A Generic and Portable Database API

Project Period: Dat6, February 2nd, 2004 – June 1st, 2004

Supervisor: Kristian Torp

Claus Abildgaard Christensen

Steen Gundersborg

Kristian de Linde

Jacob Richard Thomber

A Generic and Portable Database API

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

1st June 2004

Abstract

Most application developers do not fully master SQL as well as their favorite programming language, e.g., Java or C++. Furthermore, repetitive or equivalent SQL statements make it problematic and time consuming to test database interactions spread throughout an entire application. Hence, a need for a database abstraction layer exists. We suggest encapsulating SQL and database access via an Application Programming Interface (API).

We design and implement a generic and portable API. Generic refers to the fact that no vendor specific features, like Oracle's embedded Java, have been used in the design of the API. It aims at providing most used functionality in application development. To accommodate needs for extending the API, the design is modular. The goal of being portable is being independent from programming-language paradigms. Two versions of the API have been proposed: A static and a dynamic one. The static API catches many errors at compile-time, increasing the number of methods. In contrast, the dynamic API has fewer methods but most error checking is deferred until run-time.

Furthermore, a generator for the API has been designed and implemented. Currently, the generator is capable of generating a PL/SQL specific API for the Oracle platform. The generated API covers most commonly used functionalities in database application development.

1 Introduction

The advent use of relational database management systems in application development calls for the use of an abstraction layer. Such an abstraction layer eases the task of ensuring correctness and robustness of database applications. Therefore, it is preferable to introduce this abstraction layer as a uniform and testable Application Programming Interface (API).

The most commonly used language for interacting with databases is SQL. Most application developers fully masters one or more high-level programming languages such as Java or C++. However, they do not consider SQL to be one of these languages. Therefore, there is a basis for encapsulating SQL from the developers and hereby ease the task of developing software.

This article presents a generic API design for wrapping SQL code in an object-oriented or procedural programming language. To the best of our knowledge such a design does not exist in the field of API and database research. A design that is independent of the underlying schema, the Database Management System (DBMS), and the programming language. Furthermore, the article presents a tool that enables developers to generate the proposed API. The generator enables developers to specify precisely what the generated API will contain. The overall design of the generator is modular. This minimizes the effort of changing DBMS and programming language.

The contributions of this article are the following. First, design and implementation details on the proposed language independent APIs is presented. Secondly, a generic, comprehensive, and modular design of the APIs is given. Next, we give design details and implementation of a generator for the API.

The article presents two versions of the API. The first is a static version containing many and more specific methods, allowing for a high level of compile-time error detection. The second is a dynamic version containing fewer and more generic methods. Generic methods are more difficult to check for errors during compile-time. Thereby, the risk of encountering run-time errors is increased.

The paper is structured as follows. Section 2 looks at the related work. Section 3 presents the design of the proposed API. Section 4 gives an elaborate example of how to enforce business rules using the API. In Section 5, the article presents the difficulties encountered during the implementation of the API. Section

6 presents the design of the API generator. Then, Section 7 compares the performance of the static and the dynamic APIs. Finally, in Section 8 the conclusion and the directions of future research directions are given.

2 Related Work

The Hibernate framework [3] is an object/relational mapping tool. Hibernate provides a dynamic Java API for RDBMSs. Dynamic refers to the fact that the actual SQL statements are generated at run-time, the SQL strings are build during program execution. The goal of the framework is to develop persistent objects following common object oriented aspects – including association, inheritance, polymorphism, and composition.

Hibernate provides tools to auto-generate a database schema from a mapping file. Furthermore, it provides a tool that is able to auto-generate Java classes with object-level get and set methods from the mapping files. However, before using the features of Hibernate, the developer has to create a mapping file that describes in which tables an object should be stored and how objects are associated.

When the XML files containing the mapping information have been created, the Hibernate framework allows the developer to store objects persistently with a single method call. If the object to be stored is associated with other objects, these will also be stored in the database.

Symmetrically, the framework supports retrieval of either single objects or web of objects. Objects can be retrieved either by using a load method, that takes the object identifier (primary key) as input or by providing a list-method with a query written in the Hibernate Query Language (HQL). The HQL is an object-oriented query language with many similarities to SQL. The HQL supports the most common constructs known from SQL but also aggregate functions, subselects, inner and outer joins, as well as some features from the object-oriented paradigm – inheritance, polymorphism and associations. By having a high-level language like HQL, developers are able to move between RDBMSs simply by changing a string in a configuration file, provided the underlying RDBMS support the features used.

Hibernate provides an object-oriented Java specific API whereas this paper presents a generic design portable to all programming languages and a PL/SQL specific implementation. Furthermore, this paper presents both a static and a dynamic API.

In [9], Nink et al. present a study on how to exploit maximum schema independence while maintaining the highest level of compile-time error detection possible. They suggest an architecture consisting of a call-level interface (CLI) compiler, a cache module, and a configuration language used to ease application customization. While the CLI compiler stands for prefetching and pointer swizzling, e.g., it also generates a Run-Time System (RTS). Here, session control and database access are provided. Moreover, the CLI compiler utilizes the database schema and query knowledge along with a configuration language to compile, i.e., to generate the RTS that is late in terms of schema binding time, but late in terms of methods names to achieve a high error detection rate during compile-time. Also, in [9], they introduce an interesting compromise between the interface and the actual implementation. They conclude that even though the interface may be generic, i.e., late bound, the implementation may still be generated. This combination, introduced as *virtual late binding*, does not capture compile-time errors completely, however they claim that it ensures a good performance instead and maximizes the schema independence.

Nink et al. provide sparse implementation details, and is in general more abstract in their presentation. We focus on creating an API that is comprehensive and modular. However, we do not consider cache modules, pointer swizzling and so forth.

A technology becoming increasingly popular is Enterprise JavaBeans (EJB) [4, 6]. One type of beans, namely the entity beans, abstracts the underlying database. In a simple setting, an entity bean maps to a single row in a table. The entity bean provides access to the data through get/set methods for each attribute and finder methods that wrap SQL select-statements. Furthermore, the EJB layer handles persistency, concurrency, business rules, and security issues.

The API proposed in this article is also a database abstraction layer. However, the functionality of the methods offered by the entity beans is only a subset of the functionality of our API. Specifically, the get/set methods are covered by our attribute get/set interface and the functionality of the finder methods is covered by our list interface. Since our API does not consider handling persistency, concurrency, business rules, or security issues, our abstraction layer is thinner than the EJB abstraction layer.

In [1], Ege et al. describe a Java based API for object-oriented databases. The goal of the research is to provide the application programmer with seamless access to objects no matter if they reside in main memory or they are stored in a database. To achieve this, the authors propose a four layer architecture consisting of a Java front end, a facilitator, a communication protocol, and a database server. The authors create an abstract superclass. Any Java class that inherits from this superclass can be stored persistently

in a database. The developer can store any object by calling a persist method, that is inherited from the superclass. As with the Hibernate framework [3], any objects associated with the object to be stored, are also stored with a call to the persist method. Ege et al. only allows for retrieval of named objects. Therefore, the developer has to give each object a unique name using a so-called bind method. This name is used when the developer retrieves objects from the database.

The task of the facilitator is to translate the commands issued in the Java program into IceCubes, which are transmitted using the communication protocol. A command issued in the Java program could for instance be to retrieve a specific object or to create a new one. An IceCube is a bytestream that describes exactly what object that is to be manipulated and what sort of operation the database should perform. In order to decode an IceCube Ege et al. implements a so-called engine interface, that resides on the database. The sole purpose of this interface is to decode the IceCubes and perform the operations specified in these.

Unlike the API designed by Ege et al. the API proposed in this article is implementable in both a procedural and an object-oriented language. Furthermore, Ege et al. only provides methods for retrieving a single object, whereas this article also proposes methods to retrieve multiple rows from the database.

In [5], McLellan et al. describe how to build user-friendly APIs. The authors argue that when companies purchase APIs or program libraries the programmers become users too. Therefore, it is necessary to develop guidelines of how an API should be documented to support ease of use. McLellan et al. develops an example API and monitor actual programmers' use of this API. Through this investigation the authors are able to conclude that pseudo code, along with actual examples of how to use the code, are essential in order to make a user-friendly API. McLellan et al. solely focuses on building an API for testing its user-friendliness. We present an API where user-friendliness is only one of our design criteria.

The design and implementation of APIs have been explored in many contexts, e.g., in the biometric world. In [12], Tilton describes how the creation of a standard generic biometric API, would allow for easy substitution of biometric technologies and simple integration among biometric technologies.

3 Design of the API

This section describes a static and a dynamic API for database programming. The idea of the APIs is to increase abstraction by providing a number of methods for the application programmer. This is done by providing him with a set of interfaces with methods for manipulating the database. These interfaces are to be thought of as Java interfaces [8]. Like in Java, each interface guarantees certain functionality.

This section only covers the purpose and ideas of the interfaces. Formore details see appendix A. In the following,examples are shown in PL/SQL-like pseudo code. Error handling has beenomitted to keep focus on the idea of the API. First, we will considera small university schema that is used as an example throughout thearticle. Next, terminology needed is introduced. Hereafter theassumptions made before design and implementation is presented. InSection 3.3, the design criteria for the APIsare discussed. Then, Section 3.4 elaborates on thedesign of the static API by presenting several interfaces. In Section3.5, the dynamic API is illustrated by presenting twointerfaces that differ from their static version. Section3.6 evaluates the design criteria withrespect to the interfaces presented. Finally, Section3.7 compares the overall differences between thestatic and dynamic APIs.

The schema depicted in Figure 1 is used as an example schema when describing the API.

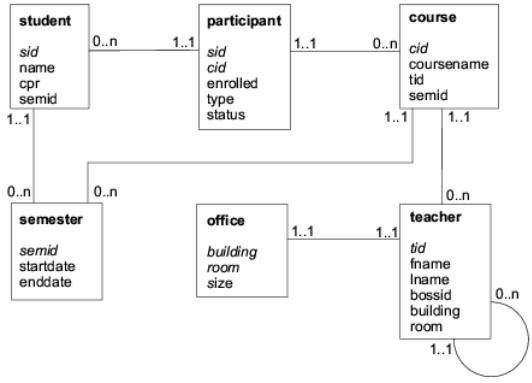

Figure 1: The University Example Schema. Attributes in italics denotes the primary key of a table

The example schema represents a small university. The university has anumber of students, information on these is stored in the *student* table. The *course* table stores thecourseidentifier (*cid*), the course name and the foreign keyteacher identifier (*tid*). The *participant* table is arelationship 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. Inaddition, the *participant* has three additional attributes.First, *enrolled* that describes when the student is enrolled tothe course. Secondly, a *type* attribute that describes if

thestudent is following the course in the traditional manner or if he isdistant learning. Finally, the *status* attribute describes ifthe student is active or inactive in the course. As depicted in theER-diagram in Figure 1, a student can follow zeroor many courses and a course can have zero or many students. Recallthe foreign key *tid* in the *course* table, this keyreferences the *teacher* table. Each course has exactly oneteacher associated. A teacher can teach zero or many courses. Ateacher can act as a supervisor for other teachers. This is modeled bythe self-reference and the *bossid* attribute on the*teacher* table. Finally, each teacher is situated in exactlyone office. This is modeled through the composite foreign keyconsisting of *building* and *room*, which references the *office* table. The *semester* table is used to connect studentsand courses to specific semesters, e.g., an introductory semester. Theprimary key of the table is the *semid* attribute, which is usedas a foreign key in the *student* and *course* tables.Moreover, the *semester* table has two date attributes indicatingthe start and end date of a semester.

3.1 Terminology

This section introduces the terms static and dynamic API. Moreover,three data structures, which are used when describing the design andimplementation of the API, are introduced. Finally, the terms *attribute name* and *attribute value* are explained. The term static denotes that the API utilizes static SQL. This meansthat every SQL statement is known at compile-time of the API. The termdynamic indicates that the SQL statements are not known untilrun-time. The first data structure is the *concept* which corresponds to aheader file in an imperative language, a class in an object-orientedlanguage or a table in the database. A concept could be the*student* table in Figure 1. The second data structure is the *record* which is an instance ofa concept; it denotes a set of attribute values. A record correspondsto a row in a table. A record from the *student* concept iscomposed of the attributes *sid*, *name*, *cpr*, and*semid*. The third data structure is the *record id* which is the primaryidentification of the record. A *record id* is a subset of theattributes of the record. This could be the *sid* on the*student* concept. The *attribute name* denotes all attribute names on a givenconcept. The *attribute value* denotes a value of the typeassociated with *attribute name*.

3.2 Assumptions

This section describes the assumptions made in this article. Theseassumptions are made to focus the scope of the article.

The database schema exists The APIs cannot be generated if the schema does not exist. The generator does not support generating the database schema given an API as the Hibernate framework does.

Schema manipulation The APIs does not support any schema manipulating operations. The APIs focus solely on doing data manipulation.

All concepts have a *record id* Most methods of the APIs require a *record id* for unique identification of records.

Database meta-data must be accessible Meta-data is required for determining the return types for methods. Attribute names are needed for method naming when generating the static API. Information about primary keys is required by most methods. Furthermore, information about unique and foreign keys is necessary for some methods.

3.3 Design Criteria

To guide the process of designing the APIs, a collection of designcriteria have been specified. Furthermore, these criteria aresubsequently used to evaluate the design of the APIs. This sectionwill describe each design criterion in turn.

Comprehensive As most developers do not favor SQL, the APIs have to provide an alternative that is easy to comprehend. This also implies that the naming conventions used should be intuitive, allowing the developer to guess the functionality of methods without having to read the actual code. The fact that the APIs are designed to be comprehensive also implies a "read-once – understand anywhere" mechanism, i.e., once the functionality of an interface on one concept is understood, understanding the functionality of the interface on any other concept is straightforward.

Auto-generation Through the generator and an XML input the developer should be able to specify exactly which methods and interfaces are to be generated. Furthermore, methods that are not supported on a given concept should not be generated. This increases the comprehensibility of the APIs. Moreover, meta-data available from the database should be exploited during the generation of the APIs.

Symmetric The APIs are designed to be symmetric. This means that if a `get` method exists, so

does a **set** method. Similarly, the APIs provide an output interface and thus an input interface performing the exact opposite functionality. Again, this is a criterion that improves the comprehensibility of the APIs, and hereby making them more appealing for a developer to use.

Orthogonal The fact that the APIs are orthogonal means that redundant functionality does not exist. For example, the APIs only provide an exist method, not a has nor an in method that are semantically equivalent. The orthogonal design also improves the comprehensibility of the APIs in that the developer only has to look for specific functionality one place.

Support frequent use An important goal of the APIs is to provide the application developer with the most used functionality, without resorting to writing SQL code. This implies that the APIs should contain methods for the most commonly used functionality. Naturally, complete APIs can never be developed. Should a developer require very specific or highly specialized functionality, the developer has to extend the API himself.

Modular The design of the APIs should be highly modular. This makes it easier to extend the APIs with new functionality.

3.4 Static API Design

In this section the interfaces contained in the static API are described. Each interface will be presented as follows. Initially, the purpose of the interface is given. Hereafter, a list of methods constituting the interface is given. Then each method is described in turn. When considering overloaded methods only one is presented. Appendix A lists the entire API. Finally, considerations for each interface are given. If an interface face has been designed with special attention to some design criteria, this is also discussed.

In Figure 2 all interfaces of the API are depicted. Most interfaces are independent from each other. An example could be the interfaces record lock and walker. However, some interfaces utilize functionality provided by other interfaces. As an example, the status interface uses the attribute get/set interface. In Figure 2 some interfaces are marked by (i). This denotes that these can be implemented as interfaces in the Java programming language [8]. For example, the names of the methods constituting the status interface remain the same regardless of the concept they are implemented on. In contrast, the names of the methods in the get/set interface changes depending on the concept. Thus, this is not implementable as a Java interface.

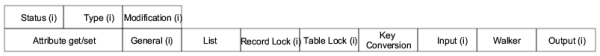

Figure 2: Relationship between the interfaces of the API.

3.4.1 Attribute Get/Set Interface

The purpose of the attribute get/set interface is to provide access to single attributes. Listing 1 shows the implementation of the getName method.

```
1  function getName(record_id) return varchar is
2    result student.name%type
3  begin
4    select name
5      into result
6      from student
7     where sid = record_id;
8     return result ;
9  end;
```

Listing 1: Implementation of the get method.

Line 1 defines the method signature and return type. Line 2 denotes the type of the result variable to be the same type as the *name* attribute of the *student* concept from Figure 1. The SQL statement in lines 4-7 retrieves the *name* attribute into the result variable that is returned in line 8.

The interface contains the following methods.

- get<attribute name>(<record id>)
  return <attribute value>
- **set** <attribute name>(<record id>, <attribute value>)

Explanation It is important to note that the API provides a method for each column in a table, i.e., if we consider the *student* concept on Figure 1 the API provides three get methods getName(<record id>), getCpr(<record id>), and getSemId(<record id>). Equivalently, the following three **set** methods are provided: setName(<record id>,name), setCpr(<record id>, cpr), and setSemId(<record id>, semid). All get methods are implemented using SQL **select** statements, similarly all **set** methods are implemented using SQL **update** statements.

Considerations As we can see from Figure 2 the attribute get/set interface is one of the basic interfaces on which both the status and type interface builds upon.

Allowing a programmer to retrieve an attribute using the get method, could lead to inconsistencies. Therefore, the get methods also support locking the record from which attributes are retrieved. Two methods for locking a record are implemented.

The programmer has the option of calling the get method with the addition of a boolean parameter. If the parameter is set to true, as in Listing 2, then the

record is locked. If the boolean parameter is not given, the parameter defaults to false, i.e., locking of records is disabled.

```
1  -- record id number 1 is locked
2  name := getName(1,true)
```

Listing 2: Enabling locking when retrieving records by using and additional parameter.

Alternatively, the programmer can use the auxiliary method enableLocking(locking boolean) which changes the default behavior of the get methods. An example is shown in Listing 3.

```
1  -- change the default behavior of the get-methods
2  enableLocking(true)
3  -- record id 1 is locked
4  name := getName(1)
```

Listing 3: Changing the default locking behavior.

Reflecting on the design criteria in Section 3.3, this interface has been designed to be comprehensive, symmetric and to support frequent use. The interface is quite similar to a standard Java bean. Thus, developers accustomed to object oriented programming should find this interface easy to use. The functionality of this interface could be covered by simply retrieving an entire <record>. However, developers accustomed to object oriented programming are used to getting and setting single attributes.

3.4.2 Type Interface

The idea behind the type interface is to provide special access to one particular attribute, namely the *type* attribute. The motivation for providing this functionality is that data of different types is often stored in a single concept. Then a *type* attribute is used to differentiate between the different types. Examples of such a concept could be the *participant* concept that has a *type* attribute, that describes whether the student follows the course on a normal basis, or if he is learning remotely. Another example from Oracle's data dictionary [10] views could be the column *all_tables.temporary* which is used to specify whether the concept is of type temporary or a persistent concept, or the *all_objects.object_type* which describes the type of the objects stored, e.g., Java object, package or package body.

The methods provided by the type interface are the following.

- getType(<record id>) **return** <attribute value>
- setType(<record id>, <attribute value>)

Explanation The getType method is used to select the *type* attribute from a <record>. The implementation of this is a **select** statement.

The setType method is implemented using an **update** statement as shown in Listing 4, here we use the *participant* concept as an example.

```
1  procedure setType(sidIn, cidIn, newType) is
2
3  begin
4    update participant set
5      type = newType
6    where sid = sidIn
7    and cid = cidIn
8  end;
```

Listing 4: Implementation of the setStatus method.

Considerations As depicted on Figure 2 the type interface builds on top of the attribute get/set interface. The get and **set** methods provided by the type interface are exactly the same as the methods in the attribute get/set interface. However, using the type interface it is only possible to manipulate a single attribute, namely the *type* attribute.

With respect to the design criteria, this interface has been designed mainly to accommodate the frequent use of the **type** attribute. If the get/set interface is implemented on the concept then the functionality of the type interface is already covered.

3.4.3 Status Interface

The status interface is similar to the type interface, in that it provides special access to one particular attribute, in this case the *status* attribute. Recall the *participant* concept in Figure 1. Here the *status* attribute indicates whether or not the student is active or inactive. An additional example, from Oracles data dictionary views [10] could be the column *all_triggers.status* that gives the status of triggers that can be enabled or disabled.

The status interface provides the following methods.

- getStatus(<record id>) **return** <attribute value>
- setStatus(<record id>, <attribute value>)

Explanation As the status interface is almost similar to the type interface. The getStatus method is implemented using a **select** statement, and the setStatus method is implemented using an **update** statement.

Considerations The implementation of the interface relies on the attribute get/set interface. In fact if the get/set interface is implemented on the concept then the functionality of the status interface is already covered. As with the type interface, the status interface also supports the frequent use design criteria.

3.4.4 General Interface

The general interface is inspired by the java.lang.Object class [7] known from the Java programming language. The methods of the interface only queries the database, i.e., no modification of the underlying data is done. Furthermore, the interface provides a method to retrieve a single <record>.

The methods provided are the following.

- toString(<record id>) **return** string
- exist (<record id>) **return** boolean
- equal(<record id>, <record id>) **return** boolean
- eequal(<record id 1>, <record id 2>, predicate) returns boolean
- clone(<record id>) **return** <record>
- get(<record id>) **return** <record>

Explanation The first method is the toString method that takes a <record id> as input and convert the row identified by the <record id> to a string. The method is implemented as a **select** statement.

The exist method accepts a <record id> as input. The method determines if the <record id> is all ready in the table. The method returns a boolean value indicating if this is the case or not.

The interface also provides an equal method. The input to this method is two <record id>s. The two <record>s identified by the <record id>s are retrieved from the database and compared. In order for this method to return true, every attribute of the two <record>s must be equal, except for the attributes constituting the primary key and any candidate keys. For example consider Table 1.

sid	*name*	*cpr*
3	Finn Jensen	1505801357
4	Hans Kjeldsen	0709783579
5	Hans Kjeldsen	2412815237

Table 1: *sid* is the primary key.

If we use equal on <record>s identified by *sid* 4 and 5 in the Table 1 the method returns true. The reason for this is, that the only attribute used in the comparison is *name*. If we included the *sid*, which is the primary key or the *cpr*, which is a candidate key, we would never have two identical <record>s.

The equal method is overloaded such that it also takes an additional parameter. The additional parameter is a predicate. The predicate describes which attributes are compared for equality. The equal method returns true if the attributes, described in the predicate, of the two rows identified by their respective <record id>s are equal.

The clone method provides a mechanism for making copies of a <record>. Naturally, a cloned record cannot be inserted into a concept as this would cause a duplicate key error.

As opposed to the get methods in the attribute get/set interface, that are used to retrieve single attributes, the get method in the general interface is used to retrieve an entire <record>. The method is implemented as a **select** statement that selects all attributes into a <record> and returns it.

Considerations The general interface is, like the attribute get/set interface, one of the fundamental interfaces, which other interfaces build upon. As with the attribute get/set interface, the general interface also allows the programmer to lock the record that is being retrieved. Thus, potential consistency problems are eliminated. As in the get/set interface this is implemented by adding a boolean parameter.

The general interface has been designed focusing on frequent use design criteria. Developers accustomed to the Java programming language will find the general interface very comprehendible as the interface is very similar to the functionality provided by java.lang.Object

3.4.5 Modification Interface

The modification interface provides methods for inserting, updating, and deleting <record>s in a concept.

The interface provides the following public methods.

- iinsert (<record>)
- uupdate(<record>)
- uupdate(<record id>, <record>)
- ddelete(<record>)
- canInsert(<record>) **return** boolean
- canUpdate(<record>) **return** boolean
- canUpdate(<record id>, <record>) **return** boolean
- canDelete(<record>) **return** boolean

Explanation The iinsert method takes a <record>, and inserts it into the concept on which it is implemented. The method is implemented as a insertion SQL statement.

The interface provides two uupdate methods. The first one takes a <record> as input and updates the

<record> identified by the <record id> in the <record> parameter to the new values, i.e., the method does not update the <record id>.

The second uupdate method takes a <record id> and <record> as input. All attributes, including the primary key, in the <record> identified by the <record id> are updated with the values specified in the <record> parameter.

The canInsert method determines if a given <record> can be inserted into the concept. It returns true if the concept is not locked. Furthermore, it is checked that the <record> to be inserted does not violate any foreign key constraints. As an example, the canInsert method implemented on the *teacher* concept from Figure 1, checks if the *building* and *room* attributes specified in the <record> in fact exists in the *office* concept, such that the foreign key constraints are not violated. However, the data to be inserted is not validated against check-constraints. Hence, even though this method returns true is not guarantied that the record can be inserted. Check-constraints can still fail and the state of the concept may change immediately after the canInsert method has completed.

The canUpdate, canUpdate, and canDelete methods checks the same conditions as the canInsert method, namely if the concept is locked and if any foreign key constraints will be violated by performing a given action.

Considerations It is debatable whether the database application programmer should be allowed to update primary keys.

3.4.6 Output Interface

The output interface is used for printing information onto different sources.

The interface provides the following public method:

- output(<record id>, <target>, <format>)
- output(list <record id>, <target>, <format>)

Explanation The output method takes three parameters. First, the object to print. Second parameter is the source onto which the <record>s should be printed. The third parameter describes the format in which the objects should be printed. At this time the possible values are comma separatet which is shown in Table 2, header-value is shown in Table 3, and name-value which is shown in Table 4. In each of the tables the row identified by the *sid* 4 from Table 1 is used. The second and third parameter are optional, if they are left out, they default to screen and a comma separated string. If more than a single <record> is to be printed, the output method also accepts a list of <record id>s as input.

4,Hans Kjeldsen,0987654321

Table 2: The comma separated format

sid	*name*	*cpr*
4	Hans Kjeldsen	0987654321

Table 3: The header-value format

sid:	4
name:	Hans Kjeldsen
cpr:	0987654321

Table 4: The name-value format

Considerations When designing this interface limiting the number of possible output formats has been a consideration.

3.4.7 Input Interface

The purpose of the input interface is to allow the application programmer to load data from files directly into concepts.

The interface provides the following method:

- input(<filename>, <path>, <format>)

Explanation The input method accepts three parameters; the filename to open, the path to the file, and the format of the file. The method opens the file and inserts all rows into a concept. As an example, we could use the output method from the output interface to write a comma separated file containing all the entries from the *student* concept. Hereafter, we could use the input method to insert the data from the file into the *student* concept again. Naturally, the input method accepts the same formats, as the output method from the output interface.

Considerations A natural extension to the input interface would be to allow input from other sources.

Together with the output interface, this interface is symmetric in the sense that everything outputted can also be inputted and vice versa.

3.4.8 Walker Interface

The purpose of the walker interface is to retrieve <record>s that are connected via foreign keys to the current <record>. The walker interface could, e.g., be used to retrieve the web of connected <record>s using foreign-key information. As an example, the method in Listing 5 shows how a teacher is found using a *cid*.

```
1  function getForeignTeacher(courseId)
2      return teacher%rowtype is
3    resRecord teacher%rowtype;
4  begin
5    select  teacher.* into resRecord
6    from teacher, course
7    where teacher.tid = course.tid
8    and course.cid = courseId;
9    return resRecord;
10 end;
```

Listing 5: Implementation of the getForeignTeacher method.

The %**rowtype** in line 2 and 3 results in a record variable containing the all attributes in the *teacher* concept. The method is a **select** statement joining the *course* and *teacher* concepts, as seen in line 6. Line 7 ensures that each teacher only is associated with courses containing that specific *tid*, and line 8 states that only a specific course is of interest.

The interface contains the following methods:

- getForeign<concept>(<record id>)
  return <concept> list of <record>
- get<Concept>sFor<Foreign concept>
  (<record id>)
  return <concept> **type** <record>

Explanation The getForeign<Concept> method is used to retrieve the <record> corresponding to the foreign key in the <record> denoted by the <record id>. The get<Concept>sFor<Foreign Concept> method is used to retrieve all the <record>s pointing to a specific foreign key value. As an example the getCoursesForTeacher yields a list of *course* records for which the *course.tid* is equal to the <record id> given as input.

Considerations Using the walker interface an application programmer can join multiple concepts without writing any SQL. For example, to list all students and the name of the courses they are following, if they are following any, the programmer would, without the API, have to write a query involving two left outer joins. Using the API, the programmer could use the list method to retrieve all students. Then, for each student use the getForeignParticipant method, to get a list of *cid*'s for courses the student is following. Finally, the getForeignTeacher is used for each <record> returned by the previous method called. Now the desired result is achieved, a list of all students and the courses, if any, the students are following is produced.

3.4.9 Key Conversion Interface

The key conversion interface is used to convert a primary key to a candidate key and vice versa. The methods provided by the interface are the following.

- <candidate key 1>2<candidate key 2>
  (candidate key 1 value)
  return <candidate 2 value>

Explanation Recall Figure 1 and focus on the *student* concept. Here, this interface would provide the methods sid2cpr(<attribute value>) and cpr2sid(<attribute value>).

Considerations Notice the symmetry between the methods. When conversion done from one candidate key to another, it is always possible to go the other way as well.

3.4.10 Record Lock Interface

The record lock interface is for row locking. The lock mode can be either a shared lock or an exclusive lock. The interface contains one method that is listed below.

- recordLock(<record id>,<lock mode>)

Explanation The method recordLock takes in as input the <record id> of the concept together with a <lock mode>. This interface does not contain a releaseLock method, however. Releasing a record lock can be accomplished in two ways. First, the user can call the commit method to actually commit changes made during the record lock session. Secondly, the rollback method can be called to undo all changes in the present rollback segment.

3.4.11 Table Lock Interface

The table lock interface allows the user to lock an entire table in either a shared lock or in an exclusive lock. It has the same commit/rollback behavior as used in the record lock interface. The table lock interface could be used to ensure consistency during a critical update of a system.

3.4.12 Predicate Package

The predicate needs to be introduced now because it is used by the next two interfaces, namely the count and the list interfaces. Before going into details with the methods of the predicate package an example of its use will be presented. Say we want to count the number of students in the *student* concept where their names is either Hans or Ole. Here, the SQL **where** clause would be the following. **where** name = 'Hans' **or** name = 'Ole'. This is modelled this using the binary tree depicted in Figure 3.

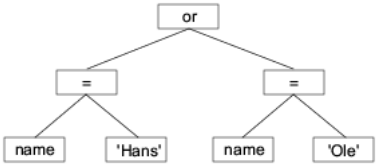

Figure 3: Binary tree denoting the **where** clause name = 'Hans' **or** name = 'Ole'.

This tree can be build using the composite design pattern [2]. Using this design pattern it is possible to model all **where** clauses. Such a tree can be built at compile-time using the two methods supplied by this package.

The interface contains the following methods.

- createPredicate(<attribute name>, <checkValue>, <op>) **return** compositePredicate
- combinePredicates(<compositePredicate>, <compositePredicate>, <op>) **return** compositePredicate

Explanation The createPredicate method makes it possible to create predicates, and it has three parameters. <attribute name> is the name of the attribute to be used for comparison. <checkValue> is the value the attribute is compared to. It should be noted that this can also be another attribute. Lastly, the parameter <op> is the operator used in the comparison, e.g., the equality operator. The method returns a compositePredicate, which maps to a where-clause condition, e.g., name = 'Hans'.

Since the attributes of concepts can have many different types <checkValue> is passed as a string. Then the DBMS implicitly converts the string to the required type. <attribute name> is also passed as a string. This means that the compiler is unable to check that the string contains a valid attribute name. In order to make this check possible, a string constant with the name of the attribute is created for each attribute of a concept. E.g., for the student concept a constant named cpr exists, representing the string 'cpr'. Then the application programmer can use these constants when requiring attribute names, making the compiler able to detect typos and invalid attribute names.

The same idea is used for the <op> parameter. However, the operator constants reside in the predicate package, as they are not associated with any single concept. The operator constants are shown in Table 5.

opAnd	and
opOr	or
opEq	equal
opNeq	not equal
opLt	less than
opGt	greater than
opLte	less than or equal
opGte	greater than or equal

Table 5: The semantics of the predicate package's constant to operator mapping

The other method of the predicate package is combinePredicates. This method makes it possible to combine predicates, in order to form larger predicates. The method has three arguments. The two first is the two predicates to be combined, and the last is the operator to use. Note that it is only possible to conjunct and disjoint predicates, no other operators are allowed.

Considerations The idea behind the predicate package can easily be used to model other clauses than **where** clauses, e.g., **from-**, **order by-**, **group by-**, and **having**-clauses. The idea of using string constants to enable compile-time syntax-checking is somewhat inelegant, but chosen for the API to remain general. Specific programming language constructs can be utilised instead, e.g., inheritance in object oriented languages with a general attribute class with specific attributes as subclasses, or PL/SQL types and subtypes.

3.4.13 Count Interface

The count interface offers two methods both called count. A count method that counts the number of elements without considering any predicate, and a method that does consider a predicate.

The interface contains the following methods.

- count() **return** integer
- count(<compositePredicate>) **return** integer

Explanation The plain count method executes the **select** count(*) SQL statement. This simply results in the number of records in a concept.

The other way to use count is by adding a predicate as the method's in-variable such that count(<compositePredicate>) would be the method call. The predicate allows for **select** count(*) statements with a **where** clause explicitly defined by the predicate. As an example one might want the number of all students whose names are "Hans" or "Ole". In other words, the SQL-statement in Listing 6 must be executed.

```
1  select  count(*)
2  from student
3  where name = "Hans"
4  or  name = "Ole"
```

Listing 6: A count SQL statement containing a **where** clause.

Lines 1-2 counts the number of records in the *student* concept. Lines 3-4 indicates that only records containing either 'Hans' or 'Ole' are counted. The equivalent count method to the SQL statement in Listing 6 is shown in Listing 7.

```
1  function count(predicate) return integer
2  begin
3    select  count(*)
4      into result
5      from student
6      where predicate.toString()
7     return result ;
8  end;
```

Listing 7: Implementation of the count with predicate method.

Lines 3-5 counts the number of records in the *student* concept. In line 6 the predicate is turned into a string equivalent to the **where** clause in line 3 and 4 in Listing 6. The process of building a predicate is facilitated by the predicate package as explained previously in Section 3.4.12. To implement the count method in Listing 6 a predicate must be devised. This predicate is devised in Listing 8.

```
1  procedure example is
2    compositePredicate hansPred;
3    compositePredicate olePred;
4    compositePredicate combiPred;
5    integer result ;
6  begin
7    -- assign values
8    hansPred =
9    createPredicate(student.name,"Hans",opEq);
10
11   olePred =
12   createPredicate(student.name,"Ole",opEq);
13
14   -- combine predicates
15   combiPred =
16   combinePredicates(hansPred,olePred,opOr);
17
18   -- run the count method with the
19   -- combiPred as in-variable.
20   result = student.count(combiPred);
21 end;
```

Listing 8: Sample use of the count with predicate method.

In lines 2-5 the variables are declared. In lines 9 and 10 the first predicate is devised. The HansPred variable is equivalent to the name = 'Hans' part of the **where** clause. Similarly, in lines 11 and 12 the name = 'Ole' part of the **where** clause is devised. Lines 15 and 16 combines the two predicates using the combinePredicates method. Thus, the complete **where** clause is created.

Here the SQL has been completely encapsulated, a high level of compile-time error detection is achieved, and hence; possible runtime-errors are reduced.

Considerations The predicate package provides compile time error detection when building any of the where-based syntax clauses. However, this **where** clause encapsulation comes at a price. The four-line SQL statement in Listing 6 are replaced by the PL/SQL code fragment in Listing 8. In general, the predicate approach calls for more implementation effort. In fact, the addition of one condition to the **where** clause yields the creation of two new predicates. This results in $2n - 1$ predicates needed to represent an SQL **where** clause with n conditions.

Conceptually, the count interface also belongs to the general interface as this interface does not manipulate data in the underlying concept.

3.4.14 List Interface

The list interface is used to retrieve a list of records. It offers three methods to facilitate this. One method to do plain queries, and one to use the **select** operator with a **where** clause. The interface contains the following methods.

- list () **return** list of records
- list <attribute name>()
  return list of <attribute value>s
- list (<compositePredicate>) **return** list of <record>s

Explanation Similar to the two count methods implemented in the count interface, the list interface uses the predicate package to form the **where** clause. The implementation is quite similar to that shown in Listing 7. The only difference is that list returns a list of records, instead of the number of records.

Considerations The similarity between the count and the list method allows for implementation-reuse. Once a predicate has been declared, it can be used either by list or by count to either return the list of records that satisfies the **where** clause or to simply return the number of records that satisfies the predicate in the particular concept.

The list interface also provides locking facilities in the same manner as the attribute get/set and general interface. However, it is important to remember that enabling locking and calling the list method, is equivalent to performing a table lock which could result in the disadvantages described in Section 3.4.11.

3.5 Dynamic Design

This section describes the get/set and the key conversion interfaces that illustrate the difference between a static and a dynamic design and implementation. It will focus on the benefits and disadvantages on each of the designs. The dynamic API contains the same interfaces as the static API. The interfaces not described in the section are available in Appendix A.2. As with the static API design, some details have been omitted from the discussion, all the details are available in Appendix A.2.

3.5.1 Attribute Get/Set Interface

As described in Section 3.4.1, one could choose to design the get/set interface statically. However, in this section a dynamic approach is presented. The dynamic get/set attribute interface consists of two methods.

- get(<attribute name>, <record id>)
  return <attribute value>
- **set**(<attribute name>, <record id>,
  <attribute value>)

Explanation These methods look very similar to those of the static API in Section 3.4.1. The only difference is that the <attribute name> is now a parameter and not a part of the method name. The method names are intuitive. The get method retrieves the <attribute name> identified by the <record id>. The **set** method sets the <attribute value> to the <attribute name> identified by <record id>. Listing 9 shows how to use the dynamic get method for retrieving the *name* attribute from the *student* concept.

```
1  function get(attributeName, recordId)
2   return varchar is
3    result  varchar
4  begin
5    select  attributeName
6      into  result
7      from student
8     where sid = recordId;
9     return result ;
10 end;
11
12 attributeName = 'name';
13 recordId = 1;
14
15 get(attributeName, recordId);
```

Listing 9: Implementation of how to use the dynamic get method on the *student* concept.

In lines 1-10 the *name* attribute is retrieved and returned. It is important to note that the name of the attribute needed is sent as a parameter and the result always is cast to a varchar. In lines 12-13 the variable needed is set and in line 15 the get method is called.

Considerations A benefit of the dynamic version of the get/set interface is that it reduces the number of methods compared to the static version. However, the risk of exceptions increases. For instance, the application programmer may declare an invalid type of <attribute value> for the <attribute name> that he is trying to set. Furthermore, the application programmer may try to either get or **set** an <attribute name> that does not exist; hence, also invoking an exception.

3.5.2 Key Conversion Interface

In the dynamic version of the key conversion interface, the candidate keys are no longer hard wired to the method names it provides. Instead, the dynamic key conversion interface provides only one method that is listed below.

- key2key(<key in>,<key value>,<key out>)

Explanation The key2key method takes three arguments. The <key in> describes the name of the known candidate key where <key value> is the value of this key. <key out> is the name of the desired candidate key. Listing 10 shows how to use the dynamic key2key method for converting the *cpr* attribute to *sid* attribute in the *student* concept.

```
1  function key2key(keyIn,keyValue,keyOut)
2   return varchar is
3    result  varchar
4  begin
5    select  keyOut into result
6    from student
7    where keyIn = keyValue;
8    return result ;
9  end
10
11 keyIn = 'cpr';
12 keyValue = '1505801357';
13 keyOut = 'sid';
14
15 key2key(keyIn,keyValue,keyOut);
```

Listing 10: Implementation of the key2key method on the student concept.

In lines 1-9 the new keyOut is found according to the keyIn and the keyValue. It is important to note that the name of the known key as well as the name of the desired key is given as a parameter. Furthermore, both the keyValue and the result has to be upcasted to a varchar. In lines 11-13 the variables needed is set and in line 15 the key2key method is called.

Considerations This interface provides a much more flexible way of doing key conversion. However, it comes with the price of possible run-time errors. The application programmer may pass invalid <attribute name>s for keys. Also, it requires that the programmer knows exactly which <attribute name>s that are candidate keys. In addition, the dynamic

way is also more schema independent. The static version is not. For example, to generate the static API the attribute names are needed in order to create the method names. This is not needed when generating the dynamic API.

3.6 Evaluating design criteria

This section will evaluate the design criteria from Section 3.3. Each of the criteria will be discussed with respect to both the static and the dynamic API.

Comprehensive Method names for both the static and the dynamic API are designed to be intuitive for the developer. This means that the developer can guess the functionality of a method simply by looking at the name. As it was shown in Section 3.5 the get/set and the list interface use different naming conventions in the two API versions. The fact that the attribute name is not incorporated in the method name in the dynamic API, means that developer has to know what attributes exists on a given concept. This could make the dynamic API less comprehendible compared to the static API.

Auto-generation In order to generate the static API, access to meta data is necessary. For instance, the return type of the methods in the get/set interface are found using meta data. When considering the get/set interface in the dynamic implementation, only meta data about primary keys is needed. As described in Section 3.7 the return types in the dynamic API are cast to the varchar type. Thus, when the developer utilized the dynamic API type information will be lost.

Symmetric Most methods with exception of the methods in the record lock and table lock interfaces have a symmetric counterpart. This means that if the developer can output something, then he can input it again. As described in Section 3.3 this also improves the comprehensibility of the APIs. As described in Sections 3.4.10 and 3.4.11 no counterpart to the recordLock and lockTable exists. To unlock a record or a table the developer will have to commit or rollback the current transaction.

Orthogonal Both the static and the dynamic API have been implemented to avoid overlapping functionality between methods. However, one could argue that the recordGet method overlaps with the methods provided by the get/set interface. Since developers used to the object-oriented paradigm are accustomed to get and set methods that works attribute level, the existence of the get/set interface aids both the comprehensibility of the API as well as the support for frequent use.

Support frequent use As described in Section 3.3 the APIs are designed to provide the most often used functionality. To achieve this goal the design has been validated through visits to two danish software houses, Atira ApS and Logimatic Software A/S.

Modular Both design and implementation of the APIs are highly modular. This enables users to generate just the interfaces needed.

3.7 Comparison

In this section we are going to look at the differences from the static to the dynamic API. The differences that will be discussed are listed in Table 6.

Static SQL	Dynamic SQL
Compile time errors	Run-time errors
No type casting	Explicit type cast
More schema dependent	Less schema dependent
Many methods	Fewer methods
Specific methods	Generic methods

Table 6: Differences between static and dynamic SQL

A major difference between static and dynamic SQL is when errors are caught. When using the static interface many errors will be found on compile time. As an example consider the *teacher* concept from Figure 1. If a programmer by accident call the method getFlame, that does not exist, instead of the getFname method, a compilation error would occur. Consider this example using the dynamic API, now he calls the get method specifying that he wants the 'flame' column. As shown in Figure 1 the flame attribute does not exist, however, this error will not surface until run-time.

Considering the return type of the get methods in the dynamic API a risk for errors occur. When calling, for example, the static getCPR method on the *student* concept, the programmer knows that the method returns an integer. When using the static API the return type of a method is always known. This is not the case with the dynamic API. As the dynamic API only provides a single get method used to retrieve attributes, all values have to be upcasted to a general value, e.g., a varchar in the PL/SQL programming language or an Object [7] in the Java programming language.

Obviously, the static API requires more information about the underlying schema than dynamic API. For example, in order to create the static attribute

get/set interface all attributes in a concept have to be known. The dynamic attribute get/set interface only requires information about the primary keys of the underlying schema, the same situation applies for the key conversion interface. The schema independence increases the risk for run-time errors due to the loss of type information and the ability to specify false attribute names.

As a result of the nature of dynamic SQL, the dynamic API has very few generic methods opposed to the static API that has many specific methods. Again, the attribute get/set interface provides a valuable example. The implementation of the attribute get/set interface on the *teacher* concept, yields a total of five get and five **set** methods (excluding a get and **set** method for the <record id>), whereas the same interface in the dynamic version always provides exactly one get method and one **set** method.

3.8 Views

In this section the API treatment of views will be described. Three types of views are considered. Views that are read-only, views that can be updated but does not allow insertions, and views where it is possible to both update and insert rows. These three types of views are the ones that can occur and therefore the ones that must be handled. The read-only views are handled by only allowing methods that does not update or insert rows in the database. The views that can be updated are handled by excluding methods that inserts rows into the database. The views where both updates and insertions of rows are possible are handled as an ordinary table.

The challenge is to classify as one of the three types described above. Currently, the user must supply the information needed to determine which category the view belongs to.

4 An Elaborated Example

Most companies value business rules to be of utmost importance. This section illustrates how to implement business logic using the static API and our sample university schema. Below, two examples of imaginable business rules are presented. However, implementation details are only provided for one of them.

Business rules in our example university schema described in Section 3 could be something like "The database course on a given semester must have at least 20 students participating" and "a teacher may teach no more than three courses on a given semester". These rules are implementable using object-oriented or procedural programming language and the APIs. A possible solution to the first rule is shown in Listing 11.

```
1  begin
2    comp1 := predicate.create_predicate(
3      'semid',1, predicate.OP_EQ);
4    comp2 := predicate.create_predicate(
5      'coursename','database',predicate.OP_EQ);
6    composite := predicate.combine_predicates(
7      comp1, comp2, predicate.OP_AND);
8    cid_tab := courseinterface.list_cid(composite);
9    for i in cid_tab.first .. cid_tab.last loop
10     comp3 := predicate.create_predicate(
11       'cid', cid_tab(i), predicate.OP_EQ);
12     stud_number := participantinterface.ccount(
13       comp3);
14     if (stud_number > 20) then
15       print('Sufficient students');
16     else
17       print('Insufficient students');
18     end if;
19   end loop;
20 end;
```

Listing 11: Business rule example: The database course on a given semester must have at least 20 students participating.

In lines 2-7, a predicate corresponding to the SQL **where** clause semid = 1 **and** coursename = 'database' is created. The predicate and the list method on the course interface is then, in line 8, used to retrieve all the relevant *cid*s from the *course* concept. The loop in lines 9-19 creates a new **where** clause predicate and it is used in the count method on the participant interface. The result is then used to decide if there are a sufficient number of students.

In the above manner, business rules can be enforced using the API and some object-oriented or procedural code.

The above example was created using the static API. Naturally, it is possible to create the same example using the dynamic API.

5 Implementation Issues

In this section important implementation issues are described. To the best of our knowledge, there exists no related work that explains the implementation difficulties encountered during their development process. Initially, a choice had to be made between an object-oriented and a procedural programming language. Furthermore, problems experienced with database permissions are discussed. The implementation is performed in PL/SQL on the Oracle 9i platform. The experiences developing the APIs using the PL/SQL programming language are presented, and we discuss if choosing another programming language would have been more beneficial.

Object-oriented versus procedural paradigm

The design aimed at being programming

language independent in respect to the object-oriented paradigm and the procedural paradigm. However, parts of the APIs could more easily have been implemented in an object-oriented language, e.g., the predicate package in Section 3.4.12. This package is implemented using the composite design pattern that is an object-oriented pattern [2]. However, it is not easily implemented in a procedural language like PL/SQL; yet, it was still possible to use PL/SQL types to successfully implement it. This is mainly because PL/SQL types resemble object-oriented language constructs. More specifically they support inheritance.

Permissions It is important to note that when using PL/SQL and in particular the utl_file package [11], files are written on the server, instead of on the client. In other words, the application programmer needs to have write permission on the database server. Permissions in general can be an issue, for instance in the table lock interface.

Database Platform Although the design of the APIs is generic the actual APIs are implemented on the Oracle 9i database platform. Due to the advantages of the generic design of the APIs they are implementable on other database platforms as well.

PL/SQL implementation As mentioned above, the database platform used was the Oracle 9i. Therefore, it was very appealing to use Oracle's own programming language PL/SQL. Except for the problems during the implementation of the predicate package, we encountered no problems using PL/SQL. By using the %**type** language construct it is possible to completely avoid the type impedance mismatch problem. If the API were implemented in another programming language, e.g., Java there would be a mismatch between the types used by the DBMS and the types used in the programming language. For example the Oracle DBMS has several types to represent a string, e.g., varchar, varchar2, nvarchar, or clob types, whereas the Java programming language is able to encapsulate all of these types in the string type.

6 Auto Generation of the API

Auto generation of the API is time saving for the developer, since he is spared developing the API from scratch. Furthermore, the API is resistant to schema changes, as the API only has to be regenerated. Therefore, the API is applicable to different schemas instantaneously. This section describes the design of our API generator. Initially, the core components that are used to represent the API code are described. Hereafter, the overall customization process is described. This generator generates PL/SQL code for the Oracle platform and is implemented in C#.

If the generator was to produce source code in another language than PL/SQL, at the very least all toString methods in the components would have to be re-implemented. The toString methods are the only methods that produce source code. The remaining methods are only used internally in the generator. Furthermore, components for language specific constructs should be added. Currently, the generator supports for the generation of all interfaces except the interfaces listed below.

- Output interface
- List interface
- Modification interface
- Record lock interface
- Table lock interface
- Input interface

With an additional effort these interfaces could easily be implemented.

6.1 The Components in the Generator

In this section each component of the generator will be described. A UML diagram of the generator is depicted in Figure 4. These components are designed to represent an entire PL/SQL package, including all the constructs of a package; procedures, functions, variables, constants, type, and subtype declarations.

DatabaseObject The DatabaseObject is an abstract superclass representing any object stored in the DBMS. The class contains a name describing the object, and a comment attached to that particular object. All other classes inherit from DatabaseObject.

Class The class Class represents a PL/SQL package. The class contains lists of all objects required to build a package. This includes interfaces, variables, constants and type declarations. The toString method on this class produces all PL/SQL source code for a package, i.e., both the header and the body of the package. More precisely the toString method iterates through each of its lists and calls the toString method on each of the objects in the lists.

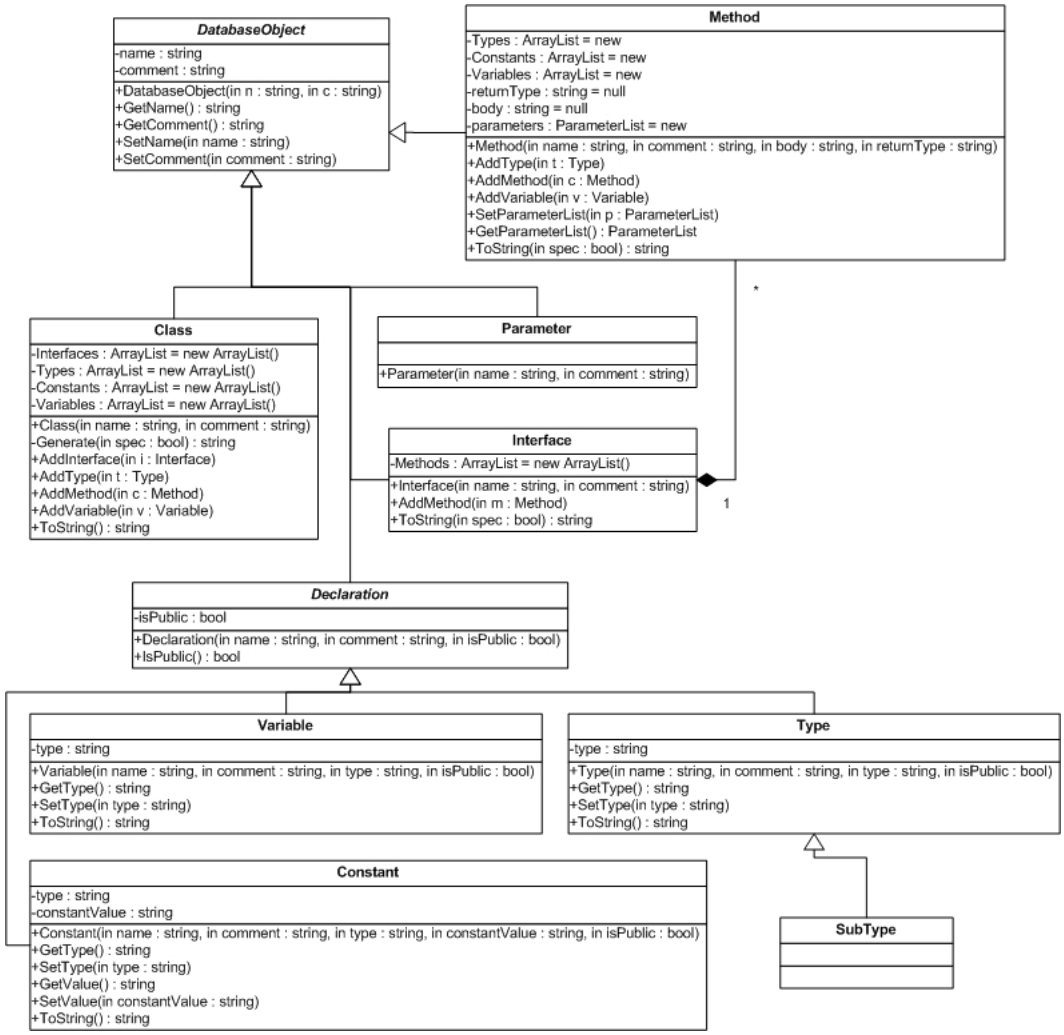

Figure 4: UML diagram depicting the API generator.

Interface The Interface class represents for example the general interface, that was described in Section 3.4.4. As shown in Figure 4, the Interface class has aggregation to the Method, Types, Constants and Variable classes. Note that the interface notion does not exist in the PL/SQL language. In the generator, this class merely functions as a container for methods in an API interface. If the generator was to generate an object-oriented language this class would be necessary.

Method The Method class is used to represent a PL/SQL procedure or function. The class has aggregations to Type, SubType, Variables, Parameters, and Constants. The return type of the method is given as input to the constructor of the Method class. The body of the method is setup through the constructor of the class. An alternative way of specifying the body could have been to model this through classes too. However, this would require a large number of additional classes to represent control structure and flow control mechanisms and, therefore, it was not chosen.

Declaration The Declaration class is an abstract super class. The Variable, Constant and Parameter classes inherits from this class. Common for these classes is that they are represented by a name, a comment, and a type. The name and comment attributes are inherited from DatabaseObject. The Declaration class provides the type attribute.

Variable This class is used to represent a variable. A variable might be initialized with a value; therefore, the constructor of Variable is overloaded in order to handle this case. As a variable can be declared in a package header, a package body, or in a method body the attribute accessModifier is added. The possible values of this variable are public, protected, and private, indicating where the attribute should be declared. In PL/SQL public corresponds to declaring the variable in the package header. A protected variable is declared in the package body. Thus, the variable is only accessible internally in the package. Finally, a private variable is declared internally in a method.

Constant The Constant class is used to model constants. The difference between a variable and a constant is that a constant has to be initialized to some value.

Parameter A parameter is a variable without an initialization. The class represents the parameters given as input to a method.

Type The Type class is used to represent a PL/SQL type. For example a **table** of students defined from the *student* concept: **type** stud_tab **is table** of student%**rowtype**. As with the Variable and Constant classes the Type also have an accessModifier attribute, that describes where the type is to be defined.

SubType This class represents a PL/SQL subtype. For example a *student* record defined from the *student* concept: **subtype** stud_rec **is** student%**rowtype**. The SubType class also has an accessModifier to that specifies where the SubType should be declared.

In the following section the customization possibilities of the API is described.

6.2 Customization

An important aspect of auto generating the API is being able to customize what is actually generated, tailoring the API to specific needs. For example, some administrators might not want to include the table lock interface due to potential performance problems. Other administrators would like to exclude other interfaces, or even specific methods due to security considerations, company policies etc.

The generator utilizes an XML file for configuring what is generated. By default, all interfaces are fully generated for each concept. If another behavior is wanted it must be specified for each concept though the XML file. As the number of concepts grows, this task becomes increasingly time-consuming. Therefore, it is reasonable to permit reconfiguring the default template for the concepts to a specific template matching the overall API requirements. Thus, it is only necessary to define special requirements for specific concepts, greatly reducing the amount of work needed to configure the generation of the API.

7 Performance Study

This section compares the performance of the static and the dynamic API. Initially, the both the technical and test setup are presented. Hereafter, the actual tests performed are discussed in detail. The tests consist of comparing the auto generated get and list methods of both APIs. Finally, the overhead imposed by the API is discussed. These methods are chosen to have both data vague and data intensive methods.

7.1 Setup

This section presents the technical setup of the tests.

The database in use is an Oracle9i Enterprise Edition Release 9.2.0.2.0, running on a PC with dual 466MHz Intel Celeron processors with 384MB of RAM using the Microsoft Windows 2000 professional operating system. As mentioned in Section 5 the implementation of the APIs is done in Oracles PL/SQL language. During testing only the user performing tests is using the PC.

The test is conducted on a single table. This table has 18 columns and contains 50.000 rows. Each row takes up an average space of 249 bytes. The types of the columns are numbers, dates, and varchars. The primary is indexed.

Two tests are performed. Before each test the database is restarted to clear any cached data, and to ensure each test has the same preconditions. Moreover, each test is run three times, and the resulting time is an average of the three.

The first test illustrates the possible performance gain achieved by knowing the SQL statement at compile-time. For this purpose, the get methods are used. Each method is executed 5.000 times with different <record id>s.

The second test compares the performance of data-intensive operations. Here, the list methods are used. Each method is executed 10 times. Every execution retrieves all rows from the table into main memory.

7.2 Performance Test

In Figure 5 the running times of the methods are shown. From the figure we see that the static get method is more than twice as fast as the dynamic get method. This clearly illustrates that there is a substancial advantage in knowing the SQL statement at compile-time. However, it should be noted that not all languages are capable of utilizing this advantage. For example, in the Java programming language every SQL query would degenerate to dynamic SQL, via the JDBC database driver.

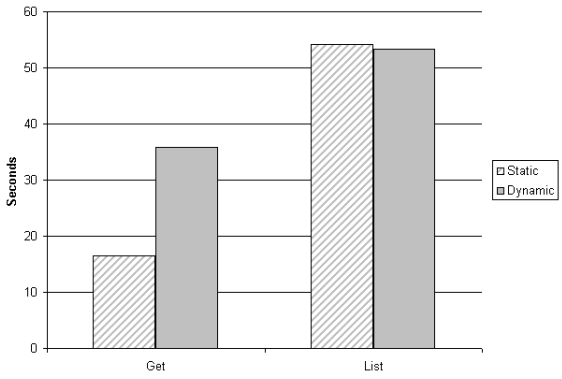

Figure 5: A comparison of the two APIs

From the figure we also see that the list methods perform nearly on par. The difference of almost one second is most likely due to small interferences from the operating system and background processes. Thus, for data intensive methods the possible performance gain of static SQL is negligible.

7.3 Evaluating API overhead

There are several factors involved when evaluating the overhead imposed by the API. As a minimum, an extra method needs to be called, compared to executing an SQL statement directly. Furthermore, methods like the dynamic list and count, that both take a predicate as input uses dynamic SQL. This makes it necessary to typecast the returned result. If the API is not used and the predicate is known at compile-time, languages like PL/SQL or SQLJ can take advantage of knowing the SQL statement at compile-time, which avoids type casts and allows for performance gains as discussed in Section 7.2. In other languages that use a database API like JDBC, this makes no difference as only dynamic SQL is supported.

8 Conclusion

In this paper, we presented and discussed the issues of designing and implementing a generic API. Furthermore, we presented an generator, that generates the database API in the PL/SQL language. The design of the API is generic and independent of the underlying DBMS, database schema and programming-language paradigm, i.e., the API is implementable in both an object oriented or a procedural programming language. Databases become more and more used, and the fact that most developers do not have SQL as a primary language, creates the need for a database abstraction layer, i.e., an API to allow easier development.

As described in Section 3.7 a benefit of the static API are that, compared to the dynamic version, many errors are caught at compile-time. Furthermore, the static API preserves type information, whereas the dynamic API always upcasts the return type to a more general type. On the other hand, the dynamic API offers fewer but more generic methods. Futhermore, the dynamic API is more resistant to changes in the underlying schema. This means that the dynamic API can withstand the addition or removal of attributes, as long as the <record id> is unchanged. Whereas the static API requires recompilation upon changes to the database schema.

As shown in Section 7 the static API is, in some cases, twice as fast as the dynamic API.

Future work includes several issues. The API does not offer special services the eases the job of adding additional interfaces or methods. The ability to easily add methods and interfaces would make it easier to use on existing applications. The generator should also be available for additional language. Furthermore, the generator is currently only capable of generation the static version of the API. The current implementation of the API generator does not offer a GUI for customising the API generated. This would minimize the effort needed to use the API generator.

Acknowledgements

We would like to thank System administrator Wladyslaw Andrzej Pietraszek and the Junta, from the Department of Computer Science at Aalborg University, for borrowing us their schema.

Furthermore, we thank Atira ApS and Logimatic Software A/S for their comments and suggestions to our initial API design.

References

[1] Raimund K. Ege, Yaman Battikhi, Philippe Pardo, Jinny Uppal, and Naphtali Rishe. A modular java API for object-oriented databases. In *COMPSAC '98 - 22nd International Computer Software and Applications Conference*, pages 55–60, 1998.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] Hibernate. http://www.hibernate.org. As of 31.05.2004.

[4] jGuru. Enterprise JavaBeans Fundamentals. http://java.sun.com/developer/onlineTraining/EJBIntro/EJBIntro.html. As of 31.05.2004.

[5] Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay Spinuzzi. Building more usable APIs. *IEEE Software*, 15(3), 1998.

[6] Sun Microsystems. Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/. As of 31.05.2004.

[7] Sun Microsystems. J2SE 1.4.2 API Specification. http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html. As of 31.05.2004.

[8] Sun Microsystems. The Java Tutorial. http://java.sun.com/docs/books/tutorial/java/interpack/interfaces.html. As of 31.05.2004.

[9] Udo Nink, Theo Härder, and Norbert Ritter. Generating call-level interfaces for advanced database application programming. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 575–586, 1999.

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

[11] Oracle. Oracle9i supplied PL/SQL packages and types reference. http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96612/u_file.htm. As of 31.05.2004.

[12] Catherine J. Tilton. An emerging biometric API industry standard. *IEEE Computer*, 33(2), 2000.

A APIs

The following sections present the static and the dynamic API for interacting with the database.
The APIs will be presented in the following format. Initially each a short description of the purpose of the interface will be given. Then each of the methods included in the interface will be described.
Each method description will contain the following: A method name, the method signature, a short description, the return value, pre- and postconditions and finally exceptions.

A.1 Static API

In this section the complete generic static API is presented.

A.1.1 Attribute Get/Set Interface

Description: The attribute get/set interface is used to retrieve single attributes from the database. A get and a set method is provided for each attribute in a record, e.g., `getName()` and `getPrice()`.

get methods
`get<attribute name>(<record id>) return <attribute value>`
`<record id>` is the unique identifier, equivalent to a primary key of a table. The get methods retrieves the value of the of the `<attribute name>` attribute identified by the `<record id>`. It should be noted that there are get and set methods for each attribute in a concept.
Returns
The `<attribute value>` of the `<attribute name>`.
Preconditions
`<record id>` not null.
`<record id>` exist in the database.
Postconditions
`<attribute value>` unchanged in the database.

set methods
`set<attribute name>(<record id>, <attribute value>)`
Sets the value of `<attribute name>` in the row identified by `<record id>` to `<attribute value>`.
Returns
Nothing.
Preconditions
`<record id>` not null.
`<record id>` exists in the database.
`<attribute value>` not null if declared mandatory in the database.
`<attribute value>` in the correct domain.
Postconditions
The value of `<attribute name>` is updated to the supplied `<attribute value>`.

A.1.2 General Interface

Description: The general interface provides a series of general methods. None of these methods alters the underlying database. The interface is inspired by the java.lang.Object class [7] known from the Java programming language. Additionally, a method for retrieving a single record is provided.

toString method
`toString(<record id>) return String`
Retrieves the `<record>` identified by `<record id>` and convert it to a string.
Returns
A string representation of a `<record>`, i.e., all attributes are printed.
Preconditions
`<record id>` not null.
`<record id>` exists in the database.

Postconditions The returned string is not null.

toString method
`toString(<record>) return String`
Converts a record to a string.
Returns
A string representation of a record, i.e., all attributes of the a record are returned.
Preconditions
`<record id>` not null.
`<record id>` exists in the database.
Postconditions
None.

exist method
`exist(<record id>) return boolean`
Checks if the record identified by `<record id>` exists in the database.
Returns
True if the record exists, false otherwise.
Preconditions
`<record id>` not null.
Postconditions The returned string is not null.

exist method
`exist(<record>) return boolean`
Checks if the record identified by the `<record id>`, in the `<record>`, exists in the database.
Returns
True if the record exists, false otherwise.
Preconditions
`<record id>` of the `<record>` is not null.
Postconditions
None.

equal method
`equal(<record id>, <record id>) return boolean`
Checks whether two records are equal, i.e., every attribute of the two records are compared.
Returns
True if the `<record>`s are identical, false otherwise.
Preconditions
`<record>`s are not null.
`<record id>`s exists in the database.
Postconditions
None.

equal method
`equal(<record>, <record>) return boolean`
Checks whether two records are equal.
Returns
True if the records are equal, false otherwise.
Preconditions
`<record>`s are not null.
`<record id>`s from the `<record>` exists in the database.
Postconditions
None.

clone method
`clone(<record id>) return <record>`

Makes a copy of the record. The <record id> is also copied, which means that the new copy cannot be inserted directly into the database. Since the keys are copied as well, the insertion of the newly created record would result in a duplicate key error.
Returns
A copy of the record given to the method.
Preconditions
<record>s are not null.
<record id> exists in the database.
Postconditions The returned <record> must be equal to the original <record>.

clone method
clone(<record>) return <record>
Makes a copy of the record. The <record id> is also copied, which means that the new copy cannot be inserted directly into the database. Since the keys are copied as well, the insertion of the newly created record would result in a duplicate key error.
Returns
A copy of the record given to the method.
Preconditions
<record>s are not null.
<record id> from the <record> exists in the database.
Postconditions The returned <record> must be equal to the original <record>.

Count method
count() return integer
Counts the number of rows in the underlaying concept. This method is good for post-condition checking in the modification interface.
Returns
Integer, the number of rows in the underlaying concept.
Preconditions
None.
Postconditions
None.

Count with predicate method
count(<predicate>) return integer
Counts the number of rows in the underlaying concept that satisfy the given predicate.
Returns
Integer, the number of rows in the underlaying concept.
Preconditions
None.
Postconditions
None.

get method
get(<record id>) return <record>
Gets the <record> identified by the <record id>.
Returns
The <record> identified by <record id>.
Preconditions
<record id> not null.
<record id> exist in the database.
Postconditions
None.

A.1.3 The Type Interface

Description: Data of different type is often stored in a single table. Then a type attribute is used to indicate which type the data represents. The type interface provides special access to this information.

Naturally, the same functionality could have been achieved by using the attribute get/set interface. If one chose to implement the get/set interface, then it is not necessary to implement the type interface.

get method
`getType(<record id>) return <record type>`
Gets the `<record type>` of the `<record>` identified by `<record id>`.
Returns
The value of the type field.
Preconditions
`<record id>` not null.
`<record id>` exists in the database.
`<record type>` exists on the concept.
Postconditions
None.

set method
`setType(<record id>, <record type>)`
Sets the value of the type attribute in the row identified by `<record id>` to `<record type>`.
Returns
Nothing.
Preconditions
`<record id>` not null.
`<record id>` exists in the database.
`<record type>` exists on the concept.
`<record type>` is in the correct domain.
Postconditions The type attribute is updated.

A.1.4 The Status Interface

Description: The status interface is similar to the type interface, in that it provides special access to special attributes, in this case the status attribute. As with the type interface, it is optional to implement this interface, as the same functionality can be achieved by using the standard attribute get/set interface.

get method
`getStatus(<record id>) return <record status>`
Gets the `<record status>` of the record identified by `<record id>`.
Returns
The value of the status attribute.
Preconditions
`<record id>` not null.
`<record id>` exists in the database.
`<record status>` exists on the concept.
Postconditions
None.

set method
`setStatus(<record id>, <record status>)`
Returns
Nothing.
Preconditions
`<record id>` not null.
`<record id>` exists in the database.
`<record status>` exists on the concept.
`<record status>` is in the correct domain.
Postconditions The type attribute is updated.

A.1.5 Output Interface

Description: The output interface is build on top of the general interface. In particular, the to_string method. The interface is used for printing information to various sources such as screen, file, a pipe, and so on. Non-terminals introduced:

<target> ::= <screen> | <file> | <table>
<format> ::= <header-value> | <name-value> | <xml> | <html> | <comma separated>

<header-value> example

```
id           name            address
-----------  --------------  ------------
1001         Kim             New Orleans
```

<name-value> example, in the next section we refer to this example as the employee example.

```
id:      1001
name:    Kim
address: New Orleans
```

<comma separated> example

```
1001,Kim,New Orleans
```

Print method
print(<record id>)
Prints the <record> containing <record id> to <target> using <format>. <screen> is the default <target> and <comma separated> is the default <format>.
Returns
Nothing.
Precondition
<record id> is not null.
<record id> must exist in the database table.
<target> is a valid target.
<format> is a valid format.
Postcondition
<record> is printed to <target> using <format>.

Print method
print(<record>, <target>, <format>)
Prints the <record> to <target> using <format>. <screen> is the default <target> and <comma separated> is the default <format>.
Returns
Nothing.
Precondition
<target> is a valid target.
<format> is a valid format.
Postcondition
<record> is printed to <target> using <format>.

Print method
print(list <record id>, <target>, <format>)
Prints the <record>s containing a <record id> in the list of <record id>s to <target> using <format>. <screen> is the default <target> and <comma separated> is the default <format>.

Returns
Nothing.
Precondition
<record id>s not null.
The <record id>s in the list of <record id>s must exist in the database table.
<target> is a valid target.
<format> is a valid format.
Postcondition
The list of <record>s is printed to <target> using <format>.

Print method
print(list <record>, <target>, <format>)
Prints the list of <record>s to <target> using <format>. <screen> is the default <target> and <comma separated> is the default <format>.
Returns
Nothing.
Precondition
<target> is a valid target.
<format> is a valid format.
Postcondition
The list of <record>s is printed to <target> using <format>.

A.1.6 List Interface

Description: The get interface is used to retrieve single <record>s or single <attribute value>s. In contrast the list interface is used for retrieving multiple <record>s or <attribute value>s.

By using the list interface the application programmers avoid having to declare most SQL cursors in the programming language such as Java, Python or PL/SQL. The most commonly used cursors are encapsulated by the list interface and directly available for the application programmer. Naturally, specialized cursors must still be defined and implemented by the application programmer.

List method
list() return list of <record>
Used to retrieve a list of all <record>s in the concept.
Returns
A list containing all the <record>s from the concept.
Preconditions
None.
Postconditions
list of <record> must corrospond to the contens of the underlying table.

List method
list<attribute name>(Boolean) return list of <attribute value>
Returns a list of <attribute values>s. If the boolean is set to true duplicates are eliminated.
Returns
A list containing the (possibly unique) <attribute value>s of the specified <attribute name> from the concept.
Preconditions
None.
Postconditions
list of <attribute value>s must correspond to the contens of the column named <attribute name> in the concept. If the boolean is true there must be no duplicates.

List method
list(<predicate>) return list of <record>

Returns a list of <records>s satisfying the given <predicate>.
Returns
A list containing the <record>s satisfying the given <predicate> from the concept.
Preconditions
None.
Postconditions
Each record in the list of <record>s must satisfy the <predicate> given.

List method
list<attribute name>() return list of <attribute value>
Returns a list of <attribute values>.
Returns
A list of all <attribute values> from the concept.
Preconditions
None.
Postconditions
None.

A.1.7 Modification Interface

Description: The modification interface provides methods for manipulating the underlying schema, by means of insert, update, and delete methods.

Insert method
insert(<record>) return <record id>
Inserts the record, and returns the new <record id>to identify the inserted record.
Returns
Returns the new <record id> of the inserted record, possibly a automatically incremented integer value.
Preconditions
The <record> must satisfy all constraints.
<record id> is not already in the table.
The table is updateable, i.e. not locked.
Postconditions
The <record> is inserted

Update method
update(<record>)
Updates the <record> in the database identified by the <record id> in the specified <record>.
Returns
Nothing.
Preconditions
The <record> is updateable and not locked.
The <record id> exists.
The <record> satisfy all constraints.
Postconditions
The <record> has been updated.

Update method
update(<record id>, <record>)
Updates the <record> identified by the specified <record id> with the new record given as the second argument Note that this can also be used to update the <record id>.
Returns
Nothing.
Preconditions
The <record> is updateable and not locked.
The <record id> given as the first argument exists.

The <record> satisfies all constraints.
Postconditions
The <record> has been updated.

Delete method
delete(<record>)
Deletes the <record> identified by the <record id> of the supplied <record> from the database.
Returns
Nothing.
Preconditions
The <record id> exists.
The table is not locked.
The <record> is not locked.
Postconditions The <record> identified by the supplied <record id> of the <record> has been deleted.

Delete method
delete(<record id>)
Deletes the <record> identified by the supplied <record id> from the database.
Returns
Nothing.
Preconditions
The specifed record exists.
The table is not locked.
The <record> is not locked.
Postconditions
The <record> identified by the <record id> has been deleted.

A.1.8 Record Lock Interface

Description: The record lock interface is for record locking. The lock mode kan be either a shared lock or an exclusive lock.

Record_lock method
record_lock(<record id>, <lock mode>) return <record>
Locks the specified <record> identified by <record id> supplied in the given lock mode.
Returns
The locked <record>
Preconditions
The record exists.
The lock mode is not null.
The lock mode exists.
Postconditions
The <record> identified by <record id> in the database is locked according to the mode specified.

Can_lock_record method
can_lock_record(<record id>, <lock mode>) return boolean
Tests if the specified <record> identified by <record id> supplied can be locked in the given lock mode. It should be noted that this method only gives an indication of the possibility to lock the <record>. On succes, a subsequent call to the record_lock method can still fail.
Returns
Whether or not the <record> identified by <record id> could be locked.
Preconditions
The record exists.
The lock mode is not null.
The lock mode exists.
Postconditions

The <record> identified by <record id> in the database is not modified.

A.1.9 Table Lock Interface

Description: The table lock interface is for table locking.

Table_lock method
lock_table<table name>(<lock mode>) return boolean
Locks the table name according to the lock mode specified.
Returns
A boolean indicating whether or not the table could be locked.
Preconditions
The <table name> is not null.
The table exists.
The <lock mode> is valid.
The <lock mode> is not null.
Postconditions
Table is locked in the database in the <lock mode> specified.

Can_lock_table method
can_lock_table<table name>(<lock mode>) return boolean
Tests if the table can be locked in the given lock mode. It should be noted that this method only gives an indication of the possibility to lock the table. On succes, a subsequent call to the lock_table method can still fail.
Returns
Whether or not the table could be locked.
Preconditions
The <table name> is not null.
The table exists.
The <lock mode> is valid.
The <lock mode> is not null.
Postconditions
The <record> identified by <record id> in the database is not modified.

A.1.10 Key Conversion Interface

Description: The key conversion interface is used to convert the primary key to unique keys and vice versa.

id2UniqueKey method
id2<first unique key name>(<record id>) return <record first unique key>.
Converts the <record id> to a record of the <record first unique key>.
Returns
The record containing the first unique key
Preconditions
<record id> not null.
<record id> exist in the database.
Postconditions
None.

uniqueKey2Id method
<first unique key name>2id(<first unique key>) return <record id>.
Converts the <first unique key name> to the <record id>.
Returns
<record id>.
Preconditions

<first unique key> not null.
<first unique key> exist in the database.
Postconditions
None.

A.1.11 Input Interface

Description: The input interface is used to load data from files into the database. As we have an output interface we must also have an input interface to be symmetric.

InputFile method
input(<file>, <format>)
Bulk loads a file. The format is the same as defined in the output interface.
Returns
Nothing.
Preconditions
<file> not null.
<file> exist.
Postconditions
None.

A.1.12 Walker Interface

Description: The walker interface is used to get other <record>s from the <record> that is currently used. The Walker interface could, e.g., be used to retrieve the web of connected records using foreign key information.

getForeignConcept method
getForeign<concept>(<record id>) return <concept> type <record>
Retrieves the <concept> type <record> associated with the <record> identified by <record id>.
Returns
concept type <record>
Preconditions
Knowledge about foreign keys.
<record id> not null.
<record id> exist in the database.
Postconditions
None.

getConceptsForForeignConcept method
get<Concept>sFor<Foreign concept>(<record id>) return <concept> type <record>
Retrieves the list of <concept> type <record>s associated with the <foreign concept> <record> identified by <record id>.
Returns
list of foreign concept type <record>s
Preconditions
Knowledge about foreign keys.
<record id> not null.
<record id> exist in the database.
Postconditions
None.

A.2 Partial Generic Dynamic API

In this section the dynamic API is presented. As most interfaces are similar to those found in the section on the static API, only a few interfaces that illustrates the differences are included.

A.2.1 Attribute get/set Interface

Description: The attribute get/set interface is used to retrieve single attributes from the database. Two general methods are provided `get()` and `set()`.

Get method `get(<attribute name>, <record id>) return String`
Returns The `String` containing the value of the `<attribute name>` from the record identified by `<record id>`.
Preconditions
`<record id>` not null.
`<record id>` exist in the database.
`<attribute name>` must exist in the database.
Postconditions
The value of `<attribute name>` for the `<record>` identified by `<record id>` remains unchanged in the database.

Set method `set(<attribute name>, <record id>, <attribute value>)`
Sets the value of `<attribute name>` in the `record` identified by `<record id>` to `<attribute value>`.
Returns
Nothing.
Preconditions
`<record id>` not null.
`<record id>` exists in the database.
`<attribute name>` must exist in the database.
`<attribute value>` not null if declared mandatory in the database.
`<attribute value>` must be in the correct domain.
Postconditions
The value of `<attribute name>` have been updated to the value `<attribute value>`.

A.2.2 Key Conversion Interface

Description: The key conversion interface is used to convert a candidate key to another candidate key.

key2key method
`key2key(<key_in name>, <key value>, <key_out name>)` return `<key_out value>`.
Converts the value `<key value>` of candidate key `<key_in name>` to a candidate key `<key_out name>`.
Returns
A string containing the value of the converted candidate key.
Preconditions
`<record id>` not null.
`<record id>` exist in the database.
`<key_in name>` exist in the database.
`<key_out name>` exist in the database.
Postconditions
None.