermore, dependent concepts cannot be tested by more than one user at a time. This is because that the two unit tests would try to insert the same data twice, which will cause a primary key violation. Secondly, it is important to provide reasonable estimates of the maximum execution time of the unit tests. If the developer makes too optimistic an estimate, other unit tests will remove his unit test data while the unit test is still executing. On the other hand, a too pessimistic estimate will cause other unit tests to wait unnecessarily in case of a unit test crash.

6 Performance Optimization of the Test Framework

Time spent waiting for tests to complete reduces the amount of time the developer can devote to other aspects of the development.

This section describes how the performance of the test framework can be improved when testing large test suites. This is done by minimizing the number of calls to the setup and teardown methods. For example, consider a scenario where a developer want to test the *student* concept and hereafter the *participant* concept. Following the approach described in Section 4.7, all concepts would be restored to their initial state, by calling the teardown method on the *student* concept, after completion of the unit test on the *student* concept. Hereafter, the setup method on the *participant* concept would start by loading data, first into the *semester* concept and hereafter the *student* concept. In this scenario it would be beneficial if only the *student* concept is torn down and setup, while the *semester* concept remains unchanged. In this way, one call to a setup and one call to a teardown method are removed, thus the time spend setting up the test fixture is minimized. To optimize the testing process it is necessary to find an ordering of the unit test. The following section describes how this is done.

6.1 Ordering Unit Tests

To find the order in which the test suites should be executed, we create a directed graph representing the concepts and their dependencies from the underlying schema. The graph is created by querying the Oracle data dictionary. Then we check if the graph is acyclic. If this is the case, the unit test can be ordered. Otherwise, no optimizations can be performed. A directed acyclic graph (DAG) representation of the database example schema from Figure 1 is shown in Figure 9. The concepts are vertices and the foreign keys between them are edges.

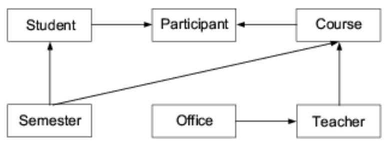

Figure 9: Directed acyclic graph representing the university schema of Figure 1.

Hereafter, a topological sort [6] is performed on this graph. Topological sort only works on DAGs, therefore, we ignore self-references as the reference from *teacher* to *teacher*. The result of a topological sort on Figure 9 could look as depicted in Figure 10.

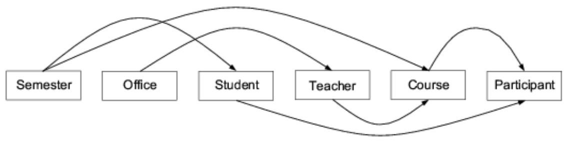

Figure 10: Topological sort of Figure 9.

The topological sort defines the order in which the unit tests in the test suite should be executed. For instance, we see from Figure 10 that the *semester* concept is the first[1] concept to be tested.

Furthermore, the teardown methods are simplified such that they only clean up the concept on which they are implemented. For instance, the teardown method on the *student* concept cleans up the *student* interface and terminates. In the former implementation the teardown method of the *student* concept, would call the teardown method on the *semester* concept.

As mentioned, the *semester* concept is the first to be tested. On completion of the test, the teardown method cleans up the concept. Hereafter, the *office* concept is to be tested. Upon completion of the test, the *office* concept is cleaned up. Then, the *student* concept is to be tested. The setup method automatically sets up the *semester* concept. The test on *student* concept is then performed. Hereafter, the modified teardown method cleans the *student* concept, but leaves the *semester* concept active. The *teacher* concept requires the *office* concept, thus this is setup. Upon completion of the *teacher* unit test, the *office* concept remains setup, and only the *teacher* concept is torn down. Then, the *course* concept is tested. The *course* concept sets up the *teacher* semester. The *office* semester is already active, and is therefore not setup again. Finally, the test continues by setting up the *participant* concept. The setup method on the *participant* concept sets up the *student* concept. Note that the *semester* concept already was setup, from when the *student* concept began testing.

The task of ordering the unit test is handled by the test framework without user involvement.

Please note, that when a concept is tested, the unit test data is setup, test cases are executed, and the data is torn down again. When the test data for this concept is used by other unit tests, the test data is setup but teardown is deferred until completion of the test suite. Therefore, subsequent calls to the setup method simply returns because the concept already has been setup.

[1]Please note that a topological sort on the university schema could also yield a result where the *office* concept is the first to be tested.

7 The utPLSQL Framework

In this section the popular unit test framework utPLSQL is examined. First, it is described how a single concept is tested using the utPLSQL framework. Hereafter, an example of testing a dependent concept is presented. Then, the test approach taken by utPLSQL is compared to our approach.

7.1 utPLSQL

The utPLSQL framework is a test framework for the Oracle PL/SQL programming language. It is modelled on the JUnit framework following the extreme programming approach. This means that all test cases and unit tests should be independent.

For the utPLSQL framework to execute a test case it has to conform to certain requirements. For each package the developer wish to test, a separate test package has to be developed. The test package should contain a setup and a teardown method. These are, similarly to the approach proposed in this article, used to create and remove data structures used in the tests. Furthermore, the test package should contain all test cases for the methods to be tested. Each test case consists of a call to one of the functions or procedures to be tested and a call to an assert package. The assert package is used to ensure that certain conditions hold. If all assertions for a test package hold the test cases are successful.

7.1.1 Testing an Independent Concept

This section describes how to perform the test from Section 4.7.1 using the utPLSQL framework. The test is conducted on a package wrapping the *office* table. The method to be tested is the **insert** method. The PL/SQL code for the **insert** method is shown in Listing 8.

```
procedure insert(rec office_rec ) is
  begin
    insert into office
    values(rec.building, rec.room, rec.size )
  end;
```

Listing 8: The insert method.

The **insert** method takes a record as input. The office_rec parameter contains all attributes in the office table. Lines 3 to 4 insert a new row into the office table.

As the **insert** method does not return any value, it is harder to check the correctness of the method. Therefore, we use an auxiliary method to check if a row has been inserted. The exist method is listed in Listing 9.

```
function exist (rec office_rec )
    return boolean is
  b_exist    boolean;
```

```
 4    count_number integer;
 5    begin
 6      select  count(*)
 7      into count_number
 8      from office
 9      where building = rec.building
10      and room = rec.room;
11      if  ( count_number > 0) then
12        b_exist := true;
13      else
14        b_exist := false;
15      end if;
16      return b_exist;
17  end;
```

Listing 9: The exist method.

Lines 6 to 10 count the number of records that matches the input given to the method. If this number is greater than 0, the method returns true, otherwise false is returned.

To test the **insert** method we create a test package as shown in Listing 10.

```
 1  create  or  replace  package body ut_insert  is
 2    procedure ut_setup  is
 3      begin null;  end;
 4
 5    procedure ut_teardown  is
 6      begin
 7        execute immediate
 8        'delete  from office
 9         where room = 'B2'
10           and building = '110a''
11      end;
12
13    procedure ut_insert  is
14      begin
15        insert ('B2','110a',15);
16
17        ut.Assert.eq(
18          exist ('B2','110',15),
19          TRUE
20        );
21    end;
22  end;
```

Listing 10: A utPLSQL unit test package for testing the insert method.

All utPLSQL test packages have a setup and a teardown method. However, in this example no setup is needed. Therefore, the body of the setup method is empty (null). The teardown method is automatically called after completion of the test. The teardown method removes the row inserted by the ut_insert method. In line 15 a row is inserted into the office table. In line 17 the eq method, supplied by the utPLSQL framework, is used to validate if the correct row is inserted into the *office* table. In line 18 the exist method, that was described in Listing 9, is called to check if the row inserted in line 15 exist. In line 19 the expected return value from the exist method is given. If the exist method returns true the test case is considered a success, otherwise it is a failure. The test completes by calling the teardown method that cleans up the *office* table.

7.1.2 Testing connected concepts

This section will show how to test a concept that is connected to another concept through foreign keys. As in Section 4.7.2 the subject of the test is the *teacher* table. As in the previous section, the method to be tested is the **insert** method[2]. Recall from Figure 1 that the *teacher* table is connected to the *office* table. In other words, we cannot insert a row into the *teacher* table without the *office* table is populated. The test package is shown in Listing 11.

```
 1  create  or  replace  package body ut_insert  is
 2    procedure ut_setup  is
 3      begin
 4        execute immediate
 5          'insert  into  office
 6           value('B2','110a',15)';
 7      end;
 8
 9    procedure ut_teardown  is
10      begin
11        execute immediate
12          'delete  from teacher
13           where tid = 10';
14
15        execute immediate
16        'delete  from office
17         where room = 'B2'
18           and building = '110a''
19      end;
20
21    procedure ut_insert  is
22      begin
23        insert (10,'Hans','Jensen',
24                null,'B2','110a');
25
26        ut.Assert.eq(
27          exist (10,'Hans','Jensen',
28                  null,'B2','110a'),
29            TRUE
30        );
31    end;
32  end;
```

Listing 11: A utPLSQL unit test package for testing the *teacher* concept.

As opposed to the previous section, the setup method is not empty. In lines 2-7, a row is inserted into the *office* table. The primary key of this row will be used when inserting into the *teacher* table. In line 9, the teardown method is declared. In line 12, the row inserted into the *teacher* concept is removed. Lines 15-18 cleans up the *office* table, by deleting the row inserted by the setup method. Please note that it is important to remove the inserted rows in a specific order, to avoid breaking referential integrity constraints. The ut_insert method in lines 21-30 inserts a row into the *teacher* table using the primary key from the row inserted into the *office* table. As in the previous section, the Assert package is used to validate the result from the exist method.

[2]The code for the **insert** and the exist methods are the same as in Section 7.1.1, except that the table is no longer *office* but *teacher*. Thus, the attributes have changed.

7.1.3 Comparison

This section provides a series of guidelines describing when it is beneficial to use the utPLSQL framework, and when the approach described in this article should be preferred.

As shown in Listings 10 and 11 the setup and teardown methods quickly become larger. Imagine, a unit test conducted on the *participant* table from Figure 1. The test fixture quickly increases when a table is dependent on more than one other table. Using utPLSQL, the developer has to create the test fixture in the setup method by copying and pasting code from existing unit tests. Similarly, the teardown method have to be constructed manually. This is not the case in the approach proposed in this article. Section 4.6 described how the the presented framework could reuse setup and teardown methods. Thus, the workload for the developer is minimized. Therefore, it is beneficial to use the approach proposed in this article, when testing connected concepts. The framework proposed in this article gives a partially automatically creation of setup and teardown methods. However, the utPLSQL framework is well-suited when a limited test fixture is needed. Especially, the assert package allows the developer to easily test the correctness of programs.

8 Oracle Specific Functionality

This section describes how to use Oracle specific functionality that could aid to achieve the goals of the database unit test framework. This functionality includes the flashback query and workspace management. Although the design criteria specify a vendor independent unit test framework, this section looks at what could have been exploited in Oracle DBMS in order to make the implementation easier.

8.1 The Flashback Query

The Oracle DBMS offers the possibility to query data at a point in time. By default, database operations query the most recently committed data. However, the flashback query can be used to query data from a previous database state. There are two ways to achieve this, either by using a special system change number or by specifying a point in time. To query historical data could prove beneficial in the unit test framework as it allows for the database to return to its state before the unit tests were executed. Using the flashback functionality does not eliminate the need for creating the test fixture. However, it would make the teardown method unnessesary. This flashback functionality is well-suited when the unit test framework is only considered to be in a single-user environment. If the testing environment is a multi-user environment the framework cannot flashback in time when one of the testers is finished testing, as test fixture may be used by another test.

8.2 Workspace Management

Workspace Management allows for multiple versions of rows in user-tables. The versioning infrastructure is not visible to the users of the database. SQL **select**, **insert**, **update**, and **delete** statements continue to operate as normal. New row versions are logically group in each workspace until these versions are explicitly merged into production data or simply discarded. Users in a workspace always see a transactionally consistent view of the entire database, i.e., they see changes made in their current workspace plus the rest of the data in the database. This data either existed when the workspace was created or when the workspace was most recently refreshed with changes from the parent workspace. The workspace management could be exploited in a multi-user unit test framework environment. By allowing each user to test in their own workspace, i.e., they have their own version of the tables or rows they want to test. Their testing activity does not affect any other concurrent tester. By performing the unit tests in a virtual environment without committing the changes to the parent workspace offers maximum concurrency as it isolates the multiple testers completely.

9 Implementation

This section describes core implementation issues of the test framework. It should be noted that this section only covers the implementation of the single user version of the framework, because the changes to the setup and teardown methods are limited, and have already been described in Section 5. The framework is implemented on the Oracle 9i platform. Initially, the generation of setup and teardown methods are described. Hereafter, the implementation of the run method is described. By auto generating the setup and teardown methods, the framework becomes more resistant to changes in the database schema. For example, should a new concept be added to the schema, the developer simply generates a new set of setup and teardown methods. Thus, the workload required to accommodate changes is reduced. Finally, an approach for coupling our framework and the utPLSQL framework is proposed.

9.1 The setup and teardown methods

The generator for the setup and teardown methods is written in C#. Furthermore, the object-oriented framework to represent PL/SQL constructs presented in [5] is used to output PL/SQL source code. As described in Section 4.7 the setup and teardown methods automatically call other setup and teardown methods as needed. The dependencies between concepts are determined using the RDBMS meta data. It is therefore necessary to have access to information about foreign keys. As an example, to find the dependencies of the *participant* concept from Figure 1 on the Oracle 9i platform, the query in Listing 12 is executed.

```
1  select distinct
2    ucc.table_name as table_name,
3    uc2.table_name as references_table
4  from
5    user_cons_columns ucc
6  inner join
7    user_constraints uc
8  on
9    ucc.constraint_name = uc.constraint_name
10 inner join
11   user_constraints uc2
12 on
13   uc.r_constraint_name = uc2.constraint_name
14 where
15   uc.constraint_type = 'R' -- foreign key
16 and ucc.table_name <> uc2.table_name
17 and ucc.table_name = 'PARTICIPANT';
```

Listing 12: Finding dependencies for the *participant* concept.

The distinct clause in line 1 ensures multiple references to the same concept are ignored. Without the distinct clause the *teacher* concept would return two references, one for the *building* attribute and one for the *room* attributes. Lines 2 and 3 project the attributes needed. Lines 4 to 14 join the tables needed to find the referenced concepts. The *user_cons_columns* view contains information of the constraints on a given attribute. The *user_constraints* view is used to find the attributes a given constraint references. Line 15 states that only constraints of type 'R' are interesting. In the Oracle data dictionary 'R' indicates that a constraint is a foreign key. Recall from Figure 1 that the *teacher* concept has a self-reference. Line 16 removes self-references in the result. If self-references were not removed, the setup method on the *teacher* concept would contain a call to the setup method on the *teacher* concept. This would generate a recursive method that never terminates. Line 17 specifies for which concept the setup and teardown are being generated.

Note that the query finds dependencies between tables and not between unit tests. It could also be done between unit tests. The result of the query in Listing 12 is the concepts *student* and *teacher*. Thus, the setup method for the *participant* concept looks as shown in Listing 13.

```
1  procedure setup(self_test) is
2  begin
3    if ( not stack.is_active('participant')) then
4      student.setup(false );
5      teacher.setup(false );
6      stack.push('participant');
7
8      if ( not self_test ) then
9        -- insert data here
10     end if;
11   end if;
12 end;
```

Listing 13: The auto generated setup method of the *participant* concept.

In line 3, it is checked whether the *participant* concept is already on the persistent stack. If the *participant* concept is not active, the *student* and *teacher* concept are setup. In line 6, the *participant* concept is pushed onto the stack. In line 9, the developer should make a call to the methods that inserts data using the public constants.

Similarly, the teardown method is auto generated such that proper cleanup is performed.

9.2 The run method

As described in Section 4.4 the tester calls the run method to initiate a test. Thus, the task of the run method is to setup up the test fixture, execute the test methods, and perform the necessary clean up. For example, to test the *participant* concept, the run method calls the setup method on the *participant* concept. Hereafter, all test methods on the *participant* concept are executed. Finally, the teardown method on the *participant* concept cleans the database. This section describes how the run method locates all test methods on a concept.

As with the setup and teardown methods, access to meta data is also required for the run method to find test methods. On the Oracle 9i platform the views *user_procedures* and *user_source* are used. The *user_procedures* view contains all the active methods. Unfortunately, the methods are sorted alphabetically according to the method names. As test methods are dependent it is necessary to join the *user_procedure* with the *user_source* in order to preserve the same ordering of methods as in the source code. The join between the to views is shown in Listing 14.

```
select procedure_name from (
  select distinct
    up.procedure_name,
    us.line
  from
    user_procedures up
  inner join
    user_source us
  on
    (ltrim(lower(us.text)) like 'function %'
    or ltrim(lower(us.text)) like 'procedure %')
  and (lower(us.text) like
        '%' || lower(up.procedure_name) || ' %'
    or lower(us.text) like
15          '%' || lower(up.procedure_name) || '(%'
16       or lower(us.text) like
17          '%' || lower(up.procedure_name) || ';%')
18     where
19        up.object_name = sought_package_name
20        and up.object_name = us.name
21        and us.type = PACKAGE
22     order by us.line );
```

Listing 14: The SQL statement used to find all active procedures and functions in a package.

Lines 9-17 contain the join clause. Two things need to be satisfied to fulfill the join clause. First, a row in the *user_source* view has to contain either the word function or the word procedure. This is checked in lines 10 and 11. Second, the method name from the *user_procedures* view has to match the name found in the *user_source* view. The procedure name should be followed by a white space, an opening parenthesis or a semicolon. Lines 19 and 20, specifies which PL/SQL packages the methods should be found in. Line 21 ensures that only the package header is considered.

9.3 Integration with utPLSQL

As described in the previous section, the utPLSQL framework provides an assert library that helps the developer in testing his programs. However, as described the utPLSQL requires the developer to copy and paste source code between setup and teardown methods when testing dependent concepts. Therefore, it is interesting to combine the two frameworks to achieve the best from both worlds, namely the assert library and the large user community from the utPLSQL world, and the extensive reuse of code from our world. This section describes how the two frameworks can, with very little effort, be combined.

As described in Section 7.1.3 the primary difference between the two frameworks is the setup and teardown methods. The overall strategy for combining the frameworks is to modify our framework such that utPLSQL follows our test strategy, i.e., reuse the setup and teardown methods instead of copying them.

First, the generator from Section 9 is modified such that the source code produced follows the conventions of the utPLSQL framework. This means that procedure names should have the ut_ prefix. The developer will now use the generated setup and teardown methods as described in 4.4. This implies that the developer should follow all conventions described in Section 4.5 except for the naming conventions, that now should conform to the utPLSQL naming conventions. Thus, the public constants must be published. Furthermore, the developer should remember that test cases are dependent. Moreover, the stack and the methods associated with it must be included.

Hereafter, the developer follows the standard utPLSQL way of writing test cases using the assert library. Again the developer should recall the conventions from Section 4.5, as the order in which test suites and test cases are executed is now important.

With this slight modification of the generator, the two frameworks have been combined to achieve the best of both.

10 Performance Analysis

This section describes the performance improvement gained by using the test methods proposed in this article. Initially, the different approaches to testing are described. Hereafter, the cost of testing the university schema is evaluated.

10.1 The simple approach

The simple approach uses the same bottom-up approach as the approach proposed in this article. The difference is that not only the setup and teardown methods are executed, but also the actual test cases. For example, to test the *semester* concept from Figure 1, the test is setup, the test cases are executed, and finally the test fixture is torn down. If the *student* concept is to be tested, the entire *semester* test, including the setup and teardown methods, are executed. Hereafter, the *student* concept is setup, tested and torn down. For each concept to be tested, all concepts that the current concept depends upon are also tested. Thus, to test the entire university schema from Figure 1 the tests in Table 1 are performed.

Test	**Dependencies**
Semester	
Office	
Student	Semester
Teacher	Office
Course	Semester, Office, Teacher
Participant	Semester, Office, Teacher, Course, Student

Table 1: The tests of the university schema and their dependencies.

The first column in Table 1 shows the concept to be tested. The second column shows the dependencies of that test. For example, to test the *teacher* concept, the *office* concept is also tested.

10.2 Our approach

The approach proposed in this article resembles the approach in the previous section. The difference is that the test cases are not executed for depending concepts when they are setup. Only the setup and teardown methods are reused. For example, to test the *student* concept the setup method on the *semester* is used. The test cases for the *student* concept is executed. Then the *student* concept is torn down. Finally, the *semester* is torn down as well. Thus, compared to the simple approach, the test cases for dependant concepts are not executed. However, the same number of setup and teardown methods is used.

10.3 Using topological sort

As described in Section 6 our approach is further optimized by specifying the order of the unit tests and by modifying the teardown methods. Table 2 shows the steps performed in testing the university schema.

Test	**Steps performed**
Semester	sem.setup, sem.test, sem.teardown
Office	off.setup, off.test, off.teardown
Student	stu.setup, sem.setup, stu.test, stu.teardown
Teacher	tea.setup, off.setup, tea.test, tea.teardown
Course	cou.setup, tea.setup, off.setup, sem.setup, cou.test, cou.teardown
Participant	par.setup, cou.setup, tea.setup, off.setup, stud.setup, sem.setup, par.test, par.teardown
Clean up	par.teardown, cou.teardown, tea.teardown, stu.teardown, off.teardown, sem.teardown

Table 2: The steps taken by the topological approach.

10.4 Comparison

This section compares the performance of the three approaches presented. The performance is measured according to the number of calls to the setup and teardown methods, and to the number of unit tests performed when the entire university schema is to be tested. The number of unit tests refers to the number of times all the test cases for a concept is executed. Table 3 show a comparison of the three approaches.

	# setups	# teardowns	# tests
Simple	16	16	16
Our appr.	16	16	6
Topologic	15	12	6

Table 3: Number of methods call when testing the university example schema

From Table 3 we see that the number of times each concept is tested drops dramatically from the simple approach to the two alternative approaches. As the test cases typically carry more lines of code than the setup and teardown methods, this reduction is most likely causing a bigger performance gain than the reduction of setup and teardown methods. Note that executing six tests are minimal. Using the topological sort approach, it is possible to cut off one setup method and four teardown methods and thus, further improving the performance of the test framework.

The reduction of one setup method is caused when the setup method from the *participant* concept is executed. The setup method of the *teacher* concept returns immediately and skips the setup of the *office* concept, as the *teacher* concept already has been set up. Hence, the *office* concept must have been set up already as well. The difference between the number of teardown methods executed is caused by the deferral of teardown in the topological approach. In general, the more relations between the concepts in the database schema the greater the benefit is from using topological sorting.

11 Conclusion

In this paper, we presented and discussed the issues of designing and implementing a unit test framework. Alternative approaches for testing database applications require that all test cases should be independent. In this article we argue that it is an advantage for test cases to depend on each other.

The contribution of this paper is a design and an implementation of a unit test framework. The framework supports multiple users testing simultaneously when the conventions of the framework are followed. Compared to existing frameworks, the approach proposed in this article minimizes the test fixture. The minimal test fixture is achieved by only having setup and teardown code in a single place, whereas existing frameworks requires the developer to copy source code between test packages. Furthermore, because existing frameworks require that test cases must be independent, they must perform setup and teardown between every test case. In our framework, this is only done once per unit test. Moreover, the test fixture is further minimized by allowing concepts to remain set up through several unit tests.

The framework gives the possibility of automatic cleanup of the database, in case of a system crash or a malfunctioning test case.

The ideas presented in this article, are not only applicable in a database setting. It was shown how references between objects using collections resemble the foreign keys in databases. Thus, the ideas of the framework are also applicable to programs using association or aggregation.

With slight modifications of the generator presented, we are able to apply the ideas from this article into the utPLSQL framework and hereby achieve the best features from both frameworks.

In the following, future work is discussed. A specific problem is that some tables have surrogate primary keys that automatically increment. This poses a problem because the next key value is unknown at compile-time. This makes it more difficult for the test programmer to satisfy the constraints given by the test framework, namely that usable primary keys must be available at compile-time.

At this moment, the framework does not allow testing of real-time constraints. Imagine a conveyor belt in an airport transporting luggage from passengers to their planes. Here, it might be desirable to specify and test a constraint, stating that a piece of luggage should reach the airplane no later than one minute after the passenger checked it in.

Acknowledgements

For providing us with references to related work, we thank Brian Nielsen of CISS, Aalborg University.

References

[1] Dbunit. http://dbunit.sourceforge.net/. As of 31.05.2004.

[2] Kent Beck. *Extreme Programming Explained - Embrace Change*. Addison-Wesley, 2000.

[3] Kent Beck and Erich Gamma. Junit cookbook. http://junit.sourceforge.net/doc/cookbook /cookbook.htm. As of 31.05.2004.

[4] David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weber. A framework for testing database applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 147–157, 2000.

[5] Claus Abildgaard Christensen, Steen Gundersborg, Kristian de Linde, and Jacob Richard Thornber. A generic and portable database api. http://www.cs.aau.dk/∼kdl/master/, 2004. Aalborg University.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 2001.

[7] Haifa University Software Engineering Course. http://cs.haifa.ac.il/courses/softe/JUnit.ppt. As of 31.05.2004.

[8] Webopedia Regression Testing Definition. http://www.webopedia.com/TERM/R/regression_testing.html. As of 31.05.2004.

[9] Steven Feuerstein. utPLSQL project. http://utplsql.sourceforge.net/. As of 31.05.2004.

[10] Ramzi A. Haraty, Nash'at Mansour, and Bassel Daou. Regression testing of database applications. In *Proceedings of the 2001 ACM SAC*, pages 285–289, 2001.

[11] Paul C. Jorgensen. *Software Testing - A Craftsman's Approach*. CRC Press LLC, second edition, 2002.

[12] JUnit. http://www.junit.org. As of 31.05.2004.

[13] Willian E. Lewis. *Software Testing and Continuous Quality Improvement*. Auerbach, 2000.

[14] Oracle. Catalog Views / Data Dictionary Views. http://otn.oracle.com/pls/db901/ db901.catalog_views?remark=homepage. As of 31.05.2004.

[15] Oracle. Managing Viws, Sequences, and Synonyms. http://download-west.oracle.com/docs/ cd/B10501_01/server.920/a96521/ views.htm#581. As of 31.05.2004.

[16] William Perry. *Effective Methods for Software Testing*. John Wiley & Sons, Inc., 1995.

[17] Don Wells. Extreme programming: A gentle introduction. http://www.extremeprogramming.org/people.html. As of 31.05.2004.