USING SQL FROM SMALLTALK

2nd edition

Juhani Mäkelä

Helsinki 25.03.1991

Sarja C
Arkistonumero 1991/ . Erikoistyö

HELSINGIN YLIOPISTO
Tietojenkäsittelyopin laitos

HELSINGIN YLIOPISTO - HELSINGFORS UNIVERSITET

Tiedekunta/Osasto - Fakultet/Sektion	Laitos - institution
Matemaattis-luonnontieteellinen	Tietojenkäsittelyopin laitos

Tekijä - Författare
Juhani Mäkelä

Työn nimi - Arbetets titel
Using SQL from Smalltalk, 2nd edition

Oppiaine - Läroämne
Tietojenkäsittelyoppi

Työn laji - Arbetets Art	Aika - Datum	Sivumäärä - Sidoantal
Erikoistyö	25. maaliskuuta 1991	75 + liitteet 151

Tiivistelmä- Referat

Object-oriented programming techniques are slowly becoming more and more widely accepted especially in applications with a graphical user interface. Meanwhile, the development in local-area networks has brought the resources of effective relational-database-servers accessible to personal workstations. While these two technologies have been well studied at their own areas, there has been little effort to combine them. One reason for that might be the fundamental differences in the two environment's ways of handling and understanding data.

This paper is an attempt to derive some general outlines for database-intensive applications dealing with a relational database. The starting point is to define an application as a means of maintaining the dependencies and enforcing the semantic constraints of the data it operates on. The design is done in an object-oriented fashion.

For testing the suggested design a prototype application is implemented by using state-of-the-art hardware, SQLBase multi-user database server and Smalltalk/V PM as the programming language. The subjects of evaluation are Smalltalk as an application development environment and the feasibility of the suggested interface between an object-oriented progamming language and a relational database.

Computing Reviews (1987) classification: D.2.6, D.3.3, E.2, H.2.3, J.1

Avainsanat - Nyckelord
Object-oriented programming, relational databases, Smalltalk

Säilytyspaikka - Förvaringställe

Muita tietoja - Övriga uppgifter

TABLE OF CONTENTS

1. INTRODUCTION....1
   1.1. About the second edition....4
2. MAPPING OBJECTS TO RELATIONS....6
   2.1. The relational database-model....7
      2.1.1. Semantic constraints....9
      2.1.2. Dependencies....10
         2.1.2.1. Semantic dependency....10
         2.1.2.2. Functional dependency....11
         2.1.2.3. Inclusion dependency....12
         2.1.2.4. Multi-valued dependency....12
   2.2. The object-oriented data-model....13
      2.2.1. Active data....14
      2.2.2. The problem with object identity....15
3. PROGRAMMING IN SMALLTALK/V PM....18
   3.1. A programming example....18
      3.1.1. TextFileDifference....21
      3.1.2. TextLine....25
4. AN OBJECT-ORIENTED DATABASE INTERFACE....28
   4.1. The Database-class hierarchy....28
   4.2. Notation and terminology....29
   4.3. Tuple....30
   4.4. Relation....35
   4.5. Table and KeyedTable....37
   4.6. Table-subclasses....40
      4.6.1. Collecting data from tables....42
   4.7. Database....44
   4.8. The low-level connection....45
      4.8.1. External calls from Smalltalk/V PM....45
      4.8.2. The lowest level: dbEngine.dll....47
      4.8.3. The type conversion....49
      4.8.4. DbCursor....50
   4.9. An example call....51
5. IMAGE OF AN APPLICATION: THE USER INTERFACE....56
   5.1. Basic dialog elements....56
   5.2. Reusable classes....58
      5.2.1. Dictionary dialog....58
      5.2.2. Tuple dialog....59
      5.2.3. Modeless tuple dialog....59
      5.2.4. Table dialog....60

5.2.5. Table inspector....60
5.2.6. Report Window....61
5.3. Guarding inclusion dependences....62
6. A CASE STUDY: THE WORKTIME MANAGER....64
6.1. The environment and overall structure....66
6.2. Evaluating the application....67
6.2.1. Starting and how long it takes....67
6.2.2. Programming efficiency....71
6.2.3. Other programming aspects....71
7. CONCLUSIONS....73
8. BIBLIOGRAPHY....75
9. APPENDIX A: WORKTIME DATABASE....76
10. APPENDIX B: WORKING LOG....77
11. APPENDIX C: DATABASE CLASSES....81
11.1. Tuple....81
11.2. Relation....84
11.3. Table....89
11.4. KeyedTable....91
11.5. Database....94
11.6. DbCursor....95
11.7. DbEngineDLL....103
11.8. DbDataDescriptor....108
11.9. DbEngineConstants....109
12. APPENDIX D: USER-INTERFACE CLASSES....110
12.1. DictionaryDialog....110
12.2. TupleDialog....115
12.3. ModelessTupleDialog....117
12.4. TableDialog....118
12.5. TableInspector....121
12.6. ReportWindow....124
13. APPENDIX E: UTILITIES AND ENHANCEMENTS....127
13.1. TextLine....127
13.2. TextFileDifference....128
13.3. Changes to the standard environment....129
14. APPENDIX F: THE WORKTIME MANAGER....145
14.1. WorktimeManager....145
14.2. AccountTuple....149
14.3. AccountTable....150
14.4. AccountsDialog....152
14.5. AccountDialog....152

14.6. NewAccountDialog 153
14.7. AccountsInspector 154
14.8. TaskTuple 155
14.9. TaskTable 157
14.10. TasksDialog 159
14.11. WorkTuple 160
14.12. WorkTable 161
14.13. WorksDialog 164
14.14. ProjectTuple 164
14.15. ProjectTable 165
14.16. ProjectsDialog 167
14.17. MonthTuple 168
14.18. MonthDialog 169
14.19. ReportIntervalDialog 169
15. APPENDIX G: DBENGINE 171
15.1. The interface 171
15.2. The implementation 178
15.3. Test programs 204

1. INTRODUCTION

During last decade the object-oriented programming paradigm (OOP) has slowly made its way through to commercial software production. What started as a strictly special purpose simulation language (Simula), and for a moment got name as a radically different but hopelessly unpractical experimental language (Smalltalk), is now finally being adopted by mainstream programmers as the next great improvement after structured programming. Today it is very difficult to find any new software product which wouldn't have the "OOP" label attached to it in some way.

What actually makes a product object-oriented, then? For some, a user-interface with windows, buttons, menus and mouse seems to be enough. For others, the existence of classes and inheritance is the critical factor. Not surprisingly, as the idea has gained more popularity, many old-time programmers have discovered object-oriented features in almost anything they have been working with.

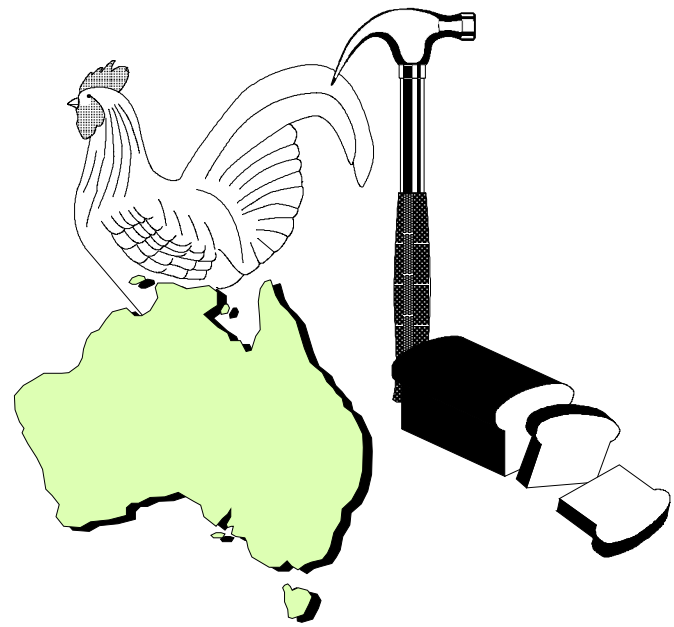
True object-oriented paradigm goes however deeper than just introducing a few new language constructs. The very essence of it is the aspiration to bring programming closer to reality by giving the programmer means for defining the program constructs as similar as possible to their counterparts in reality. While the traditional procedural style relies heavily on a small set of extremely reduced mathematical abstractions like integers, real numbers and character-strings, OOP makes it possible to use more domain-specific concepts. Ages in stead of integers, geographical distances in stead of real numbers and people's names in stead of character strings.

It is important to notice that a great deal of the code in traditional programs is usually aimed for implementing the natural constraints and dependencies of such domain-specific concepts. Nevertheless, such code tends to get scattered around the program if not totally neglected, which makes procedural programs difficult to manage. Everybody has heard stories about an 107-year old lady who receives an order to attend comprehensive school because somebody forgot to add the overflow-check somewhere in the program. In object-oriented programming these semantic patterns of behaviour have a natural and flexible organization as a protective shell around the actual data representation. This notion

of abstract data-types is in a central role in object-oriented programming.

The paradigm has proven to be quite a successful one as long as the whole system follows it. The problems begin when an object-oriented program starts to communicate with its environment, where different standards and paradigms still reign.

The controversy between it and the relational database-model is particularly clear. The relational model has been at least as successful in databases as OOP has been in programming. Nevertheless, it has a very different approach to data organization. Where OOP employs data abstraction, relational model uses a reduced set of entities at the attribute-level with no option to define own types. When OOP allows unlimited linking of objects to other objects to form hierarchical and graph-like structures, a two-dimensional array is the only data structure in the relational model. And while OOP suggests or enforces to encapsulate the actual data representation of an object and allows access to it only in a controlled manner through a user-defined interface, the relational model in its existing incarnations provides no means for expressing any non-trivial constraints or dependencies. The two worlds are truly separate in deed.



Name	Weight	Age	Population
"Clara"	0.570	0.9	1
"Australia"		1,000,000,000	17,000,000

"Hammer"	0.380	3	
"Bread"	0.200	0.00034	

OOP uses representatives of "things" or "objects" as the building blocks of a program while a relational database sees everything as tables of numbers and character strings.

The reason why relational databases are like that is the fundamental idea of keeping data apart from the ways it is used. A relational database consists of simple two-dimensional arrays called tables, which can be freely operated by a set of mathematical relations. The rational is the fact that changing the organization of the database is far too elaborate to do every time a new report or some other view to the data is needed. By disregarding most of the complexities of reality the model can offer methods of creating such views easily in a declarative manner, by help of a query language. The price that has to be paid is the loss in expressivity: a relational database knows very little about the entities it stores.

Nevertheless, when adding new data to a database it is important that constraints are not violated and that dependencies are retained. That's why all applications that use a relational database have to contain parts which make sure that only valid data is entered. According to my personal experience, such controlling modules form the main bulk of work in practical application development, especially because they are typically programmed in a relatively low-level host language like C or COBOL.

Object-oriented languages are worth of consideration for the job, because of their ability of expressing complicated relationships between objects in an explicit and well-organized way. This paper introduces one way of designing and implementing an interface to a relational database in a very object-oriented environment. This study has been carried out in the erikoistyö-group of Juha Vihavainen at Helsinki University during 1990. It has been ordered by and delivered to the Product Development of Nokia Edacom Oy.

1.1. About the second edition

An earlier version of this paper was published and distributed in last June. The object-oriented model of the relational database in it was quite inadequate in many respects. Since then, the model has been extended and improved, most of the existing classes have

been redesigned and re-implemented and new classes have been added. The solution is also used more intensively to test the capability of classes to bind together complex database constructs. As a result, many points of criticism in the previous edition no longer exist.

The user-interface parts have also been greatly improved. Consequently, the example application has turned from a micky-mouse-program into a more real-like piece of software with several dialogues, inspectors and reports.

Along with the changes in software, the document has been completely re-organized rewritten. It now contains a brief introduction to programming in Smalltalk. The weightpoint has been transferred from the implementation to the design, which is now presented in Smalltalk. The image of this document has been improved by adding pictures. Reported errors have been corrected and the language has been made easier to read and understand.

2. MAPPING OBJECTS TO RELATIONS

The purpose of a database is to store data for later usage and make it available for others to access and use. The instances that read data from the database and write new data into it are typically computer programs. For exchanging data with the database they have to map portions of its data-structures to their own local memory and perform read- and write-operations. In our case this means mapping objects in the program's object space to relations and tuples in the database and vice versa.

The co-existence of the two data-models - object-oriented and relational - is something very essential in this study. In older systems the program's and the database's data-models used to be quite alike: they used the same basic set of primitive types and similar methods of organizing data into records and arrays. The task of exchanging data involved mainly just copying blocks of memory from one buffer to another. Things have changed, however. Both the relational model and object-oriented programming languages have their own, highly abstract data-models which not only have very little in common but also more or less hide the actual organization of data. Therefore, a process for decoding the data from ones internal format into a common form and then coding it back into the other's internal format is needed. As a matter of fact, this is the case also when procedural languages are used: SQL's *tuple* cannot be directly copied into a C's *struct*; in stead the fields must be transferred one at a time, and very often also the representation of atomic values requires a conversion.

Although this decoding-coding scheme inevitably means some extra overhead at runtime and thus reduced performance, it has its good sides as well. It makes the programs very independent from a particular database product and also enable the communication of very different programs through the common database. In any case, a thick layer of software is needed between the primitive database-services and the application code. The properties of this layer are naturally dictated by the characteristics of the two data-models.

2.1. The relational database-model

As the name tells, the central concept in the relational database model is that of a **relation**, which tells whether a specific combination of key-value pairs exists in the database or not. A key

may be a column-name of an existing table or an expression derived from them, and the value connected to it is either an instance of one of the simple data-types (string, number or date/time -value) or **null**, which means no-value-at-all. A combination of such pairs is called a **tuple**, and its components **attributes.** The database consists of **tables**, which are unordered sets of tuples with the same keyset. Tables are relations, but new relations can also be defined by help of operations of the relational algebra [Ullm88, p. 55-57]. These operations are:

union A union of two relations R and S (R **U** S) is a relation which includes the tuples in R, in S or in both. R and S should have the same number of columns with compatible types.

set difference A difference of relations R and S (R **-** S) contains those tuples in R which do not belong to S.

cartesian product of R and S (R **x** S) is a relation which contains tuples formed so that each tuple in S is concatenated to each tuple in R.

projection Projection from R is a relation where the tuples are formed so that some of the keys of R are selected and of each tuple of R a new tuple is formed by taking the values connected to these keys.

selection Selection from R is a relation which contains those tuples from R which satisfy some logical condition.

These operations can be combined to expressions as usual. For example, the familiar concept of intersection of two sets can be defined like this: R **U** S **-** ((R **-** S) **U** (S **-** R)). Relations defined by such expressions are often called **views**. Therefore, there are two kinds of relations: tables, which do actually exist in the database, and views, which do not exist but are evaluated at request. Although the two are in some respects equal - for instance both can be used as operands in algebraic operations - there are also differences. Most notably, only tables can be written to; views are by nature read-only data.

In practice the database is not handled directly by help of the algebraic operations but through a query language, which is closer to natural language. For picking a specific item of data from the database the algebraic projection and selection are applied for instance like this: "get the value of the 'address'-field from 'employees'-table in the tuple(s) where the value of the 'name'-field

is 'Jones'". The standard query language with relational database is called SQL, and in its terms this query would look like this:

```
select ADDRESS
from EMPLOYEES
where NAME = 'Jones'
```

Notice that we cannot know whether there are none, one or many such tuples and the result of the query may hence be an array of 0-n items.

For an intelligent human user such a view suits well and comes quite close to what she intuitively wants to do. For a program, on the other hand, it is too unpredictable. That's why the application programmer interface for relational databases typically employ another, a more mechanic interface, which sees relations as two-dimensional arrays. Consequently, after evaluating a query like the one above, the program could enquire the number of rows in the result by one function call and then acquire the rows one at a time by another function call. Many tools provide also a method to access rows by a row-number.

The relational model is strongly value-oriented and declarative. It means that a user rather specifies *what* she wants from the database than *how* to get it and the underlying database engine is then supposed to find the fastest way to get the results. This mechanism has to nice results: for one, it makes very easy for a user to pick up just the data she is interested in without having to think how the database is physically organized. And on the other hand, it gives the database designer a freedom to choose the physical table decomposition so that redundancy is minimized without having to think how and for what the data is then used. Unfortunately, this works well only as long as reading from the database is concerned. When writing to it, all but the simplest views become obsolete and suddenly we have to be very aware of which attributes actually are located in which tables at the lowest level. Therefore, an application program has to know how to decompose a complex data-structure into physical tables when storing one into the database.

Another defect of the current query and data definition languages is that they provide almost none means for expressing constraints and dependencies between data elements, tuples and tables.

2.1.1. Semantic constraints

The constraints which can be expressed by SQL restrict mostly to simple type-checking, like forbidding to assign a string into an attribute which should be a number. The two other restrictions which can be set when defining a table is to say that an attribute may not have a null value and all attributes in a table must have unique values.

The use of an attribute imposes almost always much severe constraints, called semantic constraints. They reflect the fact that the few general types which are used for all purposes have much larger domains than a specific use would need. For instance, the range of possible date-values would span from the beginning of historical time to a couple of hundred years in future. On the other hand, when a date-attribute is used to represent a living person's birthday, no values outside the range from 1.1.1860 to today should be allowed.

Albeit simple, such constraints have to be checked against violations every time the database is modified. The relational model in general does not provide any means to enforce semantic constraints, even if some commercial SQL-products do [Fink90]. Furthermore, in many cases the constraint is much more intricate than just a simple range in a linear space. So, there should be a mechanism for attaching to each attribute a formula, a script or a portion of code which is activated each time the value of the attribute is changed, which checks that the given value is valid and if not, takes further steps to correct the state of things in a controlled and user-friendly manner.

2.1.2. Dependencies

The allowed values of attributes may also depend on the values of other attributes, either in the same tuple or in other tuples. These dependencies can be harshly divided to four groups: semantic dependency, functional dependency, multi-valued dependency and inclusion dependency.

2.1.2.1. Semantic dependency

Such dependencies may exist between attributes of the same tuple and they are due to the fact that the attributes are somehow semantically connected. In an ordering-invoicing system, the

article-tuple might have an attribute for telling the number of customers who have ordered the article and another which indicates the total amount of the article in orders. Now, if the first attribute is zero, the other must be zero too and vice versa. Or if a person's age is 9, her marital status obviously cannot be anything else but "single".

Such dependencies are strongly related to common sense, which has been considered unique to human beings only. That's because they are very often difficult to formulate and the border between valid and invalid values is hard to draw. Were the girl in the above example aged 15, it is no longer so definitely obvious that she couldn't actually be married. However, there should be no reason why also a computer system could not ask in such cases: "But this is most unusual, are You positively sure that there is no mistake?", while in the first case it could be steeper: "This surely cannot be so: either the age or the marital status of this person must be wrong!" Again, the more delicate and flexible programming methods are used, the more of the errors and mistakes can be detected as soon as they are entered.

2.1.2.2. Functional dependency

Another class of dependencies much easier to formulate is the one which dictates the value of an attribute according the values of that attribute in other tuples and other attributes in them. To be exact: a functional dependency between columns A and B in relation R means that if the values of A are equal in two tuples, then the values of B have to be equal in the same two tuples [Ullm88, p. 379-382]. For instance, imagine that we have a relation of goods and their suppliers:

Item	Supplier	Address
Hammer	Acme Inc.	West street 114
Nail	Acme Inc.	???

Now the address of the company has to be same in both cases, provided that each company has only one address and that the name of the company uniquely identifies it. Within these premises, the 'address'-attribute is functionally dependent from the 'supplier'-attribute.

Very often a relation has one or more attributes of which every other attribute is functionally dependent. Then those attributes form a **key** to the relation: when keys in two tuples are the same all other

attributes have to be the same as well and the tuples are identical. However, when there are also other functional dependencies, the table contains redundant data and integrity problems will arise. In the example above nothing prevents a user from entering different address values to different places. The problem can be solved by dividing the data in two tables like this:

Item	Supplier
Hammer	Acme Inc.
Nail	Acme Inc.

Supplier	Address
Acme Inc.	West Street 114

Such a process is called normalization, or rather one step of it. Beside solving the problem of insconsistency, it saves space and also sweat: if 'Acme Inc.' would someday move to East Street it is enough to update one tuple in stead of the hundreds in the first table.

2.1.2.3. Inclusion dependency

The normalization creates some new problems, however. By removing the undesirable functional dependency it establishes a new one, called inclusion dependency [Ullm88, p. 423]. It can be formulated like this: if attribute A in relation R has an inclusion dependency to attribute B in relation S, then if A has value X there must be at least one tuple in S where B has also value X. In other words: no item should have a supplier which is not mentioned in the suppliers-relation. Variations of the theme may further tell whether S should contain exactly one or more tuples with B as X. Thus, the clever trick of normalization has replaced problem with a new one, though a slightly easier to deal with. In the end it is the software's responsibility to ensure that no orphan - or dangling - tuples are entered to the items-relation. Also the changes in the supplier's tuple should be reflected in the items-tuple: if a tuple in the former is deleted or modified, those tuples in the latter should be deleted or modified accordingly.

2.1.2.4. Multi-valued dependency

There is yet another kind of dependency, called multi-valued one [Ullm88, p. 413-420]. It tells that if in relation R, which contains attributes a,b and c, there are tuples ab_1c_1 and ab_2c_2 there has to be also tuples ab_1c_2 and ab_2c_1 in it. The rationale for such a

dependency is usually the fact that attributes b and c come from a cartesian product of two relations and to each tuple there is a third value attached. For instance, let's assume that we have two suppliers which both provide a full-set of carpenter's tools. If we know that 'Acme Inc.' offers 'hammers' at 2.95$ and 'United Tools' 'nails' at 1.00$, there should also be an answer to questions how much are hammers at 'United Tools' and at what price does 'Acme Inc.' supply nails.

Removing multi-valued dependencies so that such questions would not remain unanswered is yet another step in the normalization-process. Not surprisingly, also here the cost is introducing new inclusion dependencies.

2.2. The object-oriented data-model

The relational database model puts a lot of weight on the uniformity of data items. All tuples in a relation must have the same attributes with equal meanings. This implies some restrictions to the diversity of data - even if we wouldn't store hens, continents and hammers into the same relation. There are always some aspects which just have to be set aside to make the infinitely complex reality to fit into our simple relation schemes. The whole idea of databases is the ability to handle large numbers of instances within uniform patterns and operations.

Nevertheless, this is where the object-oriented approach differs most from the relational scheme. In stead of common forms it emphasizes a close correspondence to reality. In its purest form this means that each entity in the real world must have exactly one counterpart in the program, and that this computerized model reflects and imitates the properties of it as completely as possible [Lieb86]. One part of the idea is that each of these simulating objects carries its own program code with itself and that the code applies only that object. Therefore, instances do not differ only in their data contents but also in the way they behave, though some common behaviour can be created by allowing the objects consult each other in responding to messages.

This extreme ideology of prototyping appears in a somewhat milder form in Smalltalk. There a class is the unifying factor among objects, so that all instances of the same class behave the

same way and have the same data contents. However, there are still two major differences to the passive-data concept of the relational model: the ideas of active-data and object identity.

2.2.1. Active data

The notion of active data means that an object cannot be referenced to and modified without it noticing. That is, we can only ask an object to tell some of its properties or perform some task, which possibly changes its data-contents. In Smalltalk, the contents of an object are protected so that they cannot be referenced directly. So, in the first place the programmer has an option to choose whether to allow access to them at all. And even if she does that, she still can do that whatever way she prefers and impose any constraints that are needed.

Active data is really not something characteristic only to object-oriented programming, but it can be found from the artificial intelligence field as well, where terms 'daemons', 'triggers', 'defaults' and 'virtual attributes' describe its various incarnations. A virtual attribute is a data field which really has no storage but is dynamically computed each time it is accessed. A 'daemon' or a 'trigger' is a portion of code which is automatically executed when a field is set or some other more general condition becomes true. A 'default' provides an assumption as a value of an attribute in lieu of a fact. This terminology comes from Winston [Wins84, p. 255-256], and as he points out, such clever presentation for constraining knowledge about data is necessity in complex problem spaces.

The notion of active data is however very much built-in the object-oriented programming. In a way, in Smalltalk all attributes are virtual since they all have to be accessed via a method dedicated for this. Consequently, all attributes are automatically guarded by daemons or triggers, and so on. One result of this is that object-oriented programming suits perfectly in artificial intelligence [Tell89].

2.2.2. The problem with object identity

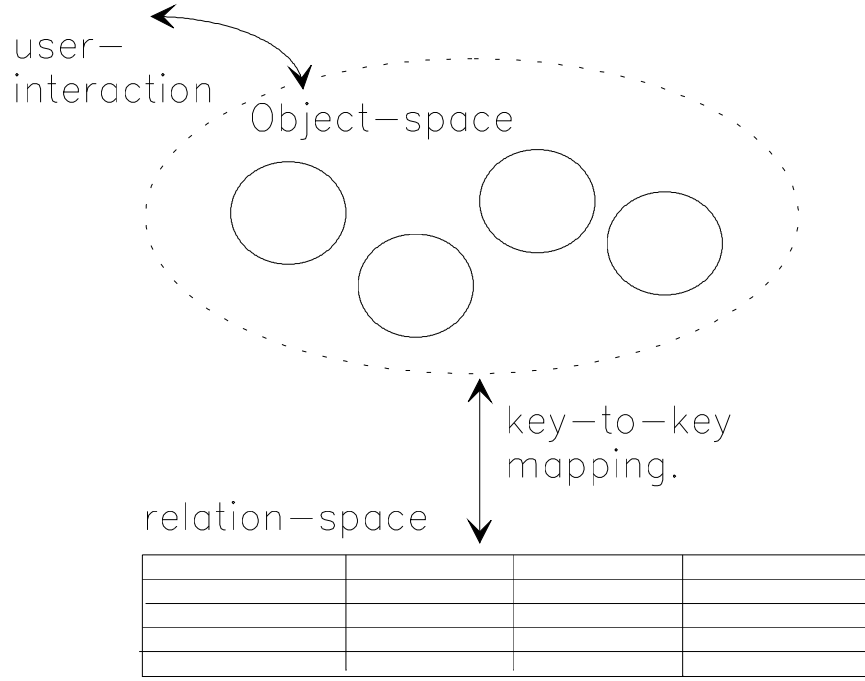
Object identity means that any object can be separated from other objects by some unique, inherent property of it. While the actual implementation of this "social security id" of objects is usually hidden from the programmer, anyhow, this means that an object can be referred as "this" without fear of confusion. It also means

that questions "are these two objects the same objects" and "are these two objects similar" can have different answers. The significance of this naturally varies depending on the class: any "1" is as good as some other "1", since integer numbers are pure abstractions. On the other hand, in a customer register two objects are definitely not the same even if they happened to contain the same name or birth-date.

The absence of object-identity seems to separate fundamentally the relational database from the memory space of an object-oriented program [Ullm88]. As mentioned, in a relation a tuple can only be identified as a combination of its attributes' values, and another tuple with the same values cannot be separated from the other. This difficulty is however mostly illusory, because also in the relational model the tuples usually have unique keys, especially in tables.

A table without an unique key, namely is difficult to handle. The identical rows are always changed and deleted all at once, and there is very little use for them being many. Even if this could be thought as some kind of a "bag"-model (a bag is a set where each element may occur more than once), in practice a simple counter attached to each row does the same trick with much smaller space consumption. Therefore, we could enhance the relational model by a subclass of keyed tables, with the additional constraint that each key value combination is unique and may thus occur at most once.

Also in the object-memory space the same restriction must hold good. That is to say, no two objects which are not the same can have the same key value. An object can be created according to the data in a tuple, and a tuple can be extracted out of an object. This way a one-to-one binding between objects in the memory space and tuples in the keyed tables of the database is established.



The objects in the memory-space are bound to tuples in the relation-space according to corresponding key-values.

The object-memory can be seen as the front-end work-memory or a tuple-cache of the database. When data is fetched from the database, intelligent active-data objects are created for each tuple in a keyed table. The modifications are written first to the object-space which then can be made consistent with the database at will. An object with a key value not resident in the table stands for a pending addition, and a discarded object attached to an existing tuple stands for a pending deletion of that tuple. When objects are modified, the modifications are written to the tuples objects stand for, and so on. At which intervals the object-space is equalized with the relations is largely a matter of optimization. Since relational databases are designed to handle large amounts of data efficiently rather than small requests quickly, having a large object-space improves usability in interactive applications.

It should be noticed, however, that the model only with tables that have an unique key. What comes to other tables and in particular views and queries, a more rigid and abstract model is needed at the object-oriented side. A result of a query is hence represented as a two-dimensional array and not as a dictionary of objects. Even if arrays are objects too, they come much closer to the anonymous world of "1"s and "2"s with no active-data operations and

meaningful identity. As a result, the following mapping between relational concepts and classes could be imagined:

Relational model	**Object-Oriented programming**
Tuple in a keyed table	Instance of an application-specific class, "active data" in use
Tuple in some other table	A dictionary of key-value pairs
A row in a relation	An array of numbers and strings in an array of rows

3. PROGRAMMING IN SMALLTALK/V PM

Programming in Smalltalk differs in two ways from programming in languages as Ada or C. For one, the language and the environment is extremely dynamic. The system is usually based on some kind of a semi-interpreter, which means that everything can be run and tried out at once, like in BASIC or LISP. Since the methods in the classes are strictly isolated, a method typically can be tested as soon as it has been programmed, provided that the services it uses from other classes have been implemented. This enables bottom-up-style programming, where code is incrementationally tested and there is no large gap between the programming phase and the testing phase. Yet, thanks to the powerful isolation mechanisms, nothing prevents the programmer from later improving the underlying implementation or porting it to other environments. As an example of this, the de-facto Smalltalk standard has been implemented in Sun News, Macintosh and MS-Windows and the code written in one of them runs *unmodified* also in the others [Udel90].

The other characteristic is the vast amount of ready-to-use classes which makes programming very efficient for a user who knows these classes and utilizes them. These include all the common arithmetical and time-related measures, a rich selection of collection and container classes, complete package of tools for making graphical-user-interfaces and so on. There is no point in trying to name them all: in stead we pick a typical programming example and show how it can be solved in Smalltalk.

3.1. A programming example

In the course of everyday programming I found myself in need of a tool to compare two about similar text files and tell their differences. The GNU toolkit contains one called "diff", written for UNIX, so I downloaded it and tried to port it into OS/2. After some 7-8 hours of wasted work I gave up. The application is written in C, it contains some 9300 source lines, and changing the wild pointer usage customary in UNIX so that OS/2's memory protection wouldn't have trapped simply turned out to be too elaborate.

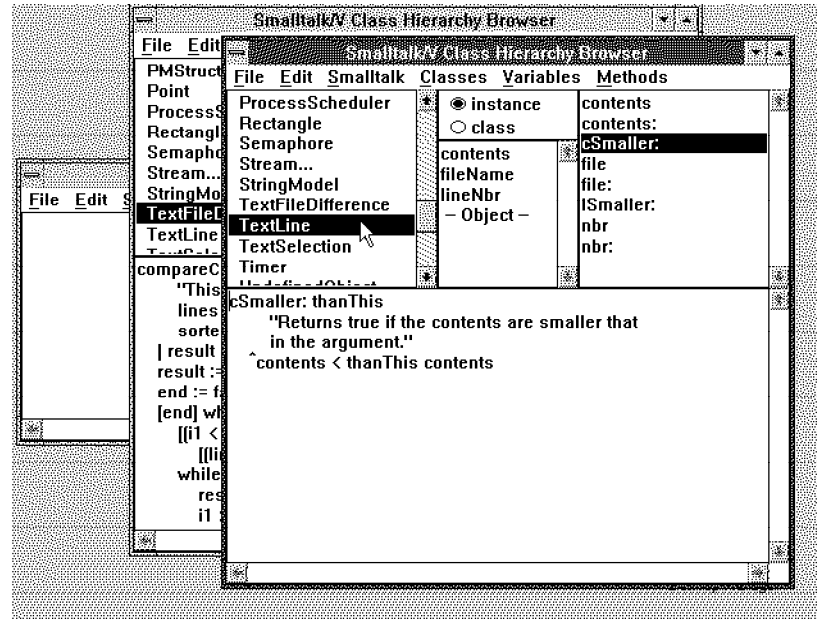
Then I figured out a minimal solution of my own. It goes like this: first sort both text files according to their contents into two lists. Then start comparing the two lists: as long as the elements equal, both lists can be proceeded. If they differ, the smaller element is added into a third list and the list it came from is proceeded. This is repeated until one of the lists is fully handled, after which all the elements left in the other list can be added to the result list.

Finally, the result list is sorted, this time according to line-numbers in the original files, and printed out. Each line is preceded by the file name it comes from and the line number in parenthesis.

The algorithm is not complete, and I doubt whether there is one for the problem. In particular, it does not report any difference if the two files contain the same lines but in a different order. It could be modified quite easily to do that too by checking also the line-numbers in the comparison, but this time that wasn unnecessary, since what was needed was simply a list of lines which couldn't be found from the other file. In the basic form the algorithm runs in $O(n*\log n)$ and making the modification would raise it to $O(n^2 \log n)$, because each time a match with different line-numbers would have found, the rest of the line-numbers would have to be incremented accordingly.

The complete sources of the implementation can be found from appendix E.

Smalltalk/V PM's user interface looks typically like this:

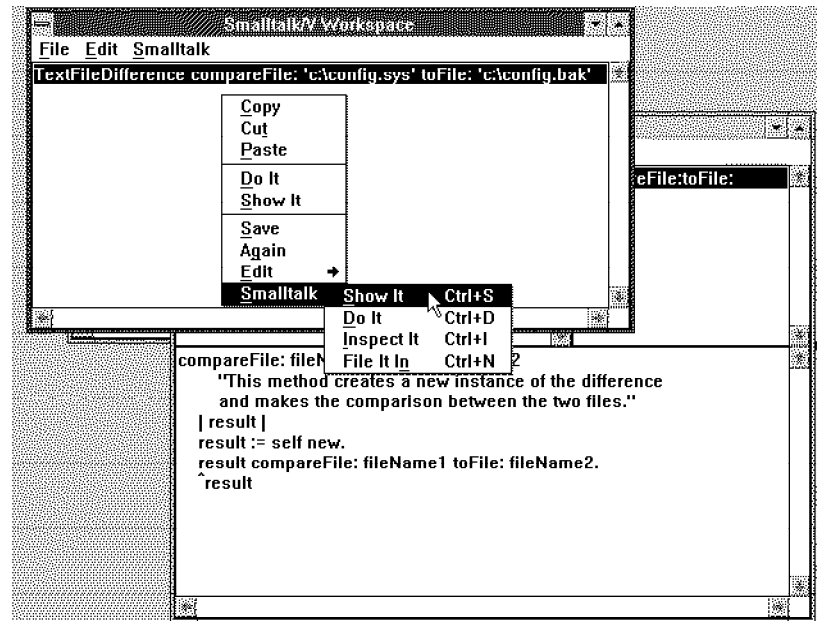


Two class-browsers and the System Transcript -window is a typical sight in Smalltalk programming.

The two top-windows are called class-hierarchy-browsers, which are used for viewing and editing the program code. The browser's window is divided into five subareas called panes. On the left there is the class-pane, which contains a hierarchical list of all classes. The methods of the selected class are in a list-pane on the right, and the code of the selected method is in a text-pane on the bottom. The code can be modified and as soon as it is saved by clicking the mouse on it, it is incrementationally compiled and can be run. If errors are detected, the cursor jumps to the problematic expression. Actions are invoked by selecting them from menus, which can be activated from the menu-bar (macintosh-style pulldown-menu) or right from the pane (motif-style pop-up menu) by pressing the right-button of the mouse.

The third window is a text-window called the System Transcript. A text-window contains just one pane in which text can be written to and executed as Smalltalk statements at will. For instance, to evaluate 3+3 the text "3+3" is selected by the mouse, "show it" is selected from the "Smalltalk"-menu, and "6" appears to the window just after the selected area. Just the same, the system could be exited by selecting 'Smalltalk exit', while that of course can be done also by selecting it from the system menu. And for making the comparison with files "config.sys" and "config.bak" to see how the configuration file has changed, we select and execute the

statement "TextFileDifference compareFile: 'config.sys' toFile: 'config.bak'".



Smalltalk-statements are executed by selecting them in a text-window and choosing "Smalltalk - do it" or "Smalltalk - show it" from the menu, which this time is a two-level pop-up one.

3.1.1. TextFileDifference

In Smalltalk everything starts from finding out the right classes. What we now are seeking for is the difference of two text files, so we add a new class called "TextFileDifference" to the system. The classes are grouped hierarchical so that the more specific cases are subclasses of the more general cases, but since there is no general 'Difference'-class we make the new class a subclass of the topmost or root-class: 'Object'.

The needed method is called "compareFile:toFile:" which can be seen in the picture above. In it, a new instance of the class is created and the differences of the two files are stored into it by calling an method with the same name and arguments. (There is no possibility for mix-up: class's methods and its instances' methods are separated and may have same names without confusion. In general, objects with different classes may have different implementations for methods with the same name.)

The "compareFile:toFile:"-method reads in the contents of the two files:

```
compareFile: fileName1 toFile: fileName2
    "This method compares the contents of the two text-
    files and puts the differing lines into the diff-ordered
    collection."
  | lines1 lines2 |
  lines1 := self readContents: fileName1.
  lines2 := self readContents: fileName2.
  diff := self compareContents: lines1 with: lines2 .
```

the "diff"-variable is defined as an instance variable when the class is created, and its obvious purpose is to carry the result of the comparison for later usage. Each instance of the class has one in it and it is accessible from all methods. It is also possible to define class variables, which are also accessible from each instance but there is only one copy for all instances. This way instances of the same class can share common data. "self" refers to the object itself.

The actual comparison algorithm is in the instance-method "compareContents:with:"

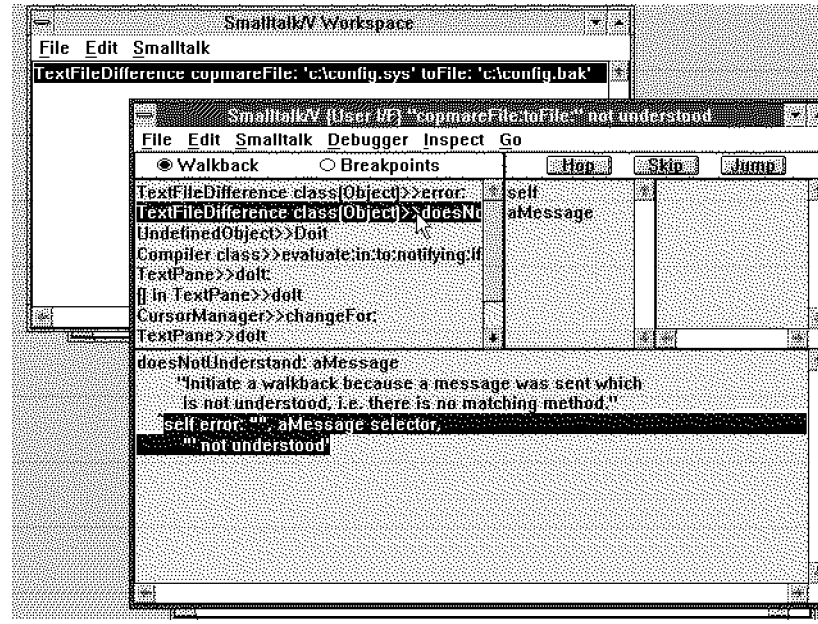
```
compareContents: lines1 with: lines2
    "This method compares two sorted collections of
    lines and adds the differing components into another
    sorted collection, this time by the line number."
  | result end i1 i2 |
  result := SortedCollection sortBlock: [:a :b | a lSmaller: b].
  end := false. i1 := 1. i2 := 1.
  [end] whileFalse: [
    [(i1 < lines1 size) and:
      [(lines1 at: i1) cSmaller: (lines2 at: i2)]]
    whileTrue: [
      result add: (lines1 at: i1).
      i1 := i1 + 1].
    [(i2 < lines2 size) and:
      [(lines2 at: i2) cSmaller: (lines1 at: i1)]]
    whileTrue: [
      result add: (lines2 at: i2).
      i2 := i2 + 1].
    (i1 = lines1 size or: [i2 = lines2 size])
      ifTrue: [end := true].
    ((lines1 at: i1) contents = (lines2 at: i2) contents)
       ifTrue: [
          i1 := i1 + 1. i2 := i2 + 1]].
  [i1 <= lines1 size] whileTrue: [
    result add: (lines1 at: i1).
    i1 := i1 + 1].
  [i2 <= lines2 size] whileTrue: [
    result add: (lines2 at: i2).
    i2 := i2 + 1].
```

```
    ^result
```

Here there is already something that is not self-evident from the context, though the meaning of the code could be figured out by remembering the description of the algorithm above. So, a little bit closer inspection is needed.

Smalltalk contains only a few syntactical constructs. Most lines are read like this: receiver, method-name and arguments intermingled, and period. The receiver is the object to which the message is send, the method-name (selector in Smalltalk) specifies the procedure that is activated and the actual parameters replace the formal parameters in the execution of the method. Since every method returns something - the receiver if nothing else - that can be assigned to a variable. Or like in C, the returned value can be ignored. ^ is the same as "return" in other languages. The variables between bars - | result end i1 i2 | - are the local or temporary variables of the method. And harshly taken, '[' stands for "begin" and ']' "end". What is in quotations, is a comment.

There are a couple of details worth noticing. For one, there is no mention of the class of either arguments or temporary variables. When an object is declared, it contains value 'undefined object' or 'nil', but assigning something else to it changes that. When objects receive messages, they consult their classes in what to. If neither the class nor any of its superclasses (ancestors) recognizes the selector, the method "doessNotUnderstand:" is activated: since that is defined in "Object" it is common to all. Therefore, as long as all objects understand the messages send to them, everything goes fine. If not, a debugger window with a message trace is automatically opened.



When an invalid message is received - like the blundered "copmareFile:toFile:" here - a debug-window is opened. The code can be inspected and even modified right in it.

Another quite an original feature is that the language does not have almost any reserved words. The context of a piece of code is implicitly indicated by its position in the class browser: what class it belongs to and whether it is a class method or an instance method. The remaining other things are usually expressed by some special characters. Even the program's logic is implemented according the message-passing model: a conditional statement looks typically like this:

```
("condition") ifTrue: ["then do this"] ifFalse: ["then do this."]
```

and its interpretation goes like this: a boolean object - either true or false - receives message "ifTrue:ifFalse:" and executes one of two portions of code it has been given as arguments. A piece of executable code is called a **block** and it can be given control simply by sending it the message "value". Unless - of course - the sender wants the block to be executed concurrently with its own code, in which case it sends message "fork" in stead and a new process is created which starts to execute the block. Elementary, isn't it?

3.1.2. TextLine

Where is the sorting then, which should form the main part of the work? Well, it appears that this is already in order. When the files

are read, the data in them is stored to lists as instances of the "TextLine"-class:

```
readContents: fileName
        "This method reads the contents of a text file
        into an ordered collection with the line-number and
        contents of each line as components."
    | iFile result lineNumber |
    lineNumber := 1.
    result := SortedCollection sortBlock: [:a :b |
        a cSmaller: b].
    iFile := File pathNameReadOnly: fileName.
    [iFile atEnd] whileFalse: [
        result add: (TextLine file: fileName nbr: lineNumber
            contents: iFile nextLine).
        lineNumber := lineNumber + 1].
    iFile close.
    ^result
```

This class is simply something like a record of three elements: the file name, the line-number and the contents of the line. A new instance is created by sending to the class message "file:nbr:contents:". These instances are then added to the "result" variable, which contains an instance of the class "SortedCollection". Such an instance has the property that data added to it is automatically inserted in the right place. The order is decided according to a given block of code which compares two elements from the collection and tells whether the first argument is smaller than the second one. In the example above, the sort block is

```
[:a :b | a cSmaller: b]
```

which can be thought as kind of a ad-hoc function without name. It has two arguments and it returns a boolean value. Since in this point the contents of the text-lines are compared, the cSmaller-method is obviously implemented in the TextLine-class like this:

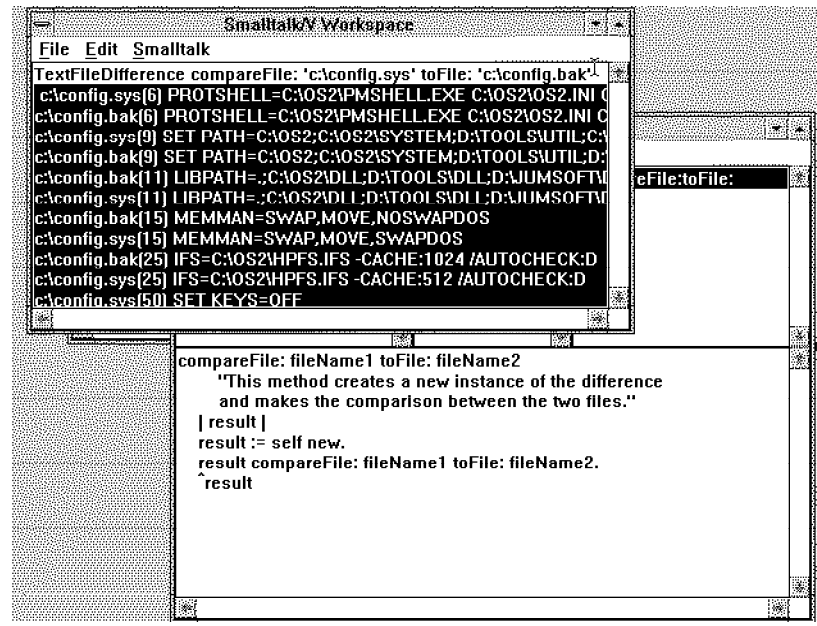
```
cSmaller: thanThis
        "Returns true if the contents are smaller that
        in the argument."
    ^contents < thanThis contents
```

In the "compareContents:with:" -method, on the other hand, the sort-block is "[:a :b | a lSmaller b]", and it's not difficult to guess that now line-numbers are compared.

What is now needed to complete the utility is just a method for printing the contents of the comparison. For this purpose the "printOn:" -method is usually used. Each class has the total freedom of implementing this method as it sees best, and that's what is done also here:

```
printOn: aStream
        "Show the contents of the difference."
    diff do: [:each |
        aStream nextPutAll: each file.
        aStream nextPut: $(.
        aStream nextPutAll: each nbr printString.
        aStream nextPutAll: ') '.
        aStream nextPutAll: each contents.
        aStream cr].
```

The "do:"-method is a set-iteration which goes through any collection and evaluates the given block with each element in the collection as argument. Since "diff" is not just any collection but a sorted collection in particular it is handled in order. And thus, the result of evaluating the "compareFile:toFile:"-method looks like this:



When "Smalltalk - show it" is selected, the result of the evaluated block is printed in the current text window.

So, the utility is about finished. The complete code can be found from the appendixes, and it appears to contain 146 lines, of which

more than a half are either blank or commented. Although the comparison to the much more complete GNU "diff" is not quite justified, the difference demonstrates fairly well the power of Smalltalk.

The code was written in one Tuesday-afternoon when I encountered the need for finding out which twenty lines in two over 1000-lines long source files were different. Writing it took less than two hours - testing included - and running the comparison took less than a minute.

4. AN OBJECT-ORIENTED DATABASE INTERFACE

In the second chapter we have already named the classes that are needed in the database interface and outlined their responsibilities. The classes are tuple on the other hand, and the various tuple collections - relation, table and a keyed table - on the other. The main responsibility of a tuple instance is to enforce the semantic constraints and dependencies of the entity it represents, while the other dependencies are taken care of by those classes which model tuple collections.

4.1. The Database-class hierarchy

This is already enough to define the class-hierarchy. All of the new classes go under the superclass "Collection". The collection-hierarchy looks approximately like this after the new classes have been added.

```
Collection
  Bag
  IndexedCollection
    FixedSizeCollection
      Array
      ByteArray
      Interval
      String
    OrderedCollection
      Process
      SortedCollection
  Relation
    Table
      KeyedTable
        UserDefinedTables...
  Set
    Dictionary
      Tuple
        UserDefinedTuples...
```

Smalltalk Collection-hierarchy with the database classes in it. The hierarchy is presented by indentation, and the new classes are in italic.

A dictionary is a general collection of key-value pairs, and since that is what a tuple basically is, we make the Tuple-class as a subclass of Dictionary. The actual tuple-types in the keyed tables in a database are then made as subclasses of Tuple. These classes are where the tuple-specific code is put in.

Since a keyed-table is a table, which in turn is a relation, we add a whole new branch to the tree right under collection for these classes. As a matter of fact, Relation could be seen also as an array and KeyedTable as a dictionary and consequently made subclasses of Array and Dictionary. This would break the Relation-Table-KeyedTable hierarchy, and since multiple inheritance is not in use we prefer it. As will be seen later, nothing is really not lost due to this restriction, since the implementation of Array or Dictionary couldn't be re-used by the database classes anyhow.

4.2. Notation and terminology

Much of the rest of this document will be descriptions of Smalltalk-classes. To make it easier to catch the essentials, the classes are documented systematically much like in the Blue Book (actually the new version is violet) [Gold88, p. 41 - 44]. The class and instance protocols are described by mentioning each method and its purpose. The methods are also grouped into categories according to their usage. Some predefined categories are:

Constructors Class-methods which are used for creating a new instance of the class. "new" and "new:" are the most often used constructors, but also others are possible.

Destructors Instance-methods which are used for making the instance discard itself and relieve all its dependants. Used mainly with objects that have connections to outer world, since garbage collection handles all internal memory reclamation.

Impulses A very general category of methods which typically causes a change in the object's internal status and return no other data but the object itself.

Attributes Another general category which is mainly meant for making queries about the object's internal status. Attributes shouldn't alter the receiver's state so successive attribute calls should return unchanged values.

Private Methods which are not meant to be used from outside but only by other methods of this class or subclasses. These are not usually mentioned at all in the protocol definitions.

Iterators Collection classes may contain iterators. An iterator is a method which goes through the collection and evaluates the given one-argument block with each element

in the collection as parameter. The most often used iterator is "do:" (see the previous chapter).

The idea behind the division to impulses and attributes is simple: a programmer should be able to query any aspects of an object at any time, without having to worry about the side-effects. Therefore, attributes musn't change the state of the receiver. Then, those methods which do change the status shouldn't return any other information so that the programmer wouldn't be temptated to use them in stead of attributes. The only information they return should be a code which tells whether the attempted operation succeeded or not. In Smalltalk even this is not necessary since the error management is handled so that no status or return codes are needed: the errors are signalled through the "error:"-method as soon as they are detected.

The examples in this chapter use the database described in the sixth chapter. Consult it and the appendix A for information on its contents.

4.3. Tuple

In its simplest, a tuple is just a dictionary with symbols as keys. There is a constructor for creating a new, empty tuple and another for constructing it according to data read from database.

Tuple class protocol

Constructors

dbData: aTuple	Returns a new tuple with the values taken from the primitive tuple.
new	Returns a new empty tuple.

Attributes

keyFields	Returns a list of the keyfields of the Table which is the primary storage of the Tuple. Usually implemented by subclasses.
virtualFields	Returns a list of the fields which have no physical representation in the database. Usually implemented in subclasses.

The two attributes above hint that this class is not primarily meant to be used for creating new instances but rather as a template for making subclasses. Such a class is often called an **abstract superclass**. Tuple is however used also as such inside the database as will be shown later.

As mentioned above, an object in Smalltalk has its internals protected so that they cannot be referenced directly. The cost of this is that the access methods have to be written separately for each component. With tuples we would not like to do this, however, since a database may contain dozens of tables with thousands of attributes, and among them the great majority doesn't require any more checking than what comes with the type definition. Writing one access method for reading an attribute value from a tuple and another for setting it for all thousands of attributes would be an overkill. That's why Tuple is a dictionary, since its components can be acquired by the standard methods:

Tuple instance protocol

at: aKey	Return the object stored with aKey as the key-value
at: aKey put: anObject	Store the object associated with the given key-value.

But what about protection then? What is needed is a way for using special access methods when they are needed and the default "at:" and "at:put:" -methods otherwise. Furthermore, this must be done so that the difference is not visible to the calling layer, so that new constraints can be added later without modifiying software elsewhere.

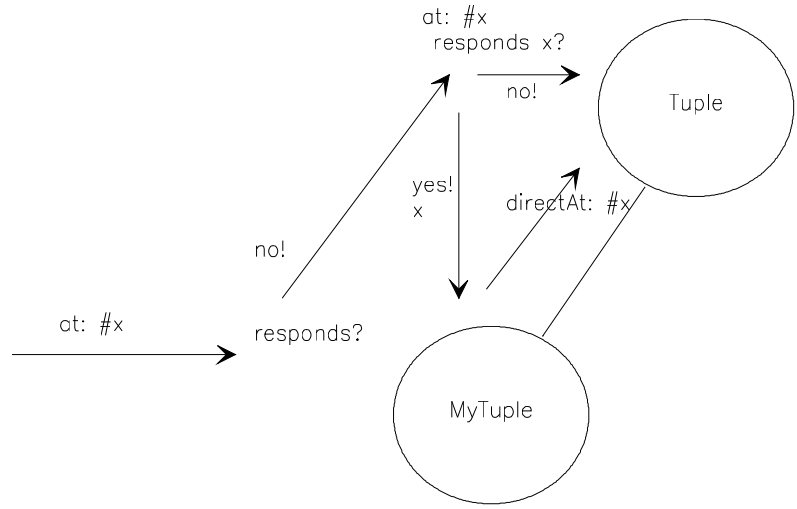
It appears that in Smalltalk we can do exactly that by re-implementing dictionary's standard access-methods in Tuple like this:

```
at: aKey put: aValue
        "If the receiver contains a binary method for setting
        the value that's used."
    | method |
    method := (aKey, ':') asSymbol.
    (self respondsTo: method)
        ifTrue: [^self perform: method with: aValue]
        ifFalse: [^self directAt: aKey put: aValue]
```

```
at: aKey ifAbsent: aBlock
    "If the receiver contains a named method for accessing
    the component, that's used."
  (self respondsTo: aKey asSymbol)
    ifTrue: [^self perform: aKey]
    ifFalse: [^self directAt: aKey ifAbsent: aBlock]
```

The idea is to define *default names* for the special access methods, test whether such methods exist for a tuple in the standard access-methods and redirect the messages to them if they do exist.

For instance, lets take a tuple X with keys #a #b #c. (The hash in the beginning of a string marks a symbol in Smalltalk.) Setting the field #a needs an extra check, while #b and #c can be set directly. So, let's define method "a:" for tuple subclass X. Now, when an instance of X receives message "at: #a put: anObj", it notices by help of the "respondsTo:"-method that there is a special access method defined for #a, called "a:", and uses that. When the key is #b or #c it uses the "directAt:put:"-method. When accessing an attribute, a similar mechanism is used; this time the name to be used is just the attributes name



When a field is read from a tuple, the special access method is used if one is defined. Otherwise, the default "directAt:"-method is used.

There is also a group of other methods for tuples:

Tuple instance protocol

Attributes

choicesAt: aKey	This method tests whether the tuple is able to give choices of what can be entered to the field denoted by the key. If so, it returns them as a string array, otherwise it returns nil.
< aKey	Compares two tuples according to their keys.
dbData	Answer the essential contents of the tuple as a primitive Tuple. Only the non-virtual elements are included.
dbKey	Returns an instance of own kind with no other fields but the keyfields.
species	A tuple is really a Dictionary by heart.

Impulses

printOn: aStream	prints the contents of a Tuple in a somewhat readable form.

The selection of methods is quite usual: there are the comparison and the "printOn:"-method which are some of the most often re-implemented methods. The "dbData"-method returns the contents of the tuple in a form suitable to be written to the database. One interesting detail is the "species"-method. All objects know the "class"-method, which returns the class of them. The "species"-method works about the same way, expect that in some cases it might return a superclass or an entirely foreign class in stead of its actual class. By claiming to be a dictionary Tuple enables some dictionary-oriented tools which can then be applied on it as well.

The basic tuple class returns an empty tuple from the new method. The fields can then be added to it later on, but a more natural method is to make a subclass which redefines the "new"-method and sets the initial fields and values. For example, a tuple which appears in the database like this:

```
create table X (A char(10), B int, C float);
```

might be created in the Smalltalk-program like this:

```
class X is subclass of Tuple

new
        "This is class X's "new" method, which sets the fields
        and their initial values."
    | result |
```

```
 result := super new.
result
   at: #a put: "";
   at: #b put: 0;
   at: #c put: 0.0.
^result
```

A number of messages can be sent to the same object at once by writing the semi-colon between them. Another detail worth noticing is the custom of writing the key-names in lowercase letters. This is due to the fact that the key-names are used for generating the potential access-methods names and Smalltalk is a case sensitive language.

4.4. Relation

Relation is the most abstract database of the three classes which store tuples. A relation in the database can be either a table or a view or just the result of executing a SQL-query. Thus, the Relation-class has two deeds: to serve as the ancestor of the more specific classes and provide means for executing and evaluating SQL-queries from programs.

A relation is created by opening it in a database. The given argument must be an opened instance of the Database-class.

Relation class protocol

Constructors	
openIn: aDatabase	Opens a new relation in the given database.
Attributes	
rowClass	Returns the component class. For a relation this is Array by default, but the subclasses may defined it more specific.

Since a query may contain expressions as result columns there is no point in trying to form a dictionary with the column names as keys. The contents of a query are represented by a two-dimensional array, from which the atomic values can be accessed by giving the row and column -number.

For defining the queries, SQL is in a way translated to Smalltalk. For each syntactic part of a query there is an own method by which its arguments are defined.

Relation instance protocol

Query methods

select: columns from: tables	This is the basic select-from method. The arguments are strings which contain the columns or expressions and table names. Other query-methods add their further terms to this.
where: terms	Defines the logical conditions by which the tuples are selected.
orderBy: columns	Defines the order in which the rows are sorted to.
groupBy: columns	Defines the grouping argument in the query.
having: condition	Defines an additional logical condition for the groups formed by the groupBy-term.
bindTo: this	Sets the values of the optional bind-variables in the query. If numbered bind variables are used, the parameter should be an array, with named ones, a dictionary, and if there is only one bind-variable numbered ":1" then it can be simply the value of it.

Access methods

row: rowNumber	Returns the given row from the query as an array of atomic elements. The query must have been defined by the query methods before this method is called.
row: rowNbr col: colNbr	Returns the nth column of the mth row, either a string, figure or a date or time -value.

Iterators

do: aBlock	Goes through the relation row-by-row and evaluates the given block with each row as argument.

Attributes

size	Tells the number of rows in the query result.

Destructors

close	Closes and releases the query.

After the relation has been created, the query has to be defined by calling at least the "select:from:"-method and other selection methods if they are needed. If bind-variables are used, the values must be set by the "bindTo:"-method and then the contents of the relation can be asked. The SQL-statement is compiled and executed automatically; if the bind-variables are set again the statement is re-executed and if the query is changed by some of the query-methods it is also compiled again.

For example, to find out the average price of hammers supplied by suppliers who supply also nails under the price of 3.95 the following Smalltalk-code is needed:

```
| query |
query := Relation openIn: ItemsDatabase.
query select: 'AVG(PRICE)' from: 'ITEMS';
   where: 'NAME = :1 AND SUPPLIER IN (',
      'SELECT SUPPLIER IN ITEMS WHERE ',
      'NAME = :2 AND PRICE <= :3)'.
query bindTo: #('Hammer', 'Nail', 3.95).
^query row:1 col:1
```

The selection criteria could be written directly inside the query but using bind-variables makes the statement somehow more general and reusable. The "ItemsDatabase"-variable above is supposed to be a global, already opened database.

4.5. Table and KeyedTable

Through a relation the database can only be read from, not written to. For this, there are the classes Table and KeyedTable. The former is more like an abstract superclass with few methods. The reason for this is the earlier mentioned fact that a table with no unique key is hard to bind with the object-space where each object has an identity of its own. Only the transaction handling methods are left at this level:

Table instance protocol

Transaction management

commit	Commits the changes to the database so that after a termination - normal or abnormal - they were the last ones which were written.
rollback	Discards the changes written to the database after the last commit.

The KeyedTable-class provides the actual user-interface for handling physical tables in the database - remember that writing through a view is usually impossible. In the constructor the name of the table is given as a parameter:

KeyedTable class protocol

Constructors

openIn: aDatabase ofName: table	Opens a physical table in the database for reading and writing.

Attributes

rowClass	Returns the class of the elements. At this level it is the general Tuple, but in subclasses it is typically a subclass of Tuple.

Once a keyed table has been opened it can be manipulated by a rich set of operations, which work very much like in a dictionary.

KeyedTable instance protocol

Access methods

at: aKey	Returns the tuple which contains the given key value or nil if none was found. The parameter is a tuple of just the key-fields, possibly returned from calling the dbKey-method of a tuple.
nextTo: aKey	Returns the tuple with the next-larger key-value compared to the given key.
prevTo: aKey	Returns the tuple with the next-smaller key-value compared to the given key.

Writing methods

at: aKey put: aTuple	Updates the tuple in the database which contains the given key-value.

add: aTuple	Adds a new tuple in the database.
delete: aKey	Deletes the tuple from the database which contains the given key-value.

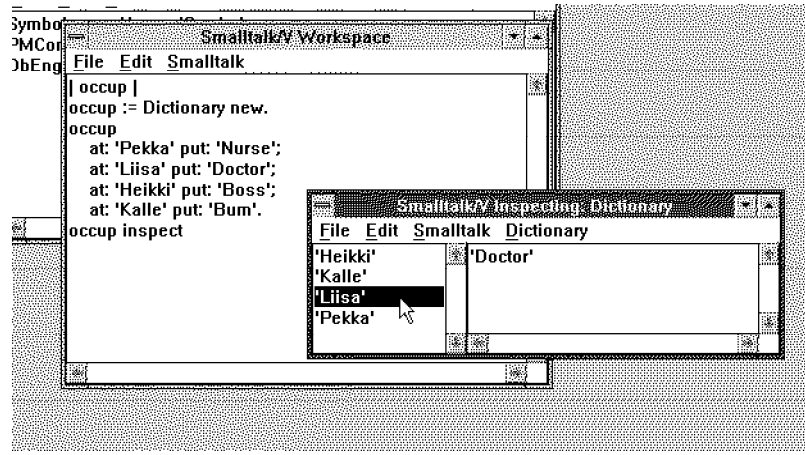
Iterators

keysDo: aBlock	Evaluates the given block with the key of each tuple in the table as argument.

Attributes

includesKey: aKey	Tells whether the table contains a tuple with the given key value.
species	Claims that a KeyedTable is really a dictionary.

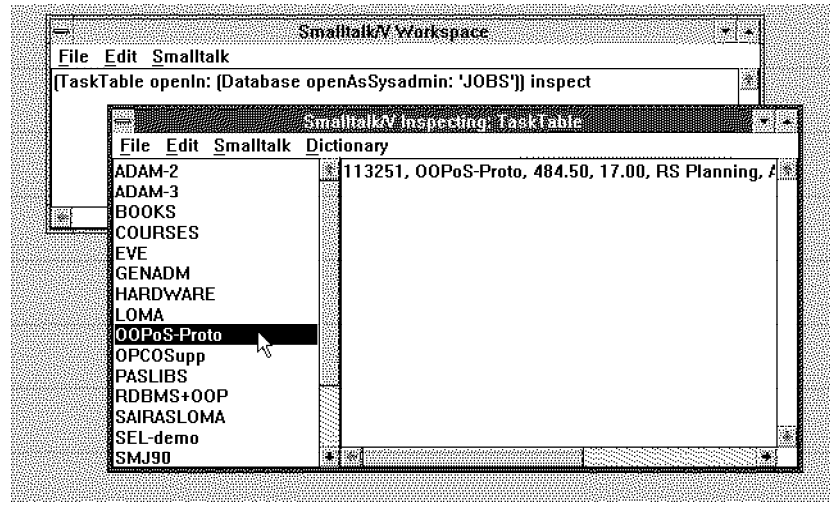
Since a KeyedTable behaves like a dictionary by implementing all of the essential operations it really can be used in many contexts where a dictionary is expected, although it is not even a subclass of the Dictionary-class. A small example of this: in Smalltalk/V PM there is a tool called Dictionary Inspector, by which any dictionary can be viewed and modified.



The "occup"-dictionary is created and assigned values to in the WorkSpace window. The "inspect"-statement causes the inspector-window to open.

In the dictionary inspector there are two panes: a list pane on the left, where the key values are visible, and a text pane on the right. By selecting a key on the left makes the associated value to become visible on the right. That value can be modified and saved to the dictionary, new values can be added to the dictionary and existing ones can be discarded.

Now, let's make an instance of a keyed-table and try to open a dictionary inspector on it:



Also keyed tables can be viewed by a dictionary inspector.

The dictionary inspector works as well with keyed tables, since they implement most of the needed dictionary methods. The table in the example contains task-definitions, with the name of the task as a key. The tuple is printed on the right (remember the tuple's printOn: method) as a text-string. Even if KeyedTable and Dictionary have completely different implementations, the similarity of protocols is enough for using all of the dictionary-tools also with keyed tables.

4.6. Table-subclasses

KeyedTable is however yet another abstract superclass. Unlike Relation, which can be used immediately for creating instances, a subclass needs to be defined for it to fix a number of things needed in the implementation.

In particular, this is the place where the needed SQL-statements are stored. Each KeyedTable-subclass must define them as class methods and they are used when the KeyedTable access-methods are called.

KeyedTable's subclasses class protocol

SQL-statements

insertStatement	Returns the statement to be used when new items are inserted to the database, like "INSERT (:a, :b, :c) TO TABLE_X".
updateStatement	The statement used in updating the tuples in the table.
deleteStatement	The delete-from-table statement.
selectThisStatement	The select statement which picks up a specific key-value: "WHERE keyField = :1"
selectNextStatement	For getting the next tuple: "WHERE keyField > :1 ORDER BY keyField
selectPrevStatement	For getting the previous tuple: "WHERE keyField < :1 ORDER BY keyField DESC"

Attributes

rowClass	Returns the application-specific tuple subclass which is the element class of this table class.

Complete examples of these can be found from the application sources. There are some rules about the bind-variable naming to get the binding work automatically with the dbKey-method of tuples, but that is not too interesting.

4.6.1. Collecting data from tables

A more interesting part is the one where a row has been fetched from the database and a new tuple is constructed according the data in it. For this, there is a method getValue which is implemented in Relation like this:

```
getValue
    "Private - the rows in a relation or a table are returned as
    instances of the row-class which is specified for Relation,
    Table and their subclasses."
  | rowClass |
  rowClass := self class rowClass.
  rowClass = Array ifTrue: [
    ^rowClass dbData: cursor value]
  ifFalse: [
    ^rowClass dbData: cursor namedValue]
```

A new instance is created of the rowClass by sending it the "dbData:"-method, with the raw data from the table as an argument. Remember still the normalization process where data

was split in a number of tables all around the database? This is now the place where the data is gathered together again.

There are two access-points in the code which can be modified for this. One is a tuple's "dbData:"-method. Every time a tuple receives that message, it knows that the rest of its attributes should be fetched from other tables. This is however a bad solution, since it mixes tuples and tables. Ideally, a tuple should exist on its own right without having to know anything about tables.

The other access point is the "getValue"-method itself. Semantically this means that each time the master-table is fetched from, some other attributes are also fetched from other tables by using whatever appropriate access methods. In the application example there is the TASKS-table which contains information of various working tasks. The ACCOUNT-table contains markings of working periods, each attached to some task. Since the task-tuple contains the amount of worked hours in the current month and year as virtual attributes, it has to get these from ACCOUNTS each time a new task-tuple is fetched from the table, So, this is how class TaskTable implements the "getValue"-method:

```
getValue
    "Private - overrides the default method. Data is fetched
    also from the account-table."
  | result taskId firstDay lastDay temp |
  result := super getValue.
  taskId := result at: #ident.
  hours select: 'SUM(WMIN)/60' from: 'ACCOUNT';
    where: 'TASK = :1'.
  hours bindTo: taskId.
  temp := hours row: 1 col: 1.
  temp isNil
    ifTrue: [result at: #totalhours put: (Fixed new: 0.0)]
    ifFalse: [result at: #totalhours put: (Fixed dbData: temp)].
  lastDay := Date today.
  firstDay := lastDay firstDayOfMonth.
  hours select: 'SUM(WMIN)/60' from: 'ACCOUNT';
    where: 'TASK = :1 AND WDATE BETWEEN :2 AND :3'.
  hours bindTo: (Array with: taskId with: firstDay with: lastDay).
  temp := hours row: 1 col: 1.
  temp isNil
    ifTrue: [result at: #thisMonthsHours put: (Fixed new: 0.0)]
    ifFalse: [result at: #thisMonthsHours put: (Fixed
      dbData: temp)].
  ^result
```

Since evaluating this method is the only way to get data out of the task-table, there never is any risk that the accounts-table would be forgotten to be read and the hours-figures would be obsolete. It also totally hides the complexity due to the fact that data for a task tuple is needed from two tables by using three queries.

In a nutshell, putting the data gathering inside the "getValue"-method solves about half of the problems caused by inclusion dependencies in the database. One of the problems that still remain is that reading both tables even if all of the data wouldn't be needed means extra overhead and slows down the system. The problems in writing to the database, on the other hand, are quite easily solvable in the user-interface, as will be seen.

4.7. Database

The purpose of the Database-class is to encapsulate the access control and provide the low-level connection for other classes. When a database is connected, an instance is created for this.

Database class protocol

Constructors

open: dbName userName: userName passWord: passWord	Returns a new database connection to the given database with those user rights which have been granted to the given user, identified by password.
openAsSysadmin: dbName	Returns a new database connection with SYSADMIN's rights.

Once a database connection has been created, it is given as a parameter when new relations, tables and keyed tables are created. Just the database connection can be used for executing such SQL-statements which require no data-exchange, like the administrative operations granting access rights, creating new tables, indices and so on:

Database instance protocol

Impulses

doSQL: aStatement	Compiles and executes the given SQL-statement to the connected database.

Destructors

close	Closes the database connection.

In an application one database instance is typically created in the beginning and then used for creating all the other needed objects.

4.8. The low-level connection

The database implementation is based on a database server which provides an application programming interface (API) for the C-language. This API cannot be called directly from Smalltalk due to some technical reasons, and it is not too nice to use anyway because of its gloriously non-abstract design. That's why a couple of more classes are needed to connect our database classes to their counterparts in the actual database.

4.8.1. External calls from Smalltalk/V PM

Foreign modules can be called from Smalltalk via an API-mechanism, which converts a message into an OS/2-like dynamic-link-library (DLL) call. These libraries are characteristic to OS/2 and they implement late binding by allowing a function to be called by its name rather than its address. The DLL-module contains a look-up-table from which the actual calling address is fetched according to the given name.

Once a DLL-entry has been activated it expects the stack to be ordered as if it were called from a statically linked module. That's why the caller has to set up the stack according to the conventions followed in the called funtion. In OS/2 there are two such conventions, called API or Pascal and C. In the former, the parameters are put on the stack in reverse order and the caller removes them from the stack after the call. In the latter the parameters are put on the stack in the same order they are declared in the call and the called function removes them from there. Smalltalk supports both these conventions. When an external call is to be made, a special method is written for it like this:

If in the DLL entry follows the API calling convention:

```
int far pascal _loadds myPlus (int a, int b); /* This is MS-C */
```

Then it is called like this:

```
myPlus a: and: b
    <api: MYPLUS int int int>
    ^self invalidArgument
```

while if the C calling convention is followed

```
int far _loadds myPlus (int a, int b);
```

Then the call should go:

```
myPlus a: and: b
    <c: myPlus int int int>
    ^self invalidArgument
```

These external-call methods should belong into a subclass of the DynamicLinkLibrary-class. The DLL is activated when an instance of the subclass is created:

```
DLLHandle := MyDLL open: 'MYDLL'
```

When this call is made, the operating system searches for a file "MYDLL.DLL" which should contain the functions and loads it into memory. After this, the functions can be called like Smalltalk-methods:

```
a := DLLHandle myPlus: 1 and: 2.
```

There are however some limitations in what can be passed as parameters from Smalltalk to DLLs. In general, all but the most primitive objects are represented as pointers in Smalltalk. The pointer can be passed to the DLL, which then can manipulate the object provided that it knows Smalltalk's internal object representation. This is not a very good solution, however, since the object representation may change in later Smalltalk versions and the DLL would be limited only for Smalltalk-usage. Such objects which contain a contiguous data-area like byte-arrays and strings can also pass a pointer to the beginning of this area. This is clearly a better method, and in general most data exchange should happen by help of such buffers, because they are relatively safe and require no knowledge of Smalltalk's internal implementation.

Small integers and boolean values are also represented as segment-offset pairs for compatibility, but so that the offset value of the

pointer really contains the value of the variable. Such a pseudo-pointer is recognized by the segment-value: a few small segment-numbers are reserved for this purpose. While this is a good solution for performance, it makes impossible to pass the address of such simple objects outside Smalltalk. Therefore, integers and booleans cannot be used as reference-parameters in DLL calls. There are ways to work around this limitation, though.

4.8.2. The lowest level: dbEngine.dll

The dbEngine-module has three objectives. First, it uses such a calling model that no pointers to integers or booleans are used but all result values of these types are returned as function results. This makes it comfortable to use from Smalltalk/V PM. The second purpose is to hide the characteristics of the specific database solution so that the using Smalltalk-code can be later ported on top of some other database server. And finally, the application interface of SQLBase is not the neatest piece of software in the world with its strange naming conventions and laconic commenting style, and using a somewhat more civilized module on top of it greatly reduces the amount of needed Smalltalk-code.

The complete interface and implementation of the module can be found from appendices. The most often used functions of it are:

dbEngine.dll protocol (C)

Constructors

dbeConnect (dbName, userId, passWord)	Returns a new connection handle to the database with access rights as granted to the given user.

Impulses

dbeCommit (conn)	Commits any changes written by the user to the database.
dbeRollback (conn)	Rolls the changes back to the previous commit-call.
dbeCompile (conn, statement)	Compiles the given SQL-statement and attaches it to the connection.
dbeExecute (conn)	Executes a previously compiled SQL-statement.

dbeSetVar (conn, varNr, value)	Sets a bind-variable in a compiled statement.
Cursor movements	
dbeToStart (conn)	Makes the first row in a query as current.
dbeForth (conn)	Makes the next row in a query as current.
Attributes	
dbeGetCol (conn, varNr)	Returns the value of the nth column of the current row in a query.
dbePoints (conn)	Tells whether the cursor points to any row.
Destructor	
dbeDisconnect (conn)	Disables the connection and relieves resources reserved for it.

The functions are typically called in an access cycle [SQLB88] which includes the following steps:

1. The database is connected to by the dbeConnect method. The handle returned will be used in the later calls.
2. An SQL-statement is compiled or a precompiled statement is retrieved.
3. The bind variables in the statement are set.
4. The statement is executed.
5. If the statement was a query, the data is fetched by moving the cursor through all rows and picking the columns from them one-by-one.
6. If the statement involved writing to the database, the changes are committed.
7. The database is disconnected.

This small C-program goes through the tasks table of the given range and prints out it's contents. For clarity, error management is left out.

```
int main (int argc, char* argv[])
{
 connection c;

 c = dbeConnect ("JOBS", "TEST", "TEST");
 dbeCompile (c,
  "SELECT ID, WORKNR, INFO \
```

```
    FROM TASKS \
    WHERE ID BETWEEN :1 AND :2");
  dbeSetVar (c, 1, argv [1]);
  dbeSetVar (c, 2, argv [2]);
  dbeExecute (c);
  for (dbeToStart (c); dbePoints (c); dbeForth(c))
    printf ("Task id = %s, worknr = %s, info = %s\n",
      dbeGetCol (c, 1), dbeGetCol (c, 2), dbeGetCol (c, 3));
  dbeDisconnect (c);
  return 0;
};
```

4.8.3. The type conversion

For representing atomic data values, SQL, C and Smalltalk all have their own type-sets which are mostly incompatible at the binary level. This means that most of the data transferred from SQL to Smalltalk or vice versa has to be converted two times: first into C's internal format and then into the target format. Since C's types are too scarce to preserve all the type-information, dbEngine passes also a constant which tells the original type. The user has to know in which actual c-type each logical dbEngine-type is presented. The type-mapping between SQL and dbEngine goes like this:

SQL	**dbEngine**	**C**
char, char (1)	character	int
char (n) | n>1	string	string
smallint, dec (n, 0) | n <= 4	short_integer	int
integer, dec (n, 0) | 5 <= n <= 9	long_integer	long
dec (n, 0) | n >= 10	very_long_integer	string
number, real, float (n), double precision, dec (n, m) | m > 0	real	string
date	date	string
time	time	string
datetime	timepoint	string

This mapping is quite minimal and does not even try to maximize the performance. In particular, real-numbers could be exchanged in their native format in stead of as strings. On the other hand, using strings helps in avoiding difficult problems when the domain of two approximately compatible types does not match exactly. Also converting the fixed precision numbers (dec (n,m)) into real numbers would cause rounding problems.

4.8.4. DbCursor

The class DbCursor operates on the dbEngine about at the same level as the c-program example. Its main task is to convert the data received from the dbEngine-module to Smalltalk instances and vice versa. This is done according to the following mapping:

dbEngine	Smalltalk/V PM
character	Character
string	String
short_integer	SmallInteger
long_integer	Integer
very_long_integer	Integer
real	Fixed
date	Date
time	Time
timepoint	-

The knowledge about which C-type to expect is encoded into the Smalltalk classes. Each class mentioned on the right has a new class method called "dbData:", which returns a new instance of the class containing that value.

4.9. An example call

To get an overview of the multiple levels of software in the database interface, let's see how one tuple is fetched from a keyed table in detail.

The action starts when an instance of a keyed table receives the "at:"-message. Its implementation is like this:

```
at: aKey
      "Returns the tuple with the given key. Uses the
       selectThisStatement-method. The parameter is a
       dictionary with appropriate keyfields."
   cursor compile: (self class selectThisStatement).
   ^self getFirst: aKey
```

The "selectThisStatement"-message is targeted to the subclass of KeyedTable which the receiver is an instance of. The statement could be for instance like this:

```
selectThisStatement
```

```
^'SELECT ID, WORKNR, NAME, INFO FROM TASKS, WORKS ',
'WHERE WORKNR = NR AND ID = :kIdent'
```

Data is fetched from two tables. Now, this string is given to the dbCursor instance "cursor" which sends it further:

```
compile: aStatement
    "Compiles an SQL-statement. (compStore is used
    for efficiency.)"
  DbLib compStore: lowHandle
      theStmnt: aStatement ofLength: (aStatement size).
  ^self checkStatus
```

DbLib is class variable which contains the dynamic link-library handle to dbEngine.dll. It has been set in the system initialization. The statement is actually compiled only if it is not yet compiled and stored by some previous action, otherwise it is just restored. The DbEngineDLL-class receives the call:

```
compStore: aHandle theStmnt: aString ofLength: anInt
    "Compiles the given SQL statement when called the first
     time, restores it otherwise."
  <api: DBECOMPSTORE short struct ushort long>
  ^self invalidArgument.
```

Now the call has proceeded to the dbEngine.dll. It gives the statement for the database server to be compiled and initializes the select-buffers:

```
IMPULSE dbeCompStore
          (connection c, string SQLStatement, int strLen)
{
  char      genName [SQLMBNL];
  short     result;

  checkCur (c);
  generateName (SQLStatement, strLen, genName);
  // Try to restore the statement first
  if (sqlret (ucurs(c), genName, NULL_TERMINATED) == OK)
    setUpCompiledStatement (c);
  else {
    // Hasn't been compiled before or is invalidated
    // by changes in the database structure.
    // Store and restore the statement.
    begin (c);
    result = sqlsto (ucurs(c), genName, NULL_TERMINATED,
      SQLStatement, strLen);
    if (result == OK) {
      result = sqlret (ucurs(c), genName, NULL_TERMINATED);
```

```
            if (result == OK)
               setUpCompiledStatement (c);
            else if (result != DBOECNF)
               signalError (cursPtr (c), (status) result);
         } else
            signalError (cursPtr (c), (status) result);
      };
      return connStatus (c);
   }; // dbeCompStore
```

If everything went well, the next thing that happens is that the "getFirst"-message is send to the keyed table instance by itself:

```
getFirst: aKey
      "Private - returns the next tuple to the given key from a set
      defined by previously compiled sql-statement."
   self bindToKey: aKey.
   cursor execute.
   cursor toStart.
   (cursor points)
      ifTrue: [^self getValue]
      ifFalse: [^nil].
```

The keyvalue is bound to the program variables like this:

```
bindToKey: aKey
      "Private - The column names in the key-subtuple are
      preceded by 'k's to separate them other fields in
      the statement."
   aKey keysDo: [:each |
      cursor setNamedVar: ('k', each) to: (aKey at: each) dbData]
```

Binding proceeds to cursor and dbEngine...

```
setNamedVar: aName to: anObject
      "Sets a named bind variable to the given object."
   self setVar: (
      DbLib bindNumber: lowHandle ofName: aName
         ofLen: (aName size))
      to: anObject.

IMPULSE dbeSetVar
            (connection c, int ofNumber, string toThis, int valLen)
{
   checkCur (c);
   begin (c);
   setBindValue (cursPtr (c), ofNumber, STRING,
      (unsigned char*) toThis, (unsigned char) valLen);
   return connStatus (c);
}; // dbeSetVar
```

The statement is executed and cursor set to the start of the result set. Finally the value is fetched by the "getValue"-method of the keyed table instance (inherited from Relation)

```
getValue
    "Private - the rows in a relation or a table are returned as
    instances of the row-class which is specified for Relation,
    Table and their subclasses."
  | rowClass |
  rowClass := self class rowClass.
  rowClass = Array ifTrue: [
    ^rowClass dbData: cursor value]
  ifFalse: [
    ^rowClass dbData: cursor namedValue]
```

The cursor builds a dictionary of the data it fetches from the database by putting the column name as a key and associating the value to that.

```
namedValue
    "Returns a dictionary which contains the names of
    the columns and their values as instances of
    their native classes."
  | result |
  result := Dictionary new.
  1 to: (self nbrofColumns) do: [:col |
    result at: (self colName: col) asLowerCase
      put: (self colValue: col)].
  ^result
```

In the "colValue"-method it has to decide how the data is converted into a Smalltalk-object. In general, this happens by the dbData: -method so it is enough to know the class.

```
colValue: colNbr
    "Returns the value of the given column in a query
    as an instance of its native class"
  | dataType |
  dataType := DbLib colDataType: lowHandle ofColumn: colNbr.
  (DbLib isNull: lowHandle column: colNbr)
    ifTrue: [
      dataType = DbeString ifTrue: [^String new:0]
        ifFalse: [^nil]]
  ifFalse: [
    dataType
    when: DbeCharacter do: [
      ^Character dbData: (self colCharValue: colNbr)];
    when: DbeString do: [
```

```
        ^self colStrValue: colNbr];
      when: DbeShortInteger do: [
        ^Integer dbData: (self colIntValue: colNbr)];
      when: DbeLongInteger do: [
        ^Integer dbData: (self colLongValue: colNbr)];
      when: DbeVeryLongInteger do: [
        ^(self colStrValue: colNbr) asInteger];
      when: DbeReal do: [
        ^Fixed dbData: (self colStrValue: colNbr)];
      when: DbeDate do: [
        ^Date dbData: (self colStrValue: colNbr)];
      when: DbeTime do: [
        ^Time dbData: (self colStrValue: colNbr)];
      when: DbeTimePoint do: [
        "No class for timepoint yet!!"
        ^self colStrValue: colNbr]]
```

This constructed raw database tuple is then passed to the Tuple-class as an argument in the "dbData:"-method. It delegates the problem to the application-specific subclass's "dbAssociationPut:" -method, which should know what to except from the database.

```
dbData: aTuple
    "Private - construct a new object according to a row
     of column-name-value-associations fetched from a
     database table."
  aTuple associationsDo: [:each |
    self dbAssociationPut: each]
```

There is a default-method for this in Tuple, but if the column names in the database differ from the field names in the tuple the conversion has to be done manually:

```
dbAssociationPut: anAssociation
    "The names differ in the tuple and in the database."
  (anAssociation key)
    when: 'id'      do: [^self at: #ident put: anAssociation value];
    when: 'name' do: [^self at: #workName put:
      anAssociation value].
  super dbAssociationPut: anAssociation
```

Apparently, what goes under the label "id" in the database is called #ident in the object and "name" translates to #workName. Were there any other data conversions to be made, this is the place.

All in all, fetching a tuple from the database can be made ex-tempore at the application level like this:

```
tuple := (TasksTable openIn:
```

```
(Database openAsSysadmin: 'JOBS')) at:
  (Dictionary new at: #ident put: 'OOPoS-Proto')
```

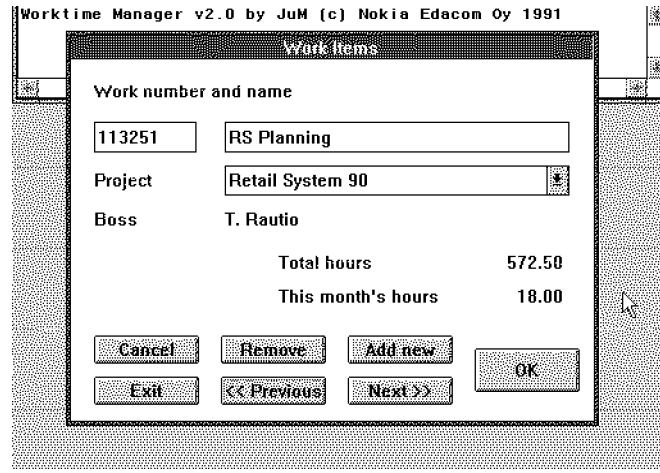
Looking at the amount of code that is hidden behind this short expression perhaps tells something of the abstracting powers of Smalltalk.

5. IMAGE OF AN APPLICATION: THE USER INTERFACE

For making a real application we need yet tools for the user-interface: something by which tuples and relations can be viewed, manipulated and managed on the screen. What follows is a short description of such tools. Since the subject is somewhat out of the main theme, the details are mainly overlooked where they do not have significance from the database's point of view.

5.1. Basic dialog elements

The basic building block in making an application in a semi-graphical environment like OS/2's Presentation Manager is a dialog. It is a form-like window which contains a set of controls by which the underlying data can be viewed and manipulated. The most important controls are texts, entry-fields, buttons and menus.



A typical dialog with some header and information texts, entry fields, one combo-box and pushbuttons.

Each dialog is attached to a tuple so that each entry field, check-box, radio-button or other control is connected to some component of it. When data is entered via a control, it is send to the tuple by the "at:put:"-method. If the tuple accepts the new value, the dialog is refreshed and at the same time it is checked whether some other fields change as well. If so, the new values of them are also shown on the display.

The type of the control and the domain of the connected tuple's component are closely coupled. This is how the controls are used with different components:

- An entry field is used with a string, a numeric measure or with any other simple type that can be expressed in one string.
- A combo-box is like an entry field with the exception that the tuple provides a list - with the "choicesAt:"-method - of the allowed alternatives and nothing out of that list can be entered. A variation of this is a list-box, where more of the alternatives are visible.
- A multi-line-edit field is like an entry field with the exception that it can be scrolled horizontally, vertically or both. Consequently, very long texts can be entered in such a field. It is connected to a string-valued component of the tuple.
- A check-box has two states - on and off - and it is connected to a boolean component. Three-valued check-boxes are not supported in this implementation.
- Radio buttons form groups in which one at a time can be active. The button groups are connected to enumeration-valued components. The difference from combo-boxes is that if the domain is in any way dynamic a combo-box should be used. Only if it is quite static a radio-button-group is a good solution.
- An info-field does not allow its contents to be changed. Such a field is used with virtual or otherwise read-only-like tuple components.

The remaining controls contain pushbuttons and menus. These are not connected to tuple's components but to dialog-classes methods. Pushing a button or selecting a menu-item causes a message to be sent to the dialog instance.

A dialog is defined by an OS/2 tool called the Dialog Editor. The result is a resource file, which can be used as such or compiled into an .DLL-file by the resource compiler. Smalltalk supports both ways. When a dialog is defined, the controls are numbered. In the user-defined dialog classes this numbering has to be mapped to Smalltalk's symbols and methods.

5.2. Reusable classes

For each different dialog there has to be an own class. When an instance of the class is created, the dialog is loaded from the program's resources and shown on the display. There are several dialog classes for various purposes, so that making a subclass requires usually only specifying the dialogs id and the mapping between its controls and the tuples components. The class hierarchy looks like this:

```
Window
  ApplicationWindow
    ...
    Inspector
      Debugger
      DictionaryInspector
        TableInspector
          UserDefinedTableInspectors...
    ...
    TextWindow
      ReportWindow
  ...
  DialogBox
    DictionaryDialog
      TupleDialog
        ModelessTupleDialog
          UserDefinedModeless...
        TableDialog
          UserDefinedTableDialogs
        UserDefinedTupleDialogs
```

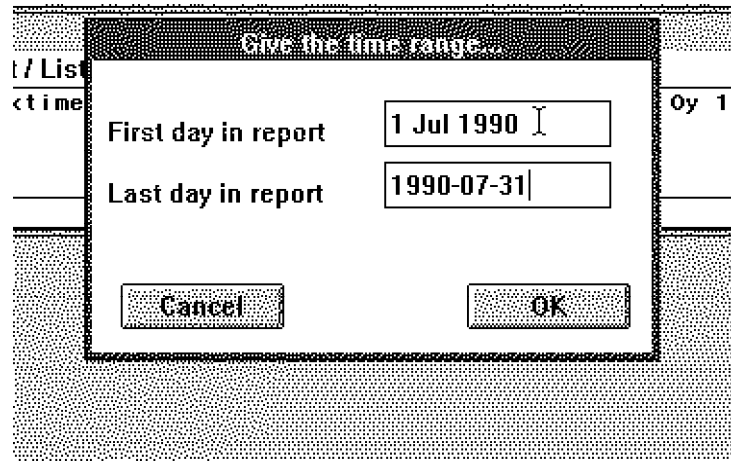
5.2.1. Dictionary dialog

At the basic level Smalltalk/V PM offers quite little support for making dialogs. The class DictionaryDialog is an enhancement which connects the dialog controls to keys of a dictionary. It recognizes the various control types and knows how to handle the input events, so that the entered values are converted into Smalltalk-objects and set as values to the dictionary. Also the new controls of OS/2 version 1.2 - combobox and multiline-edit - are implemented here.

DictionaryDialog is an abstract superclass.

5.2.2. Tuple dialog

An instance of TupleDialog class looks typically like this:



Tuple dialog has two predefined pushbuttons - OK and Cancel - in addition of the other controls.

The other controls are defined in application-specific subclasses but the two pushbuttons come as defaults. The dialog is modal, which means that until one of the two buttons is pressed, it keeps the control of the application. The dialog is invoked by the "openOn:"-method, which gets a tuple as argument

```
ApplicDialog openOn: aTuple
```

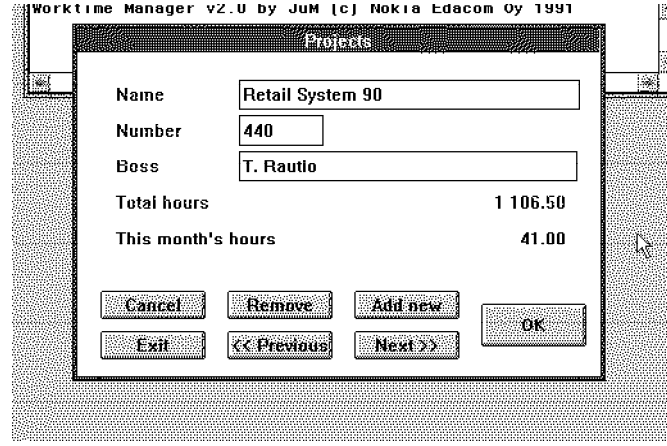
When the dialog is opened, the fields in it are initialized with the values in the tuple. If the dialog is terminated by "Cancel", the method returns "nil", otherwise it returns the updated tuple.

5.2.3. Modeless tuple dialog

A variation of the tuple dialog is a modeless one, which allows also other parts of the application to get control. The current tuple-value can be queried from it any time by the "contents"-method.

5.2.4. Table dialog

Another enhancement of the tuple dialog is the table dialog, which is opened with a keyed table instance as a parameter. It looks like this:



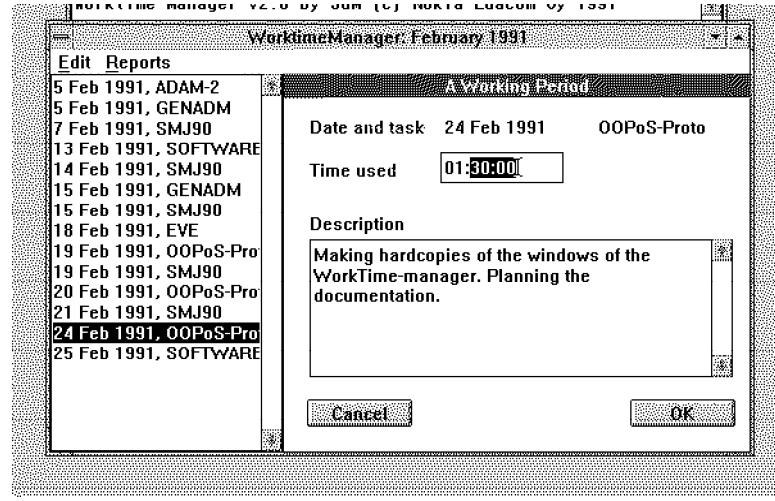
A table dialog adds the buttons needed for browsing the table and modifying it

Items can also be accessed from the table by entering a key value.

5.2.5. Table inspector

Making a table dialog is one way of managing tables in the database, but not always very appropriate. For one reason, it shows only one tuple at a time and gives no idea of what else there is in the table. Furthermore, the user has to select items blindfolded: when entering the key value she cannot know whether there is a tuple with it in the table or not. If the result is unexpected, that may be confusing.

A better model can be borrowed from the dictionary inspector. As was showed in chapter 4, one can be used for browsing tables as such. Nevertheless, it is a good idea to replace the text pane by a modeless dialog which is better suited for handling tuples. Such a hybrid window looks like this:



A table inspector consists of a list pane with the key-values in it and a modeless tuple dialog which holds the current tuple.

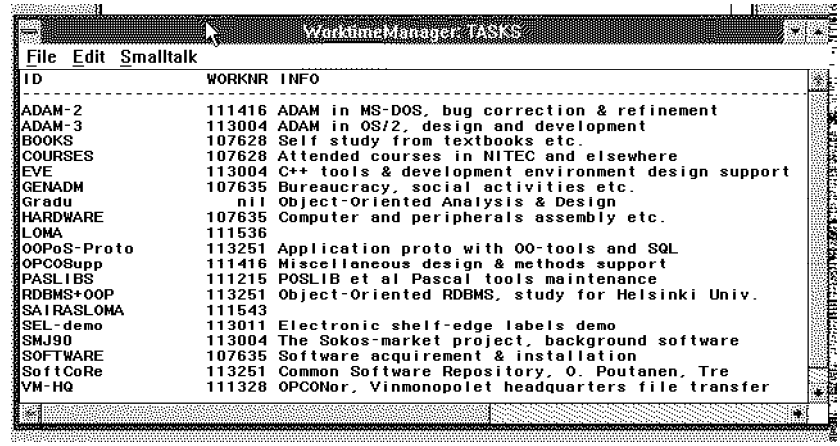
Under the "edit" menu the actions for adding new items or removing existing ones can be selected. Changing an item can be done simply by entering the new data into the fields and pressing "OK". The key-fields shouldn't be changed in the dialog, so they are defined as info-fields.

5.2.6. Report Window

Yet another tools is needed for completing the minimum toolkit for user-interfaces: a reporting tool. For this purpose there is the class ReportWindow, which is a subclass of a plain text window. Such a window can be attached to a query by the method:

```
myReport := ReportWindow new contents: aRelation.
```

And when the message "myReport show" is called, the window is opened and the rows in the relation printed to it as text. The column widths and titles get their defaults from the query but they can be redefined by report window's instance methods at will. The result would look something like this:



The basic report window contains just the column names as a header and the contents as text.

All the inherited TextWindow methods are naturally available for adding whatever headers, footers and other info into the text-pane.

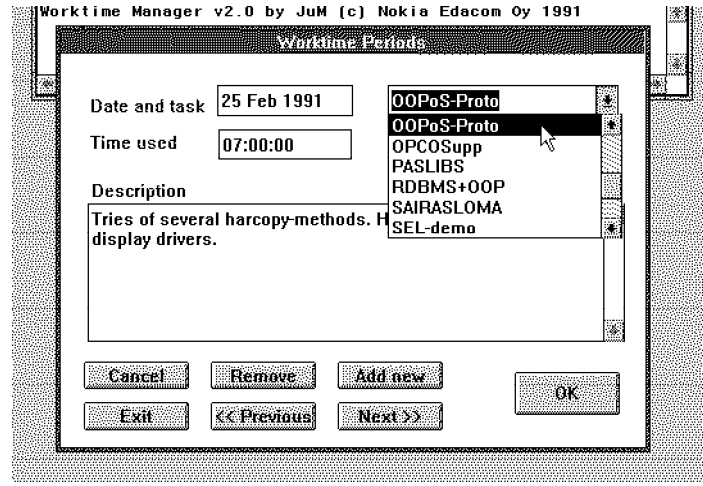
Under the "File" menu the user can select "print" which causes the contents of the text-window to be sent to the printer. The text can also be exported via the clipboard or saved into a file.

5.3. Guarding inclusion dependences

In chapter 4 it was stated that the consistency of the database regarding inclusion dependencies should be ensured already in the user interface. The mechanism of it goes like this:

1. In the tuple where the depending attribute X is, define a class method called "setXChoices: aCollection". In that method, the collection (array, set) is stored into a class variable.
2. When the application is started or a dialog is opened, the choices are initialized. The argument can be easily formed from a query by the "asSet" or "asSortedCollection" -method.
3. When data is entered in attribute X, it is checked that the new value is in the set of the allowed values. If not, it is rejected.

By help of combo-boxes and list-boxes such a mechanism can be made very easy and intuitive to use. When the set of the allowed values is not very large this is a good solution:



The "task"-attribute of an account accepts only existing task-names.

The solution above is optimum, since it keeps the checking inside a tuple without it having to know anything about the table. It cannot be used, however, if the set of allowed values contains several hundreds items or more. In such a case the tuple has to make a query to the database each time a new value is entered. That needs not to be a problem since also the checking code can always be given to the tuple as a one-argument block for instance like this:

```
MyTuple setXCheck: [:newVal |
   myRelation bindTo: newVal.
   myRelation size > 0].
```

Where the myRelation-variable contains a relation like this:

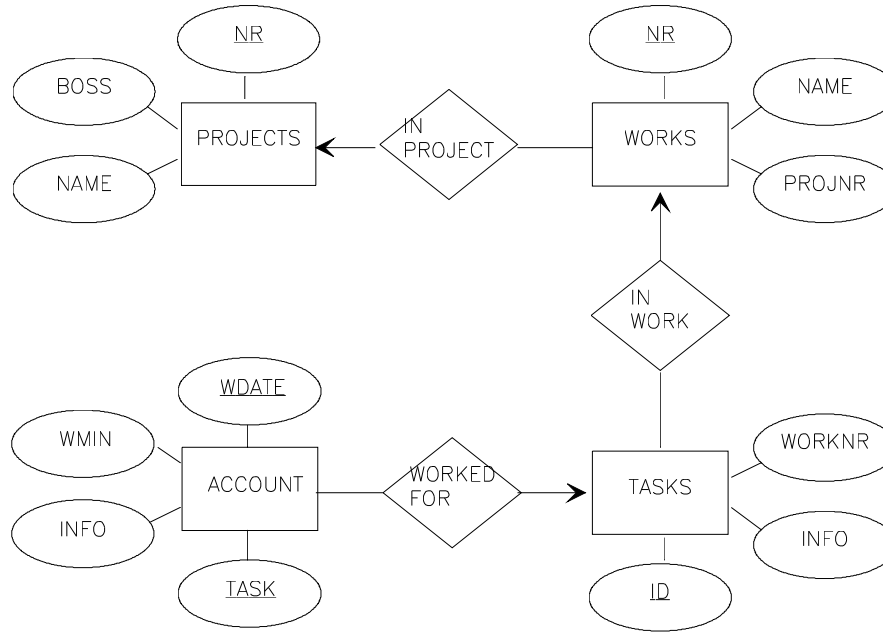
```
myRelation select: '*' from: 'YS'; where: 'X = :1'.
```

After all - this is Smalltalk.

6. A CASE STUDY: THE WORKTIME MANAGER

The best way to test our database interface and tools is to make a moderately sized application with them. A couple of times before I have tested new application-development methods by making a worktime-manager, and this is what I decided to do now, too.

The database is designed with simplicity in mind, but so that there are enough dependencies to test the model. It contains four tables and thirteen attributes.



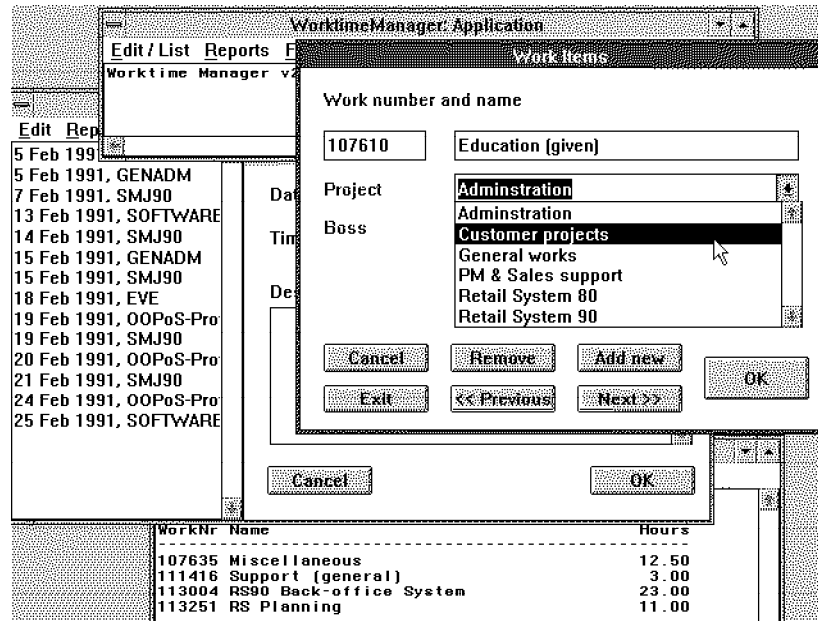
An ER-diagram of the database. Accounts, tasks, works and projects form a four-level hierarchy in the database.

The daily working periods are collected into the 'account'-table. The attributes tell for what task the work was done (TASK), how long did the period last (WMIN), at which date did it start (WDATE) and a short description of what approximately was done (INFO). The items in the 'tasks'-table represent personal activities like working for a project or being on a sick-leave. The key is a short mnemonic symbol (ID) and in addition fo that there is a short description of the task (INFO) and a link to 'works'-table. It, in turn, contains items for the resource units with own budgets. The works have numbers (NR) and names (NAME) and they are further mapped to projects (PROJNR). The 'projects'-table contains the most coarse partition of the activities from where it can be seen who manages each project (BOSS) and the activities belonging to it

and the name (NAME) and the number (NR) of the projects. The appendices contain a full SQL-definition of the tables.

The database is in third normal form: there are no undesired functional dependencies. Consequently, there are plenty of inclusion dependencies. For instance, answering the query "for whom did I work today" requires three inclusion dependencies to be satisfied: for the account there must be a task, for the task there must be a work-item, and the boss-name is found only if the work-item is connected to a project. In the application there are also virtual attributes like the total amount of working hours made for a project, which requires that the chain is gone through to the opposite direction.

The application managing the database should look something like this:

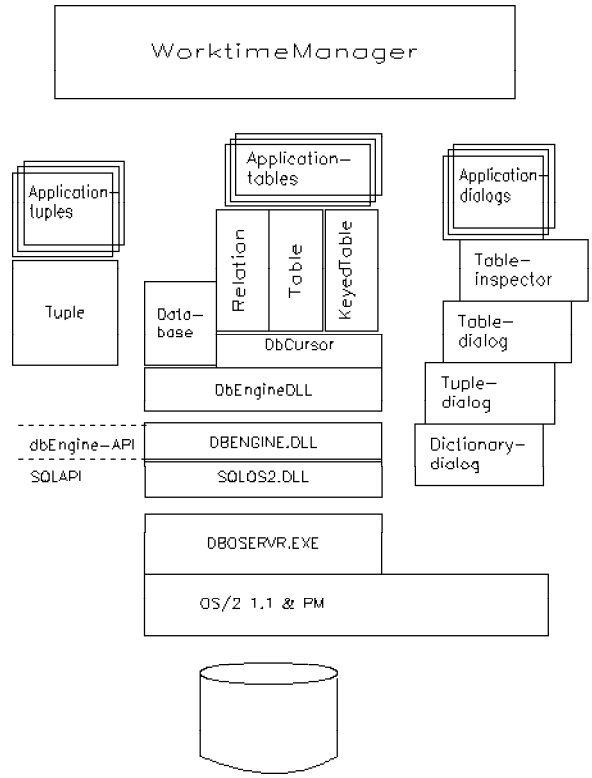


The WorktimeManager consists of table dialogs, inspectors and reports. On top there is a text window from which other windows are started.

There should be means of managing all the four tables. Two kinds of reports are needed: one which groups the hours according to the work number, and another which gives a detailed description of every day's activities. The reports can be taken on a selected month or a arbitrary interval.

6.1. The environment and overall structure

The prototype is implemented in a Nokia MikroMikko 4 m338-computer with a 20 MHz 80386 processor, 8 megabytes of central memory and a 140 MB hard disk. The machine runs OS/2 1.2, which includes the Presentation Manager -user interface. The programming language is Smalltalk/V PM version 1.1 by DigiTalk Inc. Gupta Technologies' SQLBase, version 4.0.6 was used as the database server.



The software structure of the example application in terms of who uses who. The most specific classes are on the top.

Starting from down, there is the SQLBase's multi-user database-server DBOSERVR.EXE which runs as a separate OS/2 process. It is connected via a dynamic-link library SQLOS2.DLL, which creates a connection to the server by using OS/2's queues or pipes. The DLL provides for the user the SQLAPI-application interface, which is proprietary to the product - one reason not to use it directly.

The next level consists of the dbEngine.dll with its own API. The module is written in Microsoft C version 6.0A for OS/2. Its specification and source-code can be found from appendices.

The layers on top of the dbEngine-API are implemented in Smalltalk/V PM and they are described in previous chapters. The tuple-related section on the left is pure Smalltalk-software, while making the dialogs required to use some of the Presentation Manager's services directly.

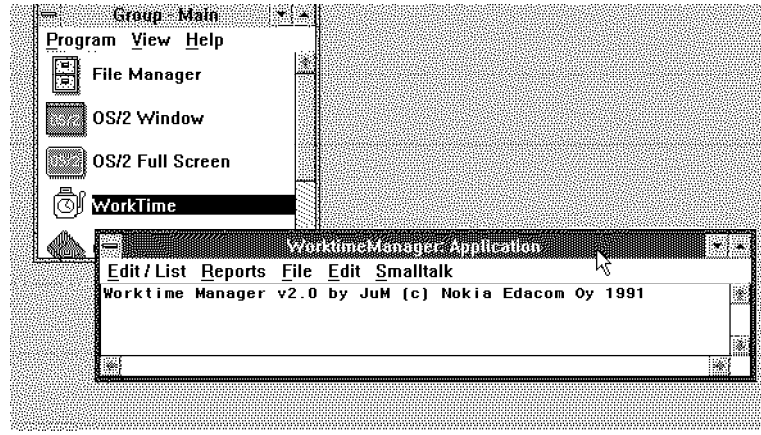
On top there are the application-specific classes: AccountTuple, AccountsTable, AccountsDialog and so on. In principle, one tuple class, one table class and one or two dialog classes for each table in the database. One table inspector has been implemented for accounts, and the WorktimeManager-class on the top binds all this together.

6.2. Evaluating the application

On remarkable feature in the application is the fact that it works. Little is left to be said about the database interface: since the previous chapters have been written after the application, the problems encountered have already been solved. Apart from efficiency, there are no open questions. There's no point in inserting all the source code here, either, but it can be found from the appendices. Hence, this chapter contains just a few notions about programming in Smalltalk and about the system in general.

6.2.1. Starting and how long it takes

The application is started from OS/2 task manager's Main group. It takes 15 seconds from it to get the top application window opened and ready to access input.



When started, the WorkTime manager opens its top window.

Getting the month-inspector opened (see picture at page 55) takes some 11 seconds more. It shows all of the account-markings of that month. After this, adding new items, selecting, modifying and deleting existing ones are all operations which take less than one second to complete.

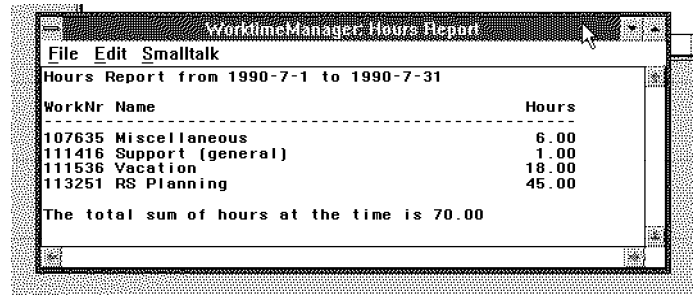
One could imagine that collecting the data from various tables takes a lot of time. For instance, each time a project-tuple is fetched from the database, the following code is executed:

```
getValue
    "Private - overrides the default method. Data is needed
    also from the account-table for the hour-sums."
  | result totalHours monthsHours projNr |
  result := (self class rowClass) dbData: cursor namedValue.
  projNr := result at: #projectNumber.

  accountTable select: 'SUM(WMIN)/60' from: 'ACCOUNT';
    where: 'TASK IN (SELECT ID FROM TASKS WHERE ',
      'WORKNR IN (SELECT NR FROM WORKS WHERE ',
      'PROJNR = :1))'.
  accountTable bindTo: (Array with: projNr).
  totalHours := accountTable row: 1 col: 1.
  totalHours isNil
    ifTrue: [totalHours := Fixed new: 0.0]
    ifFalse: [totalHours := Fixed dbData: totalHours].
  result at: #totalHours put: totalHours.

  accountTable select: 'SUM(WMIN)/60' from: 'ACCOUNT';
    where: ' WDATE BETWEEN :2 AND :3 AND ',
      'TASK IN (SELECT ID FROM TASKS WHERE ',
      'WORKNR IN (SELECT NR FROM WORKS WHERE ',
      'PROJNR = :1))'.
```

```
accountTable bindTo: (
   Array with: projNr
      with: (Date today firstDayOfMonth)
      with: (Date today)).
monthsHours := accountTable row: 1 col: 1.
monthsHours isNil
   ifTrue: [monthsHours := Fixed new: 0.0]
   ifFalse: [monthsHours := Fixed dbData: monthsHours].
result at: #thisMonthsHours put: monthsHours.
^result
```

However, it seems that the response times depend on the amount of account-tuples which are fetched. When there is little of them, the response time to "at:", "nextTo:" and "prevTo:"-methods is about 4 seconds, but when there are plenty of them it gets 7-8 seconds long. In any case, handling a table dialog is irritating slow compared to the table inspector. This is probably partly due to the fact that the latter uses only one SQL statement which is ready to be executed any time while the former switches the statement quite often. Also the result set in the "at:"-method is smaller than in the "nextTo:" and "prevTo:" -methods.

Printing a report may take some time as well. The required hours-report takes eight seconds to get completed.



The general hour report.

For making it requires the following code to be executed:

```
hoursReportTo: aReport of: dates
      "Prints the hours done under work-numbers to the given
      report."
   | query |
   query := Relation openIn: dBase.
   query select: 'WORKS. NR, WORKS. NAME, ',
         SUM (ACCOUNT.WMIN)/60'
      from: 'WORKS, TASKS, ACCOUNT';
      where: 'WORKS. NR = TASKS. WORKNR ',
         'AND TASKS. ID = ACCOUNT. TASK ',
```

```
        'AND ACCOUNT. WDATE BETWEEN ',
            :firstDay AND :lastDay';
    groupBy: 'WORKS.NR, WORKS.NAME'.
query bindTo: dates.
aReport show: 'Hours Report from ',
    (dates at: #firstDay) asISOstring,
    ' to ', (dates at: #lastDay) asISOstring.
aReport cr; cr.
aReport contents: query.
aReport
    columnTitle: 1 to: 'WorkNr';
    columnWidth: 1 to: 6;
    columnTitle: 2 to: 'Name';
    columnTitle: 3 to: 'Hours'.
aReport show.
query select: 'SUM (ACCOUNT. WMIN)/60' from: 'ACCOUNT';
    where: 'ACCOUNT. WDATE BETWEEN ',
        ':firstDay AND :lastDay ',
        'AND ACCOUNT. TASK IN ',
            '(SELECT TASKS. ID FROM TASKS ',
                'WHERE TASKS. WORKNR IN ',
                '(SELECT WORKS. NR FROM WORKS))'.
query bindTo: dates.
aReport cr.
aReport show: 'The total sum of hours at the time is '.
aReport  show: (query row:1 col:1) asString.
aReport cr.
query close
```

All in all, the response times are just about in the limits of what can be accepted. Adding one transaction, which is typically made once a day, can be done in less than a minute. Viewing the transactions of the same month is sufficiently fast. Browsing through the tasks, work-items and projects is too slow, but acceptable because these tables are rarely modified.

The response times are measured with the database server running in the same machine as a foreground process. All of the used SQL-statements are precompiled in the database. The database contained 327 account tuples, 19 task tuples, 13 work-item tuples and 6 project-tuples, and it occupied a good megabyte of disk space.

6.2.2. Programming efficiency

The whole project including getting acquainted with the subject and environment, writing two versions of this document and programming all of the software took the total of 628 hours distributed on a year's period (a complete list of all activities can be

found from appendix B). The programming result is about 7500 lines of Smalltalk- or C-code and over a hundred pages of document (almost 100 lines of code and a good one page of document per 8 hours). While this cannot be considered as in any way revolutionary effective, much of the work has been in learning how to use Smalltalk and building the basic, reusable classes. Therefore, it is difficult to estimate the effort of building new applications by using the existing base-classes. However, one measure can be taken from the distribution of the source-code, which is like this:

Module	Lines of code	% of total
dbEngine etc. (C)	2903	39
Database base classes	1450	19
User-interface base classes	807	11
General utilities and enhancements	1077	14
WorktimeManager application	1259	17

If the application's 17% share is generalized to the total effort it would mean that an application like the one documented here would require 107 hours or two weeks until finished. Whether this is a good or a bad result is left up to the reader to decide. In any case, it should not be taken too literally, since the estimate has several weak points: it compares C source lines to Smalltalk sources and it ignores the fact that after writing 6300 lines of tools one is exceptionally well prepared to write 1200 lines of application code.

6.2.3. Other programming aspects

Smalltalk's steep learning curve has often been referred to. It has been said like hitting one's nose to a cliff, when the vast amount of the existing code and classes gets clear. However, there are good reasons to asks whether the task of learning to use existing code is after all a smaller one than making it all by oneself. Moreover, during its 15-year existence the ideas and protocols in Smalltalk have become remarkably clear and logical. Alas, this cannot always be said about some more recent 'small miracles', which promise to turn every man into an application factory.

On the other hand, once one gets to know the environment, an addiction is inevitable. The environment is robust and user-friendly: almost anything can be inspected and changed at will.

During the course of this project there was not a single case when I would have had to back off from an idea because it was impossible to implement. On the contrary, I was often surprised how ideas which at first sight seemed bizarre turned out to work very well in the end.

The bad sides of Smalltalk are also often pointed out: it is a memory hog and it tends to run slowly at times. Well, what comes to this, the size of the worktime application is 683 485 bytes plus 10 kilobytes from the DLL which includes the displays. With this there comes a number of other DLL's, though. However, having some experience in OS/2 PM programming in C++ I cannot consider this as terribly much, especially because it contains the complete debugging and program development environment. And at least in this application the only true bottle-neck was the database.

7. CONCLUSIONS

In the previous version of this document the last chapter was called 'conclusions and criticism'. The reason why this second version was written in the first place was my desire to get the 'criticism' word away. Now it can be said that the model of an object-oriented interface to a relational database is about as sound and working as I can possibly see.

It seems like object-oriented programming fits quite well on top of any database to fill the empty gaps and equip the mindless tuples with a little bit of intelligence so that the database remains consistent even when exposed to careless users. Implementing the constraint checking inside tuple- and table-instances makes the system robust and reliable without making it intolerable complex at the same time.

Of course there is a number of open questions, and many aspects of database design which are not addressed at all. The first and most serious problem, of course, is the performance. Fetching data from the database tends to be too slow for interactive usage, even if plenty of hardware resources were used.

It may be so that the main reason of the problem is the irreconcilable conflict between the very basic ideas in relational databases and object-oriented programming, after all. The relational database servers are designed with maximum throughput in mind. They are capable of getting the most out of hardware resources and maximizing the amount of transactions in second. It is a regrettable fact, however, that maximum throughput means long response times and vice versa, because for being able to answer quickly the machine has to be idle and if it is idle it doesn't get any work done. Using a relational database server for a medium-size desktop application is like using an express train for delivering letters to the next block.

There are ways of fighting against this problem. One of them is to utilizy concurrency in the user-interface, so that reading and writing are performed asynchronously in a pipeline. One can imagine a couple of clever algorithms for guessing what the user intends to do next and get prepared for that in advance. Also the specifications of the applications written today reflect older environments, where there were less memory to be used but where also the way to the disk sectors was much shorter.

Whether it is a good idea to combine a relational database with object-oriented programming remains to be seen. After all, both are kind of industrial standards or becoming such. Anyhow, if somebody decides to do so I hope that she will get a few ideas from this paper to help.

8. BIBLIOGRAPHY

|Fink90] Rich Finkelstein: Multiuser Databases: the SQL. BYTE, May 1990; p. 136 - 150.

[Gold89] Adele Goldberg & David Robson: Smalltalk-80, the Language. Addison-Wesley, Reading, Massachusetts, 1989.

[Lieb86] Henry Lieberman: Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems. SIGPLAN Notices, vol. 21 (11) (OOPSLA '86). 1987. p. 214 - 223.

[SQLB88] SQLBase Technical Reference Manual, version 3.4. Gupta Technologies Inc, Menlo Park, California. 1988.

[Tell89] Ernest R. Tello: Object-Oriented Programming for Artificial Intelligence. Addison-Wesley, Reading, Massachusetts. 1989.

[Udel90] Jon Udell: Bridging Troubled Waters. BYTE, April 1990; p. 225 - 230.

[Ullm88] Jeffrey D. Ullman: Principles of Database and Knowledge-base Systems. Computer Science Press, Rockville, Maryland. 1988.

[Wins84] Patrick Winston: Artificial Intelligence, second edition. Addison-Wesley, Reading Massachusetts. 1984.

9. APPENDIX A: WORKTIME DATABASE

```
CREATE TABLE PROJECTS
(
    BOSS    CHAR(36),
    NAME    CHAR(30),
    NR      INTEGER
)

CREATE UNIQUE INDEX PROJINX1 ON PROJECTS (NR)

CREATE TABLE TASKS
(
    ID      CHAR(12),
    WORKNR  INTEGER,
    INFO    CHAR(100)
)

CREATE UNIQUE INDEX TASKINX1 ON TASKS (ID, WORKNR)
CREATE UNIQUE INDEX TASKINX2 ON TASKS (WORKNR, ID)

CREATE TABLE WORKS
(
    NR      INTEGER,
    NAME    CHAR (36),
    PROJNR  INTEGER
)

CREATE UNIQUE INDEX WORKINX1 ON WORKS (NR, PROJNR)
CREATE UNIQUE INDEX WORKINX2 ON WORKS (PROJNR, NR)

CREATE TABLE ACCOUNT
(
    WDATE   DATE,
    TASK    CHAR(12),
    INFO    CHAR(200),
    WMIN    DECIMAL(12, 0)
)

CREATE UNIQUE INDEX ACCINX1 ON ACCOUNT (WDATE,TASK)
CREATE UNIQUE INDEX ACCINX2 ON ACCOUNT (TASK, WDATE)
```

10. APPENDIX B: WORKING LOG

Date	Project	Hours	Explanation
5 Nov 1989	OOPoS-Proto	4.00	Smalltalk/V PM:n referointi s.mailin taululle 384
15 Nov 1989	RDBMS+OOP	3.00	Hallinnolliset järjestelyt
20 Nov 1989	RDBMS+OOP	4.00	
21 Nov 1989	RDBMS+OOP	1.00	
22 Nov 1989	RDBMS+OOP	2.00	
18 Dec 1989	RDBMS+OOP	2.00	
10 Jan 1990	RDBMS+OOP	4.00	Smalltalk/V PM:n installointi
16 Jan 1990	RDBMS+OOP	4.00	
23 Jan 1990	RDBMS+OOP	7.50	
7 Feb 1990	RDBMS+OOP	2.00	Smalltalk/V PM
8 Feb 1990	RDBMS+OOP	9.00	Palaverointia, demo yms.
26 Feb 1990	RDBMS+OOP	2.00	Kokeilua
2 Mar 1990	OOPoS-Proto	11.00	Käynnistyspalaveri Tampereella
12 Mar 1990	OOPoS-Proto	3.00	Projektisuunnitelma
16 Mar 1990	RDBMS+OOP	4.00	Tello: OOP for AI
19 Mar 1990	OOPoS-Proto	6.00	SQLBase installonti
20 Mar 1990	OOPoS-Proto	8.00	SQLBase inst. & testitietokanta
21 Mar 1990	OOPoS-Proto	7.50	SQLBase API
22 Mar 1990	OOPoS-Proto	8.50	Smalltalk external calls
23 Mar 1990	OOPoS-Proto	4.00	Low-level testing
26 Mar 1990	OOPoS-Proto	5.00	OO-database modeling
27 Mar 1990	OOPoS-Proto	7.50	OO-database modeling
28 Mar 1990	OOPoS-Proto	6.00	OO-database modeling
29 Mar 1990	OOPoS-Proto	6.00	OO-database modeling
30 Mar 1990	OOPoS-Proto	4.00	OO-database modeling
1 Apr 1990	OOPoS-Proto	6.00	Status reporting
16 Apr 1990	RDBMS+OOP	2.00	SQL classes design
18 Apr 1990	OOPoS-Proto	6.00	Database model documentation
19 Apr 1990	OOPoS-Proto	9.00	A meeting in Tampere
21 Apr 1990	OOPoS-Proto	3.00	Specification of the dbEngine-module
28 Apr 1990	OOPoS-Proto	3.00	dbEngine-interface
2 May 1990	OOPoS-Proto	7.00	dbEngine-implementation
3 May 1990	OOPoS-Proto	7.50	dbEngine implementation
4 May 1990	OOPoS-Proto	6.50	dbEngine implementation
7 May 1990	OOPoS-Proto	8.00	DbEngine implementation and testing
8 May 1990	OOPoS-Proto	7.00	DbEngine implementation, testing & documentation
9 May 1990	OOPoS-Proto	8.00	Making a MT-DLL out of dbEngine
10 May 1990	OOPoS-Proto	3.00	Debugging dbEngine DLL
11 May 1990	OOPoS-Proto	3.50	Testing and debugging the dbEngine implementation
14 May 1990	OOPoS-Proto	4.50	Coding the Smalltalk-API class DbEngineDLL
15 May 1990	OOPoS-Proto	6.00	Making classes Database and DbCursor
16 May 1990	OOPoS-Proto	3.00	Consulting Jukka Jokinen on dbEngine
17 May 1990	OOPoS-Proto	3.00	Writing SQL-scripts on worktime etc.
18 May 1990	OOPoS-Proto	3.00	DbCursor-class implementation
21 May 1990	OOPoS-Proto	4.00	Implementing classes DbCursor, DbTest
23 May 1990	OOPoS-Proto	6.00	Bindvar support to DbCursor, DbTest nr 3
24 May 1990	OOPoS-Proto	8.00	DbCursor Time, first tries with dialogs
29 May 1990	OOPoS-Proto	2.00	Making a simple dialog-box work
30 May 1990	OOPoS-Proto	3.00	Sending the MpsPos-db to Oulu
31 May 1990	OOPoS-Proto	4.00	Using windows and menus in PM applications
1 Jun 1990	OOPoS-Proto	4.00	Defining dialogs with various controls
5 Jun 1990	OOPoS-Proto	2.00	Petzold: Programming the OS/2 Presentation Manager
6 Jun 1990	OOPoS-Proto	7.00	Using the dialog-controls from Smalltalk/V PM
7 Jun 1990	OOPoS-Proto	8.00	Classes Tuple, TupleDialog, TestTuple, TestDialog
8 Jun 1990	OOPoS-Proto	6.00	Refining the dialog-parts, some documentation
11 Jun 1990	OOPoS-Proto	3.00	Still more refinements to the uif-classes
11 Jun 1990	RDBMS+OOP	4.00	Ullman: Database and knowledge-base systems
14 Jun 1990	OOPoS-Proto	6.00	Named bind-variables to dbEngine
14 Jun 1990	RDBMS+OOP	4.00	Ullman: Database and knowledge-base systems
15 Jun 1990	OOPoS-Proto	13.00	Classes Relation and Table
16 Jun 1990	OOPoS-Proto	10.00	Making WorktimeAccount proto-application
17 Jun 1990	RDBMS+OOP	16.00	Writing 'Using SQL from Smalltalk'
19 Jun 1990	RDBMS+OOP	23.00	Writing 'Using SQL from Smalltalk'
20 Jun 1990	RDBMS+OOP	5.00	Writing the document "Using SQL from Smalltalk"
21 Jun 1990	OOPoS-Proto	3.00	Visiting J. Jokinen, Tampere. Consulting etc.
26 Jun 1990	OOPoS-Proto	4.00	Publishing 'Using SQL from Smalltalk' in s.mail
29 Jun 1990	OOPoS-Proto	2.00	Minor adjustments to Table class, planning

```
 2 Jul 1990 OOPoS-Proto     7.50 Fighting with modeless dialogs...
 3 Jul 1990 OOPoS-Proto     5.00 ArtBasics-dialog, sealing off an SM-application
 4 Jul 1990 OOPoS-Proto     3.00 Design of primitive classes
10 Jul 1990 OOPoS-Proto     9.00 Implementing primitive arithmetic classes
11 Jul 1990 OOPoS-Proto     9.00 Two attempts to install OS/2 1.2. No success.
12 Jul 1990 OOPoS-Proto     6.00 Smalltalk v1.1 upgrade. Primitive arithmetics.
13 Jul 1990 OOPoS-Proto     2.00 EANCode class
16 Jul 1990 OOPoS-Proto     1.00 Sending primitive classes to Tampere
17 Jul 1990 OOPoS-Proto     2.50 Article Manager implementation
21 Aug 1990 OOPoS-Proto     4.00 Documentation, DB-if design
24 Aug 1990 OOPoS-Proto     4.00 Documentation: v1.1 of OODBIF.DOC
27 Aug 1990 OOPoS-Proto     2.00 Adding icons to PM applications.
29 Aug 1990 OOPoS-Proto     3.00 Implementing Relation-class with new design
 3 Sep 1990 OOPoS-Proto     2.00 Re-implementing Relation, Table...
 4 Sep 1990 OOPoS-Proto     6.50 Implementing ArticleManager and -dialog
 5 Sep 1990 OOPoS-Proto     6.00 ArticleManager application
13 Sep 1990 OOPoS-Proto     2.00 Some demonstrations of Article Manager
21 Oct 1990 OOPoS-Proto     3.00 Back to making it with Smalltalk! Oh, bliss!
10 Dec 1990 OOPoS-Proto     2.00 Automatic unloading of changed image
16 Dec 1990 OOPoS-Proto     9.00 Converting the WorkTimeto the new DB impl.
17 Dec 1990 OOPoS-Proto     9.00 Implementing comboBox and MultiLineEdit
18 Dec 1990 OOPoS-Proto     2.50 Defining the dialogs of the WT-appl.
 3 Jan 1991 OOPoS-Proto     6.00 Making dialog and tuple subclasses for the
                                 WorkTime application
15 Jan 1991 OOPoS-Proto    15.00 Making classes DictionaryDialog, TableDialog
                                 etc. and finishing the WorkTime application.
17 Jan 1991 OOPoS-Proto    10.00 Making the rest of the dialogs for the
                                 WorkTime- application and polishing the
                                 database connection. Indexing the database.
18 Jan 1991 OOPoS-Proto     3.00 Combining the dialogs into the Worktime Manager
                                 application. Debugging the dialogs.
23 Jan 1991 OOPoS-Proto     8.50 Making the ReportWindow class, correcting a
                                 number of bugs in dbEngine and further
                                 developing the WorkTime Manager
24 Jan 1991 OOPoS-Proto     7.50 Improving the report facility, polishing the
                                 dialogs and bulding null-value support into
                                 dbEngine. Making class KeyedTable.
28 Jan 1991 OOPoS-Proto     1.50 Correcting the copyright statements from the
                                 headers of exes and DLLs. Fixing a couple of
                                 bugs from DbEngine.
29 Jan 1991 OOPoS-Proto     9.50 Some documentation, changes in the dialog class
                                 hierarchy, improvements in dialogs and first
                                 versions of table inspector.
30 Jan 1991 OOPoS-Proto    11.50 Completing and debugging the tableInspector.
                                 Finishing and testing the WorktimeManager-
                                 application. Making dbEngine more robust.
31 Jan 1991 OOPoS-Proto     3.00 Setting up and giving a preliminary OOPoS-Proto
                                 demonstration to bosses.
19 Feb 1991 OOPoS-Proto     6.00 Making utilities DropDbs and DbsInfo for
                                 running the database server on the background.
                                 Adding the listing facility to the
                                 WorkTime-application and taking hardcopies of
                                 it.
20 Feb 1991 OOPoS-Proto     3.50 Some last refinements to the
                                 worktime-application. Planning of the
                                 documentation.
24 Feb 1991 OOPoS-Proto     1.50 Making hardcopies of the windows of the
                                 WorkTime-manager. Planning the documentation.
25 Feb 1991 OOPoS-Proto     8.50 Tries of several harcopy-methods. Hassling with
                                 OS/2 PM display drivers. Making a printout of
                                 the hardcopies.
 1 Mar 1991 OOPoS-Proto     2.00 Inspecting Smalltalk's multiprocessing
                                 capabilities. Some ideas on how to make the
                                 response times of database shorter by help of
                                 it.
 4 Mar 1991 RDBMS+OOP       6.00 Writing "Using SQL from Smalltalk" 2nd edition.
                                 Introduction and some pictures, mostly fighting
                                 with WfW page numbering.
 6 Mar 1991 OOPoS-Proto     6.00 A couple of new hardcopies, restoring the
                                 nilled pool dictionaries. Downloading and
                                 compiling GNU Diff in order to get a FC-like
                                 program; traps in the first try.
12 Mar 1991 OOPoS-Proto     9.00 Some more hassling with the GNU diff. Finally
                                 made an own diff in Smalltalk - works fine too.
                                 Improving the logAllChanges-method and
                                 implementing a loadSources-method.
13 Mar 1991 RDBMS+OOP       4.00 Writing the second edition of "Using SQL from
```

```
                                        Smalltalk"
14 Mar 1991 RDBMS+OOP       5.00 Writing the second edition of "Using SQL from
                                 Smalltalk"
15 Mar 1991 RDBMS+OOP       6.00 Writing the second edition of "Using SQL from
                                 Smalltalk"
18 Mar 1991 OOPoS-Proto     3.00 Writing the invitation and agenda of the
                                 project's end-meeting. Phone calls to Tampere
                                 and Kajaani.
18 Mar 1991 RDBMS+OOP       3.50 Writing the second edition of "Using SQL from
                                 Smalltalk"
19 Mar 1991 RDBMS+OOP       8.50 Writing the second edition of "Using SQL from
                                 Smalltalk"
20 Mar 1991 RDBMS+OOP       9.00 Writing the second edition of "Using SQL from
                                 Smalltalk"
21 Mar 1991 OOPoS-Proto    12.00 Having the project's end-meeting.
25 Mar 1991 RDBMS+OOP       9.00 Writing the final version for University with
                                 appendices and everything.
```

11. APPENDIX C: DATABASE CLASSES

11.1. Tuple

```
Dictionary subclass: #Tuple
  instanceVariableNames:
    'changed '
  classVariableNames: ''
  poolDictionaries: ''  !


!Tuple class methods !

dbData: aTuple
        "Returns a new tuple with the values taken from the primitive tuple."
    | result |
    result := self new.
    aTuple notNil
        ifTrue: [result dbData: aTuple].
    ^result!

keyFields
        "Returns a list of the keyfields of the Table which is the primary
        storage of the Tuple. Usually implemented by subclasses."
    ^#()!

new
        "Returns a new tuple."
    | result |
    result := super new.
    result nullChanges.
    ^result!

virtualFields
        "Returns a list of the fields which have no physical representation in
        the database. Usually implemented in subclasses."
    ^#()! !



!Tuple methods !

< aKey
        "Compares two tuples in respect of their key-fields."
    | left right |
    (self class keyFields) do: [:each |
        left := self directAt: each.
        right := aKey directAt: each.
        left < right
            ifTrue: [^true]
            ifFalse: [
                left > right ifTrue: [^false]]].
    ^false "The tuples are equal"!

at: aKey
        "Answers the component or nil if the key wasn't found."
    ^self at: aKey ifAbsent: [^nil]!

at: aKey ifAbsent: aBlock
        "If the receiver contains a named method for accessing
        the component, that's used."
    (self respondsTo: aKey asSymbol)
        ifTrue: [^self perform: aKey]
        ifFalse: [^self directAt: aKey ifAbsent: aBlock]!

at: aKey put: aValue
        "If the receiver contains a binary method for setting
        the value that's used."
    | method |
    method := (aKey, ':') asSymbol.
    (self respondsTo: method)
```

```
        ifTrue: [^self perform: method with: aValue]
        ifFalse: [^self directAt: aKey put: aValue]!

changes
        "Returns an ordered collection containing the fields
        which have changed with a changed field. The queue
        is set empty each time 'at:put' method is called."
    ^changed!

choicesAt: aKey
        "This method tests whether the tuple is able to give options on
        what can be entered to a component. If so, it returns a collection
        of them in the textual form."
    | method result |
    ((self directAt: aKey) species = Enumerated)
        ifTrue: [
            result := Set new.
            (self directAt: aKey) class imageDictionary do:
                [:each | result add: (each)].
            ^result]
        ifFalse: [
            method := (aKey , 'Choices') asSymbol.
            (self respondsTo: method)
                ifTrue: [^self perform: method]
                ifFalse: [^Set new:0]]!

dbAssociationAt: aKey
        "Answers the association which is used in the database
        to connect column names to values. By default, it is the same
        as in the tuple expect that the value is in 'dbData' form."
    ^Association key: (aKey asString) value: (self at: aKey) dbData!

dbAssociationPut: anAssociation
        "Puts the value of the given column-name-value-association
        from the database to the tuple. By default, the names are
        supposed to be equal. The value is constrcuted by the dbData-
        method of the receiving component's class."
    | recClass key value |
    key := anAssociation key asLowerCase asSymbol.
    recClass := (self directAt: key) class.
    value := recClass dbData: (anAssociation value).
    self at: key put: value.!

dbData
        "Answer the essential contents of the tuple as a primitive
        Tuple. Only the non-virtual elements are received."
    | result |
    result := Dictionary new.
    self keysDo: [:each |
        (self virtual: each)
            ifFalse: [result add:
                (self dbAssociationAt: each)]].
    ^result!

dbData: aTuple
        "Private - construct a new object according to a row of column-name
        -value-associations fetched from a database table."
    aTuple associationsDo: [:each |
        self dbAssociationPut: each]!

dbKey
        "Returns an instance of own kind with other fields but the keyfields
        pruned away."
    | result keyFields |
    keyFields := self class keyFields.
    result := self class new: (keyFields size).
    keyFields do: [:each |
        result directAt: each put: (self directAt: each)].
    ^result!

directAt: aKey
        "This is the uncontrolled access"
    ^self directAt: aKey
        ifAbsent: [self error: 'a Key (', aKey asString, ') is missing!!']!

directAt: aKey ifAbsent: aBlock
        "This is the uncontrolled access"
```

```
    ^super at: (aKey asLowerCase asSymbol) ifAbsent: aBlock!

directAt: aKey put: aValue
        "This is the uncontrolled store."
    ^super at: (aKey asLowerCase asSymbol) put: aValue!

hasChanged: aField
        "Signal to the user-interface that another field has changed
        too in addition of the one just changed."
    changed addLast: (aField asLowerCase) asSymbol!

includesKey: aKey
        "The redefinition is necessary for the case-insensitivity."
    ^super includesKey: aKey asLowerCase asSymbol!

nullChanges
        "Empty the changes-queue."
    (changed = nil)
        ifTrue: [changed := OrderedCollection new]
        ifFalse: [(changed size) timesRepeat: [changed removeFirst]]!

printOn: aStream
        "Returns the contents of a Tuple in a somewhat readable
        form."
    | first |
    first := true.
    self keysDo: [:each |
        first
            ifTrue: [first := false]
            ifFalse: [
                aStream nextPut: $,.
                aStream space].
        aStream nextPutAll: (self at: each) asString].!

setNull
        "Set all of the component values to nil."
    self keysDo: [:field |
        self at: field put: nil]!

species
        "A tuple is really a Dictionary by heart."
    ^Dictionary!

virtual: aKey
        "Tells whether a field is derived from other fields and not present
        as an independent value."
   ^(self class virtualFields) includes: (aKey asLowerCase) asSymbol! !
```

11.2. Relation

```
Collection subclass: #Relation
  instanceVariableNames:
    'cursor columns tables where group having order params pos compiled executed size
                  '
  classVariableNames: ''
  poolDictionaries: ''  !


!Relation class methods !

new
        "Cannot create a relation without fixing the tuple-type and
        the tables."
    self error: 'Use open for creating a Relation!!'!

openIn: aDatabase
        "Opens a new relation in the given database."
    | result |
    result := super new.
    result initialize: aDatabase.
    ^result!

rowClass
```

```
        "From a relation the data is returned as an array by default, since
        the names of the columns would not make sense in many cases."
    ^Array! !


!Relation methods !

at: anIndex
        "This is needed to make a relation look like other indexed collections."
    ^self row: anIndex!

bindTo: aCollection
        "Sets the values of the numbered or named bind-variables
        possibly used in the select-statement. If so, this method is
        to be called prior to any fetch method."
    params := aCollection.
    executed := false.!

close
        "Closes the relation"
    cursor disconnect.
    ^self release!

colLabels
        "Returns an array which contains the labels of the columns
        returned by the query."
    ^cursor header!

description
        "Returns an array containing the descriptions of the columns in the
        relation."
    compiled ifFalse: [
        cursor compile: (self selectStatement).
        compiled := true.
        executed := false].
    ^cursor description!

do: aBlock
        "Evaluates the given block with each tuple in the relation
        as a parameter."
    1 to: (self size) do: [:row |
         aBlock value: (self row: row)].!

first
        "Returns the first row in the relation, if any. Notice that
        since a relation can be composed of many tables it
        has to be ensured that no backwards going is tried."
    ^self row: 1!

getValue
        "Private - the rows in a relation or a table are returned as
        instances of the row-class which is specified for Relation,
        Table and their subclasses."
    | rowClass |
    rowClass := self class rowClass.
    rowClass = Array ifTrue: [
        ^rowClass dbData: cursor value]
    ifFalse: [
        ^rowClass dbData: cursor namedValue]!

groupBy: columnList
        "Sets the grouping criteria in the select statement."
    group := columnList.
    self statementChanged!

having: searchCondition
        "Sets the having-condition in the select statement."
    having := searchCondition.
    self statementChanged!

initialize: toDatabase
        "Private - initializes the instance variables."
    params := nil.   columns := ''. tables := ''.
    where := ''.        group := ''.     having := ''.
    order := ''.
    pos := 0. size := -1.
```

```
    compiled := false.  executed := false.
    cursor := DbCursor new: toDatabase.!

orderBy: columnList
        "Sets the ordering-criteria in the select-statement."
    order := columnList!

row: rowNumber
        "Returns the nth row in the relation, if one exists."
    ((rowNumber <= self size) and: [rowNumber >= 1])
        ifTrue: [
            "compile and execute the current statement, if
            necessary."
            compiled ifFalse: [
                cursor compile: (self selectStatement).
                compiled := true.
                executed := false].
            executed ifFalse: [
                self updateVars.
                cursor execute.
                executed := true.
                cursor toStart.
                pos := 1
            ].
            (rowNumber >= pos)
                ifTrue: [
                    (rowNumber - pos) timesRepeat: [cursor forth]]
                ifFalse: [
                    cursor toStart.
                    (rowNumber - 1) timesRepeat: [cursor forth]].
            pos := rowNumber.
            ^self getValue]
        ifFalse: [^nil]!

row: rowNbr col: colNbr
        "Returns the nth column at the mnth row."
    (self row: rowNbr) isNil
        ifTrue: [^nil]
        ifFalse: [^self getValue at: colNbr]!

rowDescriptor
        "Returns an array of data-item descriptors of the contents."
    ^cursor header!

select: colList from: tableList
        "This is the basic select-from -method."
    columns := colList.
    tables := tableList.
    where := ''.
    group := ''.
    having := ''.
    order := ''.
    self statementChanged!

selectStatement
        "This method returns the default select-statement. It can be
        overridden in subclasses if a limited view is desired."
    ^'SELECT ', columns, ' FROM ', tables, ' ', self terms.!

show
        "Opens a window an show the contents of the receiver in it in form
        of a report."
    | window width |
    compiled := false.
    window := ReportWindow new.
    compiled ifFalse: [
           cursor compile: (self selectStatement).
           compiled := true.
           executed := false].
    width := cursor description inject: 0 into: [:sum :col | 1 + sum + col width].
    window openRows: self size + 3 cols: width - 1 label: tables.
    window contents: self.
    window show!

size
        "Returns the number of tuples in the relation."
    (size > -1)
```

```
        ifFalse: [ "Possible earlier result is no longer valid."
            cursor compile: ('SELECT COUNT (*) FROM ',
                tables, self terms).
            compiled := false.
            self updateVars.
            cursor execute.
            cursor toStart.
            size := cursor value at: 1.
            cursor forth.
            cursor points
                ifTrue: [ "Group-by definition was used. Have to count the lines by
                  hand."
                    size := 1.
                    [cursor points]
                        whileTrue: [
                            size := size + 1.
                            cursor forth]]].
        ^size.!

species
        "A relation is a read-only indexed collection by nature."
    ^IndexedCollection!

statementChanged
        "Private - signal that the statement must be compiled again before further
        processing is done"
    compiled := false.
    size := -1.
    params := nil.!

terms
        "Returns the qualifiers of the select statement."
    | result |
    result := ''.
    (where isEmpty) ifFalse: [result := ' WHERE ', where].
    (group isEmpty) ifFalse: [result := result, ' GROUP BY ', group].
    (having isEmpty) ifFalse: [result := result, ' HAVING ', having].
    (order isEmpty) ifFalse: [result := result, ' ORDER BY ', order].
    ^result!

updateVars
        "Private - Send the bind-variable values to the dbcursor."
    (params species)
        when: Array do: [^cursor setVarsTo: params];
        when: Dictionary do: [^cursor setNamedVarsTo: params];
        when: Association do: [
            (params key species)
                when: Integer do: [
                    ^cursor setVar: params key to: params value];
                when: String do: [
                    ^cursor setNamedVar: params key to: params value]];
        when: UndefinedObject do: [^self].
    "The default is to have just one bind-variable of number 1."
    cursor setVar: 1 to: params.!

where: searchCondition
        "Sets the where-condition in the select statement."
    where := searchCondition.
    self statementChanged! !
```

11.3. Table

```
Relation subclass: #Table
  instanceVariableNames:
    'set values subSelect '
  classVariableNames: ''
  poolDictionaries: ''   !


!Table class methods !

rowClass
```

```
        "Returns the component class of a table. Subclasses usually
        override the default, which is a simple Tuple."
    ^Array! !


!Table methods !

close
        "Release the instance variables."
    ^super close!

commit
        "Closes the current transaction and starts a new one.
        All of the cursor have to be re-established."
    ^cursor commit.!

deleteFrom: table
        "Initializes a delete-statement for later execution."
    tables := table.
    where := ''.
    set := ''.
    values := ''.
    subSelect := ''.
    self statementChanged.!

execute
        "Completes an SQL-statement. Prior to this call, an update or
        insert-call should have to been made."
    | statement |
    (set = '') ifFalse: [
        "An update-statement"
        statement := 'UPDATE ', tables, ' SET ', set.
        (where = '') ifFalse: [statement := statement, ' WHERE ', where]]
    ifTrue: [
        (values = '') ifFalse: [
            "An value-kind insert-statement."
            statement := 'INSERT INTO ', tables, ' VALUES (', values, ')']
        ifTrue: [
            (subSelect = '') ifFalse: [
                "An select-kind insert-statement."
                statement := 'INSERT INTO ', tables, ' SELECT ', subSelect,
                    self terms]
            ifTrue: [
                "A delete-statement."
                statement := 'DELETE FROM ', tables, ' WHERE ', where]]].
    (cursor compile: statement) ifTrue: [
        self updateVars.
        ^cursor execute]
    ifFalse: [^false]!

insertInto: table select: subCols from: subTables
        "Initializes an insert-statement with a sub-select statement."
    tables := table.
    values := ''.
    set := ''.
    subSelect := subCols, ' FROM ' subTables.
    where := ''.
    group := ''.
    having := ''.
    self statementChanged.!

insertInto: table values: valueList
        "Initializes an insert-statement."
    tables := table.
    values := valueList.
    set := ''.
    subSelect := ''.
    where := ''.
    group := ''.
    having := ''.
    self statementChanged.!

rollback
        "Discards the current transaction and starts a new one."
    ^cursor rollback.!
```

```
setValue: aTuple
        "Private - sets the tuple to the bind-variables. This is used
        by update and insert operations."
    cursor setNamedVarsTo: aTuple dbData!

startTuple
        "This method returns one tuple, if any exist from the table. It is
        ment to be called after the opening in interactive applications
        to get a grip from the table and typically overridden in subclasses."
    ^self first!

update: table set: setList
        "Initializes an SQL update-statetement. The statement can be
        later be accurated by the where-method."
    tables := table.
    set := setList.
    subSelect := ''.
    values := ''.
    where := ''.
    self statementChanged.! !
```

11.4. KeyedTable

```
Table subclass: #KeyedTable
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''    !


!KeyedTable class methods !

openIn: aDatabase ofName: tableName
        "Opens a keyed database table for reading and writing. One selection
        is preset for enabling the queries about the size of the table."
    | result |
    result := super openIn: aDatabase.
    result select: '*' from: tableName.
    ^result!

rowClass
        "Returns the component class of a table. Subclasses usually
        override the default, which is a simple Tuple."
    ^Dictionary! !



!KeyedTable methods !

add: aTuple
        "This method inserts a tuple into the table. The keys in the dictionary
        have to match the column names to which the values are assigned."
    cursor compile: (self class insertStatement).
    self setValue: aTuple.
    ^cursor execute.!

at: aKey
        "Returns the tuple with the given key. Uses the selectThisStatement
        -method. The parameter is a dictionary with an appropriate keyfields."
    cursor compile: (self class selectThisStatement).
    ^self getFirst: aKey!

at: aKey put: aTuple
        "Sets the new field values to rows which contain the given key value."
    cursor compile: (self class updateStatement).
    self bindToKey: aKey.
    self setValue: aTuple.
    ^cursor execute.!

bindToKey: aKey
        "Private - The column names in the key-subtuple are preceded by 'k's
        to separate them other fields in the statement."
    aKey keysDo: [:each |
```

```
        cursor setNamedVar: ('k', each) to: (aKey at: each) dbData].!

delete: aKey
        "Removes the row or rows with the given value as key."
    cursor compile: (self class deleteStatement).
    self bindToKey: aKey.
    ^cursor execute.!

getFirst: aKey
        "Private - returns the next tuple to the given key from a set
        defined by previously compiled sql-statement."
    self bindToKey: aKey.
    cursor execute.
    cursor toStart.
    (cursor points)
        ifTrue: [^self getValue]
        ifFalse: [^nil].!

getKey
        "Private - Return the key of the current row. Notice that the
        getValue should be implemented by a subclass for this method
        to make any sense."
    ^(self getValue) dbKey!

includesKey: aKey
        "Tells whether the table contains a row with the given key value."
    cursor compile: (self class selectThisStatement).
    self bindToKey: aKey.
    cursor execute.
    cursor toStart.
    ^cursor points!

inspect
        "Open a dictionary inspector window
         on the receiver."
    DictionaryInspector new openOn: self!

keysDo: aBlock
        "Evaluates the given block with each key of the row of the
        table as a parameter."
    cursor compile: (self selectStatement).
    self updateVars.
    cursor execute.
    cursor toStart.
    [cursor points] whileTrue: [
         aBlock value: (self getKey).
         cursor forth].!

nextTo: aKey
        "Returns the next tuple to the given key, regardless whether such
        a key exists in the database or not."
    cursor compile: (self class selectNextStatement).
    ^self getFirst: aKey!

prevTo: aKey
        "Returns the previous tuple to the given key, regardless whether such
        a key exists in the database or not."
    cursor compile: (self class selectPrevStatement).
    ^self getFirst: aKey!

removeKey: aKey
        "Removes the row or rows with the given value as key. This
        operation is a synonym to delete:, and it's included for compa-
        tibility with other dictionary-types."
    ^self delete: aKey!

species
        "Returns dictionary as species, since that is what a keyed table
        in this model really is."
    ^Dictionary! !
```

11.5. Database

```
Object subclass: #Database
  instanceVariableNames:
    'dbName userName passWord '
  classVariableNames:
    'DLLLibrary '
  poolDictionaries: '' !


!Database class methods !

init
        "Sets ups the connection to the API-DLL. This method has
        to be called once prior to any other database routines"
    | dialog |
    "dialog := DialogBox new fromResFile: 'vpmdbeng.res'."
    dialog := DialogBox new fromModule:
        (DynamicLinkLibrary open: 'vpmdbeng') id: 1.
    dialog showWindow.
    DbCursor setLinkLibrary: (DbEngineDLL open: 'dbengine').
    dialog setItemText: 2 string: DbCursor version.
    dialog close.!

open: ofName userName: userName passWord: passWord
       "Returns a new instance which corresponds an existing
        physical database on the disk or in the LAN."
    | result  |
    result := self new.
    result dbname: ofName userName: userName
        passWord: passWord.
    ^result.!

openAsSysadmin: ofName
       "Returns a new instance which corresponds an existing
        physical database on the disk or in the LAN."
    | result  |
    result := self new.
    result dbName: ofName userName: 'SYSADM'
        passWord: 'SYSADM'.
    ^result.!

version
    ^'%FILE%, the Database class
     version %VERSION% at %DATE_OF_UPDATE% by %USER%'! !



!Database methods !

close
        "Erases the database connection."
    self release!

dbName
        "Returns the name of the database"
    ^dbName.!

dbName: toThis userName: uName passWord: pWord
        "Private - Attaches the instance of a database to
        the given name, username and password. Set the isolation mode
        as the lock-nothing by default."
    | testConnection |
    dbName := toThis.
    userName := uName.
    passWord := pWord.
    testConnection := DbCursor new: self.
    testConnection setLockNothingMode.
    testConnection disconnect.
    ^self.!

doSQL: aStatement
        "Compile, execute and commit the given SQL-statement."
    | conn |
    conn := DbCursor new connectTo: self.
```

```
    conn doSQL: aStatement.
    conn disconnect.
    ^true!

passWord
        "Returns the password of the opener."
    ^passWord.!

userName
        "Returns the name of the opener."
    ^userName.! !
```

11.6. DbCursor

```
Object subclass: #DbCursor
  instanceVariableNames:
    'lowHandle '
  classVariableNames:
    'DbLib '
  poolDictionaries:
    'DbEngineConstants ' !


!DbCursor class methods !

new: toDatabase
        "Returns a new connection to the given database."
    ^self new connectTo: toDatabase!

setLinkLibrary: toThis
        "Private: sets the cursor's classes DLL-connection for
        subsequent calls. This is called from Database init."
    DbLib := toThis!

version
        "Returns the class's own version string accompanied
        with the underlying DLL's version string. The method
        setLinkLibrary should be called prior to this one."
    ^DbLib getString: (DbLib version)! !



!DbCursor methods !

back
        "Takes the cursor one step backwards in the result set."
    DbLib back: lowHandle.
    self checkStatus!

checkStatus
        "Tests whether any error took place in the previous call."
     ((DbLib connectionStatus: lowHandle) = 0)
        ifFalse: [
            ^self userError: 'SQL ERROR' what: (
                DbLib getString: (DbLib errorDescription: lowHandle))]
        ifTrue: [^true]!

colCharValue: nbrofCol
        "Private - Returns the value of a column as a character."
    ^Character value: (
        DbLib getColChar: lowHandle ofNbr: nbrofCol)!

colClass: colNbr
        "Private - Returns the native class of the specified column in a
        query."
    (DbLib colDataType: lowHandle ofColumn: colNbr)
        when: DbeCharacter do: [^Character];
        when: DbeString do: [^String];
        when: DbeShortInteger do: [^SmallInteger];
        when: DbeLongInteger do: [^Integer];
        when: DbeVeryLongInteger do: [^Integer];
        when: DbeReal do: [^Float];
```

```
            when: DbeDate do: [^Date];
            when: DbeTime do: [^Time];
            when: DbeTimePoint do: [^String].!

colIntValue: nbrofCol
            "Private - Returns the value of a column as a small integer."
        ^DbLib getColInt: lowHandle ofNbr: nbrofCol!

colLongValue: nbrofCol
            "Private - Returns the value of a column as a long integer."
        ^DbLib getColLong: lowHandle ofNbr: nbrofCol!

colName: colNbr
            "Returns the name of a column in a query."
        ^DbLib getString: (
            DbLib colName: lowHandle ofColumn: colNbr).!

colStrValue: nbrofCol
            "Private - Returns the value of a column as a String."
        ^DbLib getString: (DbLib getCol: lowHandle ofNbr: nbrofCol)!

colValue: colNbr
            "Returns the value of the given column in a query
            as an instance of its native class"
        | dataType |
        dataType := DbLib colDataType: lowHandle ofColumn: colNbr.
        (DbLib isNull: lowHandle column: colNbr)
            ifTrue: [
                dataType = DbeString ifTrue: [^String new:0]
                    ifFalse: [^nil]]
        ifFalse: [
            dataType
            when: DbeCharacter do: [
                ^Character dbData: (self colCharValue: colNbr)];
            when: DbeString do: [
                ^self colStrValue: colNbr];
            when: DbeShortInteger do: [
                ^Integer dbData: (self colIntValue: colNbr)];
            when: DbeLongInteger do: [
                ^Integer dbData: (self colLongValue: colNbr)];
            when: DbeVeryLongInteger do: [
                ^(self colStrValue: colNbr) asInteger];
            when: DbeReal do: [
                ^Fixed dbData: (self colStrValue: colNbr)];
            when: DbeDate do: [
                ^Date dbData: (self colStrValue: colNbr)];
            when: DbeTime do: [
                ^Time dbData: (self colStrValue: colNbr)];
            when: DbeTimePoint do: [
                "No class for timepoint yet!!"
                ^self colStrValue: colNbr]].

        "A new exhange format has been introduced?"
        self userError: 'Don''t know db-exchange method nr ',
            (DbLib colDataType: lowHandle ofColumn: colNbr) printString.!

colWidth: colNbr
            "Returns the width of the data as a string in the column."
        ^DbLib colDataLen: lowHandle ofColumn: colNbr.!

commit
            "Commits all changes pending in the database."
        DbLib commit: lowHandle.
        ^self checkStatus!

compile: aStatement
            "Compiles an SQL-statement. (compStore may be used
            for efficiency.)"
        | timer |
        "Transcript show: 'SQL Compile: ', aStatement.
        Transcript cr."
        timer := Timer start.
        DbLib compStore: lowHandle
                theStmnt: aStatement ofLength: (aStatement size).
        "DbLib compile: lowHandle
                theStmnt: aStatement ofLength: (aStatement size)."
        timer show: 'dbCursor compile'.
```

```
    ^self checkStatus!

compStore: aStatement
        "Compiles an SQL-statement at the first call, restores it otherwise."

    DbLib compStore: lowHandle
            theStmnt: aStatement ofLength: (aStatement size).
    self checkStatus!

connectTo: aDatabase
        "Private - sets up a connection to an open database."
    | dbName userName passWord |
    dbName := aDatabase dbName.
    userName := aDatabase userName.
    passWord := aDatabase passWord.
    self setHandle: (
        DbLib connect: dbName nLen: (dbName size)
        userId: userName uLen: (userName size)
        pw: passWord pLen: (passWord size)).!

description
        "Returns a description of the data behind the cursor."
    | result nbrofCols |
    nbrofCols := DbLib nbrofCols: lowHandle.
    result := Array new: nbrofCols.
    1 to: nbrofCols do: [:col |
        result at: col put: (
            DbDataDescriptor name: (self colName: col)
                width: (self colWidth: col)
                class: (self colClass: col))].
    ^result!

disconnect
        "Disable the connection to the database."
    ((DbLib disconnect: lowHandle) = 0)
        ifFalse: [
            ^self userError: 'SQL ERROR' what: 'Invalid cursor handle.']
        ifTrue: [^true]!

doSQL: aStatement
        "Compiles, executes and commits a single SQL statement."
    DbLib doSQL: lowHandle theStmnt: aStatement
            ofLength: (aStatement size).
    self checkStatus!

duplicate
        "Return a fresh connection to the same database as
        the receiver with same userId and password"
    | result |
    result := DbCursor new.
    result setHandle: (DbLib duplicate: lowHandle).
    ^result.!

execute
        "Executes a compiled SQL-statement."
    | timer |
    timer := Timer start.
    DbLib execute: lowHandle.
    timer show: 'dbCursor execute'.
    ^self checkStatus!

forth
        "Takes the cursor one step further in the result set."
    DbLib forth: lowHandle.
    self checkStatus!

handle
        "Private: returns the low-level handle set by setHandle"
    ^lowHandle!

header
        "Returns an array which contains the names of the columns
        in a select-statement."
    | result nbrofCols|
    nbrofCols := DbLib nbrofCols: lowHandle.
    result := Array new: nbrofCols.
    1 to: nbrofCols do: [:col |
```

```
            result at: col put: (self colName: col)].
        ^result!

    name
            "Answers the possibly given name of the cursor."
        ^DbLib getString: (DbLib name: lowHandle)!

    name: aString
            "Gives a name to a compiled cursor by which it can
            be referred from other cursor or stored to the database."
        DbLib checkRetStatus: (
            DbLib setName: lowHandle toThis: aString ofLength: (aString size)).
            self checkStatus!

    namedValue
            "Returns a dictionary which contains the names of
            the columns and their values as instances of
            their native classes."
        | result |
        result := Dictionary new.
        1 to: (self nbrofColumns) do: [:col |
            result at: (self colName: col) asLowerCase
                put: (self colValue: col)].
        ^result!

    nbrofColumns
            "Returns the number of columns in a query"
        ^DbLib nbrofCols: lowHandle!

    nbrofRows: aTableName
            "Returns the number of rows in a physical table."
        ^DbLib nbrofRows: lowHandle ofTable: aTableName nameLen: (aTableName size)!

    points
            "Returns true if the cursor points to some row in the table."
        ^DbLib points: lowHandle!

    rollback
            "Discards all pending changes in the database"
        DbLib rollback: lowHandle.
        ^self checkStatus!

    setHandle: anInteger
            "Private: checks and sets the low-level handle returned by
             some of dbEngine's constructors."
        lowHandle := anInteger.
        self checkStatus!

    setLockAllMode
            "Tells to the database to lock all of the accessed pages."
        DbLib setIsolationLevel: lowHandle toThis: DbeLockAll.
        self checkStatus!

    setLockCurrentMode
            "Tells to the database to lock the lates accessed page."
        DbLib setIsolationLevel: lowHandle toThis: DbeLockCurrent.
        self checkStatus!

    setLockNothingMode
            "Tells to the database to lock the lates accessed page."
        DbLib setIsolationLevel: lowHandle toThis: DbeLockNothing.
        self checkStatus!

    setNamedVar: aName to: anObject
            "Sets a named bind variable to the given object."
        self setVar: (
            DbLib bindNumber: lowHandle ofName: aName
                ofLen: (aName size))
            to: anObject.!

    setNamedVarsTo: aDictionary
            "Sets a whole set of bind-variables according to the contents of the
            given dictionary."
        aDictionary keysDo: [:each |
            self  setNamedVar: each to: (aDictionary at: each)]!

    setReadOnlyMode
```

```
        "Tells to the database not to lock the accessed pages. Writing
        is not allowed in this mode."
    DbLib setIsolationLevel: lowHandle toThis: DbeReadOnly.
    self checkStatus!

setVar: ofNbr to: anObject
        "Sets a bind-variable identified by its number to contain
        the value of the object. The method chooses an optimal
        method for passing the value."
    | storeAs |

    storeAs := anObject dbData.
    (storeAs species)

       when: String do: [
            DbLib setVar: lowHandle ofNbr: ofNbr
                  toThis: storeAs ofLen: (storeAs size).
            ^self checkStatus];

        when: Character do: [
            DbLib setVarChar: lowHandle ofNbr: ofNbr
                  toThis: (storeAs asciiValue).
            ^self checkStatus];

        when: SmallInteger do: [
            DbLib setVarInt: lowHandle ofNbr: ofNbr
                  toThis: storeAs.
            ^self checkStatus];

        when: LargePositiveInteger do: [
            (storeAs <= 2147483648)
                ifTrue: [
                    DbLib setVarLong: lowHandle ofNbr: ofNbr
                          toThis: storeAs]
                ifFalse: [
                    DbLib setVar: lowHandle ofNbr: ofNbr
                        toThis: (storeAs printString)].
            ^self checkStatus];

        when: LargeNegativeInteger do: [
            (storeAs >= -2147483648)
                ifTrue: [
                    DbLib setVarLong: lowHandle ofNbr: ofNbr
                          toThis: storeAs]
                ifFalse: [
                    DbLib setVar: lowHandle ofNbr: ofNbr
                        toThis: (storeAs printString)].
            ^self checkStatus].

    "No other possible exhange forms."
    self userError: 'INTERNAL ERROR' what:
        'Don''t know how to store an instance of class ',
        storeAs class asString, ' to database.'!

setVarsTo: aCollection
        "Sets all the bind-variables in a statement to the values
        in the given collection by combining the bind-number to
        the position in the collection."
    1 to: (aCollection size) do: [:loop |
         self setVar: loop to: (aCollection at: loop)]!

strValue
        "Returns an array which contains the values as strings
        of the columns in a select-statement."
    | result nbrofCols|
    nbrofCols := DbLib nbrofCols: lowHandle.
    result := Array new: nbrofCols.
    1 to: nbrofCols do: [:col |
        result at: col put: (self colStrValue: col)].
    ^result!

toEnd
        "Takes the cursor one to the last row in the result set."
    DbLib toEnd: lowHandle.
    self checkStatus!

toStart
```

```
        "Makes a cursor point to the first row in the result set."
    DbLib toStart: lowHandle.
    self checkStatus!

value
        "Returns an array which contains the values  of the columns
        of the current row as instances of their native classes."
    | result nbrofCols|
    nbrofCols := DbLib nbrofCols: lowHandle.
    result := Array new: nbrofCols.
    1 to: nbrofCols do: [:col |
        result at: col put: (self colValue: col)].
    ^result! !
```

11.7. DbEngineDLL

```
DynamicLinkLibrary variableByteSubclass: #DbEngineDLL
  classVariableNames: ''
  poolDictionaries:
    'DbEngineConstants PMConstants '   !


!DbEngineDLL class methods !

version
    ^'%FILE%, the dbEngine -database server API
     version %VERSION% at %DATE_OF_UPDATE% by %USER%'! !



!DbEngineDLL methods !

bindNumber: handle ofName: aString ofLen: anInt
        "Private - returns a virtual bind-variable number of a named variable."
    <api: DBEBINDNUMBER short struct ushort short>
    ^self invalidArgument!

checkRetStatus: theCode
        "This method checks the returned status code from an
        impulse and puts an error message plus starts a trace-
        back if it was non-zero"
    (theCode = DbeOK)
        ifFalse: [self handleError: theCode]!

colDataLen: aHandle ofColumn: anInt
        "Returns the space which the value from the column may maximally
        need in the string-representation."
    <api: DBECOLDATALEN short short short>
    ^self invalidArgument.!

colDataType: aHandle ofColumn: anInt
        "Returns the type of the data in the column. The coding is
        defined in the dictionary DbEngineConstants."
    <api: DBECOLDATATYPE short short short>
    ^self invalidArgument.!

colName: aHandle ofColumn: anInt
        "Returns a pointer to an external string which contains
        the name by which the column is identified in the query.
        May be also an arithmetic expression."
    <api: DBECOLNAME short short handle>
    ^self invalidArgument.!

commit: aHandle
        "Commits the potential changes to the database."
    <api: DBECOMMIT short long>
    ^self invalidArgument.!

compile: aHandle theStmnt: aString ofLength: anInt
        "Compiles the given SQL statement."
    <api: DBECOMPILE short struct ushort long>
    ^self invalidArgument.!
```

```
compStore: aHandle theStmnt: aString ofLength: anInt
        "Compiles the given SQL statement when called the first time,
         restores it otherwise."
    <api: DBECOMPSTORE short struct ushort long>
    ^self invalidArgument.!

connect: dbName nLen: anInt1 userId: aString1 uLen: anInt2 pw: aString2 pLen: anInt3
        "Creates a connection to a database and returns a handle to
        access the connection. If the connection didn't succeed handle 255
        is returned."
    <api: DBECONNECT struct ushort struct ushort struct ushort short>
    ^self invalidArgument.!

connectionStatus: aHandle
        "Returns the status code of the given cursor."
    <api: DBECONNECTIONSTATUS short long>
    ^self invalidArgument!

copyString: to from: extString ofBytes: anInt
        "Copies the given number of bytes from an extrnal string to the
        data-area of the first (variable byteclass) argument. It has to
        be created prior to this call."
    <api: COPYSTRING struct handle ushort none>
    ^self invalidArgument.!

disconnect: aHandle
        "Disconnects from the database. Automatic rollback is performed
        and dynamic reservations are released."
    <api: DBEDISCONNECT short long>
    ^self invalidArgument.!

doSQL: aHandle theStmnt: aString ofLength: anInt
        "Compiles, executes and commits the given SQL-statement."
    <api: DBEDOSQL short struct ushort long>
    ^self invalidArgument.!

duplicate: aHandle
        "Creates a new connection to the same database to which the given
        handle is connected; with the same user-id and password."
    <api: DBEDUPLICATE short short>
    ^self invalidArgument.!

errorDescription: aHandle
        "Returns a pointer to an external string which contains
        a description of the status of the handle."
    <api: DBEERRORDESCRIPTION short handle>
    ^self invalidArgument.!

execute: aHandle
        "Executes the previously compiled, stored or restored statement.
        The bind-variables have to be set prior to this call."
    <api: DBEEXECUTE short long>
    ^self invalidArgument.!

forth: aHandle
        "Moves the cursor in the result set one step further.
        Returns error code 1 if no more rows were found."
    <api: DBEFORTH short long>
    ^self invalidArgument.!

getCol: aHandle ofNbr: anInt
        "Returns a pointer to an external string which contains
        the value of a column of any type in textual form."
    <api: DBEGETCOL short short handle>
    ^self invalidArgument.!

getColChar: aHandle ofNbr: anInt
        "Returns the value of a character-type-column."
    <api: DBEGETCOLCHAR short short short>
    ^self invalidArgument.!

getColInt: aHandle ofNbr: anInt
        "Returns the value of a Intacter-type-column."
    <api: DBEGETCOLINT short short short>
    ^self invalidArgument.!

getColLong: aHandle ofNbr: anInt
```

```
        "Returns the value of a Longacter-type-column."
    <api: DBEGETCOLLONG short short long>
    ^self invalidArgument.!

getString: extString
        "Returns a Smalltalk-string which includes the external
        string pointed by the argument."
    | temp |
    temp := String new: (self stringLength: extString).
    self copyString: temp from: extString ofBytes: (temp size).
    ^String dbData: temp!

handleError: theCode
        "Get the error description and display it on the screen."
    | description |
    description := self getString: (self errorDescription: theCode).
    self userError: 'SQL ERROR' what: description.!

internalStatus
        "Returns the status code of the last called operation. This method
        should be used only with the constructors and possibly some
        attributes."
    <api: DBEINTERNALSTATUS long>
    ^self invalidArgument.!

isCompiled: aHandle
        "Returns TRUE if the connection contains a compiled statement
        which can be executed."
    <api: DBEISCOMPILED short boolean>
    ^self invalidArgument.!

isConnected: aHandle
        "Answers TRUE if the argument is a valid connection to a database"
    <api: DBEISCONNECTED short boolean>
    ^self invalidArgument.!

isNull: aHandle column: anInt
        "Returns the fetch-status of a column. May be called after
        a fetch-call has been made."
    <api: DBECOLISNULL short short boolean>
    ^self invalidArgument.!

name: aHandle
        "Returns a pointer to an external string, which contains
        the current name of the cursor."
    <api: DBENAME short handle>
    ^self invalidArgument.!

nbrofCols: aHandle
        "Returns the number of columns in a query result. Can be
        called only if isCompiled returns TRUE."
    <api: DBENBROFCOLS short short>
    ^self invalidArgument.!

nbrofRows: aHandle ofTable: aString nameLen: anInt
        "Returns the number of rows in a physical table."
    <api: DBETABLEROWS short struct ushort long>
    ^self invalidArgument!

points: aHandle
        "Returns TRUE if the cursor points currently to a row in
        the result set."
    <api: DBEPOINTS short boolean>
    ^self invalidArgument.!

restoreStatement: aHandle
        "Retrieves a pre-stored statement according to the current name
        of the cursor."
    <api: DBERESTORESTMNT short long>
    ^self invalidArgument.!

rollback: aHandle
        "Discards the potential changes to the database."
    <api: DBEROLLBACK short long>
    ^self invalidArgument.!

setIsolationLevel: aHandle toThis: lockingMode
```

```
            "The persistence of shared locks in reading is regulated
            by this method. The default value for locking is read
            repeatability which means that all read pages are protected
            against changes. Since this provides poor performance
            other levels are recommended for usage."
        <api: DBESETISOLATIONLEVEL short short long>!

    setName: aHandle toThis: aString ofLength: anInt
            "Gives the cursor an unique name, by which it can be stored to the
            database for later retrieval or referenced by other cursors."
        <api: DBESETNAME short struct ushort long>
        ^self invalidArgument.!

    setVar: aHandle ofNbr: anInt1 toThis: aString ofLen: anInt2
            "Sets a numbered bind-variable of any type to the given value,
            which is expressed as a string."
        <api: DBESETVAR short ushort struct ushort long>
        ^self invalidArgument.!

    setVarChar: aHandle ofNbr: anInt toThis: aChar
            "Sets a character-type bind variable"
        <api: DBESETVARCHAR short ushort ushort long>
        ^self invalidArgument.!

    setVarInt: aHandle ofNbr: anInt toThis: theValue
            "Sets a short-integer-type bind variable"
        <api: DBESETVARINT short ushort short long>
        ^self invalidArgument.!

    setVarLong: aHandle ofNbr: anInt toThis: aLong
            "Sets a Long-integer-type bind variable"
        <api: DBESETVARLONG short ushort long long>
        ^self invalidArgument.!

    storeStatement: aHandle theStatement: aString ofLength: anInt
            "Compiles and stores a statement to the database for later usage.
            The statement can also be executed right after this call."
        <api: DBESTORESTMNT short struct ushort long>
        ^self invalidArgument.!

    stringLength: ofString
            "Returns the length of the external string pointed by the argument."
        <api: STRINGLENGTH handle ushort>
        ^self invalidArgument.!

    toStart: aHandle
            "Makes a cursor to point to the first row in the result set.
            Returns error code 1 if no rows were found."
        <api: DBETOSTART short long>
        ^self invalidArgument.!

    version
            "Returns a pointer to an external string which contains
            the underlying DLL's own version string."
        <api: DBEVERSION handle>
        ^self invalidArgument! !
```

11.8. DbDataDescriptor

```
Object subclass: #DbDataDescriptor
  instanceVariableNames:
    'name width class '
  classVariableNames: ''
  poolDictionaries: ''  !


!DbDataDescriptor class methods !

name: aString width: anInt class: aClass
        "Returns a new dbData descriptor with the initial values as contents."
    | result |
    result := self new.
    result name: aString width: anInt class: aClass.
```

```
    ^result! !


!DbDataDescriptor methods !

dataClass
        "Returns the class of the database data-item described by the receiver."
    ^class!

name
        "Returns the name of the database data-item described by the receiver."
    ^name!

name: aString
        "Sets the name in the descriptor"
    name := aString!

name: aString width: anInt class: aClass
        "Sets the values of the instance variables."
    name := aString.
    width := anInt.
    class := aClass.!

width
        "Returns the width of the database data-item described by the receiver."
    ^width!

width: anInt
        "Sets the width in the descriptor"
    width := anInt! !
```

11.9. DbEngineConstants

```
DbEngineConstants := Dictionary new.
DbEngineConstants
add: (Association key: ('DbeLockAll') value: (3));
add: (Association key: ('DbeLockNothing') value: (1));
add: (Association key: ('DbeOK') value: (0));
add: (Association key: ('DbeString') value: (1));
add: (Association key: ('DbeCharacter') value: (0));
add: (Association key: ('DbeTimePoint') value: (8));
add: (Association key: ('DbeReal') value: (5));
add: (Association key: ('DbeDate') value: (6));
add: (Association key: ('DbeReadOnly') value: (0));
add: (Association key: ('DbeTime') value: (7));
add: (Association key: ('DbeNoConnection') value: (255));
add: (Association key: ('DbeLockCurrent') value: (2));
add: (Association key: ('DbeLongInteger') value: (3));
add: (Association key: ('DbeVeryLongInteger') value: (4));
add: (Association key: ('DbeShortInteger') value: (2)).
```

12. APPENDIX D: USER-INTERFACE CLASSES

12.1. DictionaryDialog

```
DialogBox subclass: #DictionaryDialog
  instanceVariableNames:
    'contents changed binding buttonGroups entryFieldChanged mleFieldChanged '
  classVariableNames: ''
  poolDictionaries:
    'PMConstants '  !


!DictionaryDialog class methods ! !



!DictionaryDialog methods !

blankControl: control
        "Set the value of the control as empty."
    (self controlType: control)
        when: #entryField do: [
            self setItemText: control string: ''];
        when: #infoField do: [
            self setItemText: control string: ''];
        when: #multiLineField do: [
            self importToMLE: control string: ''];
        "With a check box the value is a boolean."
        when: #checkBox do: [
            self setButton: control value: false];
        when: #radioButton do: [
            self setButton: control value: false];
        when: #listBox do: [
            self deleteAllItemsInListBox: control.
            self select: 0 inListBox: control].!

checkBox: control
        "Private - a checkBox was selected. Set it on and send the boolean
        value to the contents-dictionary and show the possible side-effects."
    | field |
    field := self itemIds at: control.
    (self queryButton: control)
        ifTrue: [self setButton: control value: false]
        ifFalse: [self setButton: control value: true].
    contents at: field put: (self queryButton: control).
    changed := true.
    self fieldChanged: field.
    self refresh!

controlType: control
        "Private - return the kind of a control: entryField, multiLineField,
                   infoField,
        listBox, checkBox, radioButton, buttonGroup or static (which covers
        other static objects than info-fields and buttonGroups). Notice that
        buttonGroups have a special role in binding together a bunch of
        radio-buttons."

    | bitCode symbolicName |
    bitCode := PMWindowLibrary sendDlgItemMsg: handle
        item: control msg: WmQuerydlgcode mp1: 0 mp2: 0.

    (bitCode contains: DlgcEntryfield)
        ifTrue: [^#entryField].
    (bitCode contains: DlgcMLE)
        ifTrue: [^#multiLineField].

    (bitCode contains: DlgcButton)
        ifTrue: [
            (bitCode contains: DlgcCheckbox)
                ifTrue: [^#checkBox]
                ifFalse: [
```

```
                        (bitCode contains: DlgcPushbutton)
                            ifTrue: [^#pushButton]
                            ifFalse: [^#radioButton]]].

    (bitCode contains: DlgcStatic)
        ifTrue: [
            "The problem of telling infoFields from buttonGroups is resolved
            by the kind of the value they can display."
            symbolicName := (self itemIds at: control) asSymbol.
            (contents includesKey: symbolicName)
                ifTrue: [
                    ((contents at: symbolicName) class = Symbol)
                        ifTrue: [^#buttonGroup]
                        ifFalse: [^#infoField]]
                ifFalse: [^#static]]

        "No other choices left. Actually this can be a combo-box or a list-box."
        ifFalse: [^#listBox].!

entryFieldChanged: control
        "Private - the contents of an entry field were probably changed.
        Send the new value and echo it on the screen with the potential
        side-effects."
    | new field dataClass |
    entryFieldChanged := false.
    field := self itemIds at: control.
    dataClass := (contents at: field) class.
    new :=  dataClass fromString: (self queryItemText: control).
    (new = (contents at: field))
        ifFalse: [
            contents at: field put: new.
            self setItemText: control string: (contents at: field) asString.
            changed := true.
            self fieldChanged: field.
            self refresh].!

fieldChanged: theField
        "This method is ment to be hooked in the subclasses to get
        control over what happens on the screen."
    ^nil!

init: withBinding withContents: aDictionary
        "Initialize the instance variables. The first parameter contains
        a two-way dictionary which connects the components of the contents
        to the dialog controls. The second contains the initial contents to
        start with."
    contents := aDictionary.
    binding := withBinding.
    buttonGroups := Dictionary new.
    entryFieldChanged := false.
    mleFieldChanged := false.
    changed := false.!

initSize: aRectangle
        "Ignore this becouse dialogs cannot be resized."
    ^nil!

itemIds
        "Private - returns the mapping between tuple components and
        dialog controls. Notice that this comes from an instance variable
        rather than from a class variable."
    ^binding!

listBoxEnter: control
        "Private - a new item in a list-box was entered (double-click or enter-key)."
     self listBoxSelect: control!

listBoxSelect: control
        "Private - a new item in a list-box was selected. Send it and refresh
        the screen."
    | dataClass theItem new |
    theItem := self itemIds at: control.
    dataClass := (contents at: theItem) class.
    new := dataClass fromString: (self listBoxValue: control).
    (contents at: theItem) = new
        ifFalse: [
            contents at: theItem put: new.
```

```
            changed := true.
            self fieldChanged: theItem.
            self refresh]!

listBoxValue: control
        "Private - return the string at the selected line of a listbox."
    | length answer |
    answer := String new: 256.
    length := PMWindowLibrary
        sendDlgItemMsg: handle
        item: control
        msg: LmQueryitemtext
        mp1: ((self querySelectionInListBox: control)
            + (answer size bitShift: 16))
        mp2Struct: answer.
    ^answer copyFrom: 1 to: length!

multiLineFieldChanged: control
        "Private - the contents of a multiline edit field were probably
         changed. Send the new value and refresh the screen with
        the potential side-effects."
    | theField new |
    mleFieldChanged := false.
    theField := self itemIds at: control.
    new := self exportFromMLE: control.
    new = (contents at: theField)
        ifFalse: [ "The value has really changed."
             contents at: theField put: new.
             self setControl: control theValue: (contents at: theField).
             changed := true.
             self fieldChanged: theField.
             self refresh]!

radioButton: control
        "Private - a radio-button was pressed. Set it on and set off the
        previously selected radio button in the group."
    | oldChosen groupId|
    groupId := buttonGroups at: control.
    oldChosen := contents at: groupId.
    "Disable the old choice and enable the new one."
    self setButton: (self itemIds at: (oldChosen asString)) value: false.
    self setButton: control value: true.
    "Enable the new choice."
    contents at: groupId put: (self itemIds at: control).
    changed := true.
    self fieldChanged: groupId.
    self refresh!

setButtonGroup: control to: anEnumerated
        "Private - Sets a group of radio-buttons according to the given value,
        which must be of class enumerated or its subclass."
    | member groupId |
    groupId := self itemIds at: control.
    (anEnumerated class domain) do: [:each |
        member := self itemIds at: (each asString).
        (each = anEnumerated)
            ifTrue: [self setButton: member value: true]
            ifFalse: [self setButton: member value: false].
        "Save the groupId for processing input."
        buttonGroups at: member put: groupId]!

setControl: control theValue: anObject
        "Set the contents of a simple control in a dialog to the given object."
    (self controlType: control)
        "With entry-field or info-field the value is any printable object."
        when: #entryField do: [
            self setItemText: control string: (anObject asString)];
        when: #infoField do: [
            self setItemText: control string: (anObject asString)];
        "A multiline-edit cannot contain anything else than a string."
        when: #multiLineField do: [
            self importToMLE: control string: anObject];
        "With a check box the value is a boolean."
        when: #checkBox do: [
            self setButton: control value: anObject];
        "And with a button group it must be an enumerated type."
        when: #buttonGroup do: [
```

```
            self setButtonGroup: control to: anObject].!

setListBoxControl: control to: anObject outOf: anArray
        "Private - Set the contents of a listbox to contain the argument and set the
        cursor to the currently selected value."
    | position |
    self deleteAllItemsInListBox: control.
    anArray do: [ :each |
        self add: (each asString) toListBox: control at: LitEnd].
    position := anArray indexOf: anObject.
    self select: (position - 1) inListBox: control.!

wmControl: mp1Param with: mp2Param
        "Private - Update the buttons and fields according to the
        WM_CONTROL-messages."
    | control |
    control := mp1Param lowHalf.
    (self controlType: control)
        when: #entryField do: [
            (mp1Param highHalf)
                when: EnSetfocus do: [entryFieldChanged := false];
                when: EnChange do:   [entryFieldChanged := true];
                when: EnKillfocus do: [
                    entryFieldChanged ifTrue: [
                        self entryFieldChanged: control]]];

        when: #multiLineField do: [
            (mp1Param highHalf)
                "Keep count whether the field has changed by help of a flag
                to avoid unnecessary data transfer."
                when: MlnSetfocus do: [mleFieldChanged := false];
                when: MlnChange do: [mleFieldChanged := true];
                when: MlnKillfocus do: [
                    mleFieldChanged ifTrue: [
                        self multiLineFieldChanged: control]]];

        when: #radioButton do: [self radioButton: control];
        when: #checkBox    do: [self checkBox: control];

        when: #listBox         do: [
            (mp1Param highHalf)
                "Both list-box and combo-box are handled here."
                when: LnSelect do: [self listBoxSelect: control];
                when: LnEnter  do: [self listBoxEnter: control];
                "when: CbnLbSelect do: [self listBoxSelect: control];"
                when: CbnEnter do: [self listBoxEnter: control]].! !
```

12.2. TupleDialog

```
DictionaryDialog subclass: #TupleDialog
  instanceVariableNames:
    'cancelled refreshOn '
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants PMConstants ' !


!TupleDialog class methods !

itemIds
        "This method returns the part of the item-ids dictionary that
        is common to all tuple dialogs. It's usually enhanced by subclasses."
    | itemIds |
    itemIds := Dictionary new.
    itemIds
        at: 273 put: #ok;
        at: 'ok' put: 273;
        at: 274 put: #cancel;
        at: 'cancel' put: 274.
    ^itemIds! !
```

```
!TupleDialog methods !

blank
        "Set all the fields and controls on the display to blank."
    contents keysDo: [:each |
        self blankField: each]!

blankField: aField
        "Blank the field on the display."
    | control key |
    key := aField asString.
    (self itemIds includesKey: key)
        ifTrue: [
            self blankControl: (self itemIds at: key)].!

cancel
        "Pushbutton cancel was pressed. Return nil."
    cancelled := true.
    self close!

close
        "Return the contents of the dialog after closing"
    cancelled ifTrue: [contents := nil].
    super close.
    ^contents!

contents
        "Returns the value this object has received from the
        screen as a Tuple."
    ^contents!

contents: aTuple
        "Change the tuple in the dialog to a new one."
    changed ifTrue: [
        (MessageBox confirm:
            'There are pending modifications. Should they be discarded?')
                ifFalse: [^false]].
    aTuple isNil
        ifFalse: [
            contents := aTuple.
            self show]
        ifTrue: [self blank]. "Do not discard if nil is tried to set, just blank the
                  view."
    changed := false!

getLayOut
        "Subclass responsibility - this method fetches the layout of
        the dialog from resources."
    ^self implementedBySubclass!

init: withBinding withContents: aDictionary
        "Initialize the instance variables."
    super init: withBinding withContents: aDictionary.
    cancelled := false.
    refreshOn := false.!

ok
        "Pushbutton OK was pressed. Store the entered values
        to the database."
    self close.!

openOn: aTuple
        "Open with the tuple as initial contents.
         Make this a modal dialog and return the entered fields"
    self init: self class itemIds withContents: aTuple.
    self getLayOut.
    self showWindow.
    self show.
    self processInput.
    ^contents!

refresh
        "Private - refresh the possibly changed fields on the dialog.
         Use a flag to prevent recursive updates on screen which
        possibly never are damped."
    refreshOn ifFalse: [
        refreshOn := true.
```

```
        contents changes do: [:each | self showField: each].
        contents nullChanges.
        refreshOn := false]!

show
        "Transfer the values from the fields of the given tuple to the
         controls in the dialog. For an entry-field the tuple has to respond
         a string, for a button a boolean and for a list-box a set of strings
         which contains the possible choices."
    contents keysDo: [:field |
        self showField: field].
    contents nullChanges.!

showField: aField
        "Transfer the data from the contents-tuple to the display."
    | control key |
    key := aField asString.
    (self itemIds includesKey: key)
        ifTrue: [
            control := self itemIds at: key.
            ((self controlType: control) = #listBox)
                ifFalse: [self setControl: control theValue: (contents at: aField)]
                ifTrue: [
                    self setListBoxControl: control to: (contents at: aField)
                        outOf: (contents choicesAt: aField)]].! !
```

12.3. ModelessTupleDialog

```
TupleDialog subclass: #ModelessTupleDialog
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !


!ModelessTupleDialog class methods ! !



!ModelessTupleDialog methods !

cancel
        "Cancel was pressed. Signal the owner about that."
    owner cancelled.!

ok
        "OK was pressed. Signal the owner about that."
    owner accept.
    changed := false!

openOn: aTuple owner: aTableInspector
        "Open with the tuple as initial contents.
         Make this a modeless dialog and show it on the display."
    self init: self class itemIds withContents: aTuple.
    self owner: aTableInspector.
    self getLayOut.
    self showWindow.
    self show.! !
```

12.4. TableDialog

```
TupleDialog subclass: #TableDialog
  instanceVariableNames:
    'table recursOn '
  classVariableNames: ''
  poolDictionaries:
    'PMConstants '  !


!TableDialog class methods !
```

```
itemIds
        "This method returns the part of the item-ids dictionary that
        is common to all table dialogs."
    | itemIds |
    itemIds := super itemIds.
    itemIds
         at: 279 put: #add;
         at: 'add' put: 279;
         at: 278 put: #remove;
         at: 'remove' put: 278;
         at: 277 put: #next;
         at: 'next' put: 277;
         at: 276 put: #previous;
         at: 'previous' put: 276;
         at: 275 put: #exit;
         at: 'exit' put: 275.
   ^itemIds! !


!TableDialog methods !

add
        "Add a new item into the table."
    contents := table class rowClass new.
    self show.
    "Disable those pushbuttons which shouldn't be pressed right now."!

cancel
        "At this level canceling just restores the initial values."
    self contents: (table at: (contents dbKey)).!

close
        "Close the table before exiting."
    table commit.
    table close.
    ^super close!

exit
        "Exiting table dialog is the same as canceling tuple-dialog."
    changed ifTrue: [
        (MessageBox confirm: 'There are unwritten changes. Should they be
                  discarded?')
            ifFalse: [self ok]].
    self close!

fieldChanged: aField
        "Private - This is the random-fetch method. If a key-field is changed, and a
        record containing the new key combination is found, it's set as the
        current contents."
    | temp |
    recursOn ifFalse: [
        recursOn := true.
        (contents class keyFields includes: aField)
            ifTrue: [
                temp := table at: (contents dbKey).
                temp isNil ifFalse: [
                    "Here one could ask whether the old record should replace the new
                  one."
                    contents := temp.
                    self show]].
        recursOn := false]!

next
        "The next-pushbutton was pressed. Fetch the next item from the
        database or display a message that it couldn't be found."
    | newRow |
    newRow := table nextTo: contents dbKey.
    (newRow isNil)
        ifTrue: [
            self holdOn: 'This is the last row in the table']
        ifFalse: [
            self contents: newRow]!

ok
        "In this context ok means that the current values are written to the
        database, added if they are new and updated if they are read from
```

```
        the database."
    (table includesKey: (contents dbKey))
        ifFalse: [
            (table add: contents) ifTrue: [
                self inform: 'The new row has been added to the database.']]
        ifTrue: [
            (table at: (contents dbKey) put: contents) ifTrue: [
                self inform: 'The changes have been written to the database.']].
    changed := false!

openOn: aTable
        "Open a modal dialog with a capability of viewing and editing
        the contents of the given database table."
    | startWith |
    CursorManager execute change.
    table := aTable.
    recursOn := false.
    (table size > 0)
        ifTrue: [startWith := table startTuple]
        ifFalse: [startWith := table class rowClass new].
    CursorManager normal change.
    ^super openOn: startWith!

previous
        "The previous-pushbutton was pressed. Fetch the previous item from the
        database or display a message that it couldn't be found."
    | newRow |
    newRow := table prevTo: contents dbKey.
    (newRow isNil)
        ifTrue: [
            self holdOn: 'This is the first row in the table']
        ifFalse: [
            self contents: newRow]!

remove
        "Remove the current row from the database."
    (MessageBox confirm: 'Is the current row removed from the table?')
        ifTrue: [
            (table removeKey: contents dbKey) ifTrue: [
                self blank.
                self inform: 'The row has been removed from the table. It can be
                  restored by selecting rollback or pressing OK.'.
                changed := false]].! !
```

12.5. TableInspector

```
DictionaryInspector subclass: #TableInspector
  instanceVariableNames:
    'dialog table '
  classVariableNames: ''
  poolDictionaries:
    'PMConstants ' !


!TableInspector class methods ! !



!TableInspector methods !

accept
        "Private - The OK-button was pressed in a modeless child dialog.
        Write the contents of it to the database."
    instIndex isNil ifTrue: [^self holdOn: 'No database row selected.'].
    table at: (instList at: instIndex) key put: dialog contents.!

add
        "Private - add a new row to the table."
    | newVal |
    newVal := self keyDialogClass new openOn: (table class rowClass new).
    newVal isNil ifFalse: [
        (table includesKey: newVal dbKey)
            ifTrue: [
```

```
                ^self holdOn: 'A row with that key is already in the table.']
            ifFalse: [
                table add: newVal.
                self addKey: newVal dbKey.
                dialog contents: newVal]].!

buildMenuBar
        "Private - build the menu bar. Most of the regular text-window choices
        are not present becouse the window does not contain an editable
        text pane."
     menuWindow addMenu: self tableMenu owner: self.!

cancelled
        "The cancel-button was pressed in the child. Restore its initial
        contents."
    dialog contents: (table at: (dialog contents dbKey))!

close
        "This method closes the database connection before closing
        the window."
    table commit.
    table close.
    dialog close.
    ^super close!

commit
        "Commit the changes to the database."
    table commit!

dialog
        ^dialog!

dialogClass
        "Subclass responsibility - This method tells which class is used
        for viewing and editing the rows in the table."
    self implementedBySubclass!

enableInspectItem
        "This method is suppressed."
    ^nil!

keyDialogClass
        "Subclass responsibility - This method tells which class is used
        for prompting a new key value."
    self implementedBySubclass!

openOn: aTable
        "This operation opens a keyed database table inspector on the given
        table. It uses the dialog-classes defined by the respective methods.
        The table should have been opened prior to the call and it may contain
        whatever selection that is in effect."
    dialog := self dialogClass new.
    table := aTable.
    object := table.
    self
        label: 'Inspecting: ', table class printString.
    self addSubpane:
        (ListPane new
            owner: self;
            when: #getContents perform: #instVarList: ;
            when: #select perform: #selectInstance: ;
            when: #doubleClickSelect perform: #inspectSelection: ;
            when: #getMenu perform: #tableMenu: ;
            returnIndex: true;
            framingRatio: (
                0@0 extent: 1/3 @ 1)).
    CursorManager execute change.
    self setInstList.
    super openIn: (30@60 extent: ((self dialogSize x * 3)//2 - 1)
        @ self dialogSize y + 10).
    dialog openOn: Tuple new owner: self.
    dialog blank.
    CursorManager normal change!

remove
        "Remove the current row from the database."
    (MessageBox confirm: 'Is the given row removed from the table?')
```

```
            ifTrue: [
                super remove.
                dialog blank]!

    rollback
            "Discard the changes from the database."
        table rollback!

    selectInstance: aListPane
            "Private - A row was selected in the list pane. Show its
            contents in the dialog."
        instIndex := aListPane selection.
        instIndex isNil
            ifTrue: [dialog blank]
            ifFalse: [dialog contents: (table at: (instList at: instIndex) key)].
        dialog showWindow!

    setInstList
            "Private - Compute instList, an
             OrderedCollection of key strings
             for the list pane."
        | aSet |
        aSet := Set new: table size.
        table keysDo: [:aKey |
            aSet add:
                (Association key: aKey value: aKey printString)].
        instIndex := nil.
        (instList := SortedCollection new)
            sortBlock: [:a :b| a < b];
            addAll: aSet!

    tableMenu
            "Private - Return the menu used for editing the table."
        | menu |
        menu := Menu new.
        menu
            appendItem: '~Add row   Alt+A' selector: #add accelKey: $a accelBits: AfAlt |
                      AfChar ;
            appendItem: '~Remove rowAlt+R' selector: #remove accelKey: $r accelBits:
                      AfAlt | AfChar ;
            appendSeparator;
            appendItem: '~Commit    Alt+C' selector: #commit accelKey: $c accelBits:
                      AfAlt | AfChar ;
            appendItem: '~Undo changes     Alt+U' selector: #rollback accelKey: $u
                      accelBits: AfAlt | AfChar ;
            title: '~Edit'.
        ^menu!

    tableMenu: pane
            "Private - Set the table inspector list pane menu."
        pane setMenu: self tableMenu! !
```

12.6. ReportWindow

```
TextWindow subclass: #ReportWindow
  instanceVariableNames:
    'width height relation colDesc '
  classVariableNames: ''
  poolDictionaries: '' !


!ReportWindow class methods ! !



!ReportWindow methods !

columnTitle: ofNbr to: aString
        "Sets the title of the nth column to the given string."
    (colDesc at: ofNbr) name: aString!

columnWidth: ofNbr to: anInt
        "Sets the title of the nth column to the given string."
```

```
    (colDesc at: ofNbr) width: anInt!

contents: aRelation
        "Prints out the contents of a relation into the window."
    relation := aRelation.
    colDesc := relation description.
    width := colDesc inject: 0 into: [:sum :col | sum := sum + col width].
    height := aRelation size + 2.!

fillContentsFrom: aRelation
        "Private - Fills the contents of the text window by a tabular
        representation of the contents of the relation."
    | buffer rowWidth |
    self fillRow: colDesc with: [:col | (colDesc at: col) name].
    rowWidth := colDesc inject: 0 into: [:sum :col | sum := sum + col width].
    self next: (rowWidth - 1 + colDesc size) put: $-.
    self cr.
    aRelation do: [:row |
        self fillRow: colDesc with:
            [:col | (row at: col) asString]]!

fillRow: ofDesc with: aBlock
        "Private - Adds one row of data to the contents by calling
        the given one-argument block for getting the value of each column."
    | rowData fieldsLeft |
    "First gather the data into an array of strings."
    fieldsLeft := ofDesc size.
    rowData := Array new: fieldsLeft.
    1 to: fieldsLeft do: [:col |
        rowData at: col put: (aBlock value: col)].
    "The print rows until all the data has been printed."
    [fieldsLeft > 0] whileTrue: [
        fieldsLeft := self oneRow: rowData by: ofDesc].
    self cr.!

oneRow: anArray by: rowDesc
        "Private - Prints as much of the strings in the array as fits into the fields
        at one line and removes the printed data from the array. Return the
        number of unfinished columns."
    | first fieldWidth string temp unFinished |
    unFinished := 0.
    first := true.
    1 to: anArray size do: [:col |
        first ifTrue: [first := false] ifFalse: [self nextPut: $ ].
        fieldWidth := (rowDesc at: col) width.
        string := anArray at: col.
        (fieldWidth >= string size)
            ifTrue: [
                "The data fits on this line."
                anArray at: col put: '']
            ifFalse: [
                "Split the data, print the first part and save the rest for the next
                  round."
                temp := string splitOnBlank: fieldWidth.
                string := temp at: 1.
                anArray at: col put: (temp at: 2).
                unFinished := unFinished + 1].
        (rowDesc at: col) dataClass = String
            ifTrue: [self nextPutAll: (string justifiedLeftTo: fieldWidth)]
            ifFalse: [self nextPutAll: (string justifiedRightTo: fieldWidth)]].
    (unFinished > 0) ifTrue: [self cr].
    ^unFinished!

openRows: rowCount cols: colCount label: aString
        "Initializes the window capable of holding the given rows and columns of
        text (if possible) and makes it visible."
    self label: aString.
    file := ''.
    partial := false.
    self addSubpane:
        (TextPane new
            owner: self;
            when: #getContents perform: #stringContents:;
            when: #save perform: #saveString:;
            framingBlock: [ :box | box];
            textFont: Font defaultFixedFont).
    height := Display height min: self pane font height * (rowCount + 5).
```

```
        width := Display width min: self pane font width * (colCount + 4).
        self openIn: (Rectangle origin:
            (Display width - width) // 2 @ (Display height - height) // 2
            extent: width@height).!

show
        "Fetches the contents of the given relation and shows it in the text pane."
        self fillContentsFrom: relation! !
```

13. APPENDIX E: UTILITIES AND ENHANCEMENTS

13.1. TextLine

```
Object subclass: #TextLine
  instanceVariableNames:
    'contents lineNbr fileName '
  classVariableNames: ''
  poolDictionaries: '' !


!TextLine class methods !

file: fileName nbr: anInt contents: aString
    | result |
    result := self new.
    result file: fileName.
    result nbr: anInt.
    result contents: aString.
    ^result! !



!TextLine methods !

contents
    ^contents!

contents: aString
    contents := aString!

cSmaller: thanThis
        "Returns true if the contents are smaller that
        in the argument."
    ^contents < thanThis contents!

file
    ^fileName!

file: aString
    fileName := aString!

lSmaller: thanThis
        "Returns true if the line-number is smaller that
        in the argument."
    ^lineNbr < thanThis nbr!

nbr
    ^lineNbr!

nbr: anInt
    lineNbr := anInt! !
```

13.2. TextFileDifference

```
Object subclass: #TextFileDifference
  instanceVariableNames:
    'diff '
  classVariableNames: ''
  poolDictionaries: ''    !


!TextFileDifference class methods !

compareFile: fileName1 toFile: fileName2
        "This method creates a new instance of the difference
        and makes the comparison between the two files."
```

```
    | result |
    result := self new.
    result compareFile: fileName1 toFile: fileName2.
    ^result! !


!TextFileDifference methods !

compareContents: lines1 with: lines2
        "This method compares two sorted collections of
        lines and adds the differing components into another
        sorted collection, this time by the line number."
    | result end i1 i2 |
    result := SortedCollection sortBlock: [:a :b | a lSmaller: b].
    end := false. i1 := 1. i2 := 1.
    [end] whileFalse: [
        [(i1 < lines1 size) and:
            [(lines1 at: i1) cSmaller: (lines2 at: i2)]]
        whileTrue: [
            result add: (lines1 at: i1).
            i1 := i1 + 1].
        [(i2 < lines2 size) and:
            [(lines2 at: i2) cSmaller: (lines1 at: i1)]]
        whileTrue: [
            result add: (lines2 at: i2).
            i2 := i2 + 1].
        (i1 = lines1 size or: [i2 = lines2 size])
            ifTrue: [end := true].
        ((lines1 at: i1) contents = (lines2 at: i2) contents)
             ifTrue: [
                 i1 := i1 + 1. i2 := i2 + 1]].
    [i1 <= lines1 size] whileTrue: [
        result add: (lines1 at: i1).
        i1 := i1 + 1].
    [i2 <= lines2 size] whileTrue: [
        result add: (lines2 at: i2).
        i2 := i2 + 1].
    ^result!

compareFile: fileName1 toFile: fileName2
        "This method compares the contents of the two text-
        files and puts the differing lines into the diff-ordered
        collection."
    | lines1 lines2 |
    lines1 := self readContents: fileName1.
    lines2 := self readContents: fileName2.
    diff := self compareContents: lines1 with: lines2 .!

printOn: aStream
        "Show the contents of the difference."
    diff do: [:each |
        aStream nextPutAll: each file.
        aStream nextPut: $(.
        aStream nextPutAll: each nbr printString.
        aStream nextPutAll: ') '.
        aStream nextPutAll: each contents.
        aStream cr].!

readContents: fileName
        "This method reads the contents of a text file
        into an ordered collection with the line-number and
        contents of each line as components."
    | iFile result lineNumber |
    lineNumber := 1.
    result := SortedCollection sortBlock: [:a :b |
        a cSmaller: b].
    iFile := File pathNameReadOnly: fileName.
    [iFile atEnd] whileFalse: [
        result add: (TextLine file: fileName nbr: lineNumber
            contents: iFile nextLine).
        lineNumber := lineNumber + 1].
    iFile close.
    ^result! !
```

13.3. Changes to the standard environment

```
!Object class methods !

dbData: anObject
        "Generic - returns a new object according to the data read from
        the database."
    ^self error: 'Must be implemented by subclasses!!'! !


!Object methods !

asString
        ^self printString!

dbData
        "By default, everything is represented in the database as it is
        in memory"
    ^self!

holdOn: aMessage
        "Display a negative or restricting message."
    MessageBox titled: 'HOLD ON' withText: aMessage style: 32.
    ^false!

inform: aString
        "Inform the user about a positive event."
    MessageBox titled: 'OK' withText: aString style: 64.
    ^true!

userError: title what: aString
        "Display an error message."
    MessageBox titled: title withText: aString style: 32.
    ^false!

when: aValue do: theBlock
        "Implements a kind of a case-statement which evaluates
        the given block if the receiver equals to the given value."
    (self = aValue)
        ifTrue: [^theBlock value]! !


!Behavior methods !

fileOutChanges: toAStream
        "Files the class out into a file named after the default name."
    | classReader |
    self unchanged ifTrue: [^nil].
    toAStream cr.
    classReader := ClassReader forClass: self.
    classReader instanceHeaderOn: toAStream.
    self selectors asSortedCollection do: [ :selector |
        (self compiledMethodAt: selector) userDefined
            ifTrue: [
                toAStream
                    cr;
                    nextChunkPut: (self sourceCodeAt: selector)]].
    toAStream nextPutAll: ' !!'; cr.
    classReader release!

totallyNew
            "Answers true if none of the class's methods is from the standard
            image, which can be considered as evidence of that the receiver
            is a user-defined subclass. A class with no selectors or class
            selectors is considered an old one."
        | empty |
        empty := true.
        self selectors do: [:s |
            (self compiledMethodAt: s) userDefined
                ifFalse: [^false]
                ifTrue: [empty := false]].
        self class selectors do: [:s |
```

```
            (self class compiledMethodAt: s) userDefined
                ifFalse: [^false]
                ifTrue: [empty := false]].
        ^empty not!

unchanged
        "Answers true if all of the class's methods are from the standard
        image."
        self selectors asSortedCollection do: [:s |
            (self compiledMethodAt: s) userDefined
                ifTrue: [^false]].
        ^true! !



!Class methods !

fileOut
        "Files the class out into a file named after the default name."

    (ClassReader forClass: self) fileOut: (File
        fileName: self name
        extension: 'cls').!

fileOut: pathName
        "Files the class out into a file named after the default name."
    (ClassReader forClass: self) fileOut: pathName, (File
        fileName: self name
        extension: 'cls').! !



!IndexedCollection class methods !

dbData: aCollection
        "The data comes as dictionaries from the database. This method
        returns it as it is."
    ^aCollection! !



!CompiledMethod methods !

userDefined
        "Tells whether a the receiver is created or has been updated outside
        the original image."
        ^(self sourceIndex) = 1 or: [self sourceIndex = 2]! !



!String class methods !

dbData: aString
        "Just copies the contents."
    ^aString!

fromString: aString
        "Returns the argument becouse it already is a string."
    ^aString! !



!String methods !

asTime
        "Answers a time represented by the receiver."
    ^Time fromString: self!

dbData
        "Strings are stored as they are."
    ^self!

justifiedLeftTo: anInteger
        "Returns a string of the given size with the argument in the left
        side of it."
    | result |
```

```
    (anInteger >= self size)
        ifTrue: [
            result := String new: (anInteger - self size).
            result atAllPut: $ .
            ^self, result]
        ifFalse: [
            ^self copyFrom: 1 to: anInteger]!

justifiedRightTo: anInteger
        "Returns a string of the given size with the argument in the right
        side of it."
    | result |
    (anInteger >= self size)
        ifTrue: [
            result := String new: (anInteger - self size).
            result atAllPut: $ .
            ^result, self]
        ifFalse: [
            ^self copyFrom: (self size) - anInteger + 1 to: (self size)]!

replaceChar: aChar byChar: withChar
        "Returns a copy of the receiver in which all of the occurences
        of the first argument are replaced by the second. By JuM,
        90-05-25."
    ^self collect: [:each |
        (each = aChar)
            ifTrue: [withChar]
            ifFalse: [each]]!

splitOnBlank: anInt
        "Returns an array of two substrings of the receiver so that the splitting
        is done on a blank at pos <= anInt if one exists or at anInt otherwise.
        If the blank is found, it's excluded from both strings."
    | blankPos result |
    result := Array new: 2.
    blankPos := 0.
    1 to: anInt do: [:pos | (self at: pos) = $  ifTrue: [blankPos := pos]].
    blankPos = 0
        ifTrue: [
            result at: 1 put: (self copyFrom: 1 to: anInt).
            result at: 2 put: (self copyFrom: anInt + 1 to: self size)]
        ifFalse: [
            result at: 1 put: (self copyFrom: 1 to: blankPos - 1).
            result at: 2 put: (self copyFrom: blankPos + 1 to: self size)].
    ^result! !



!Symbol class methods !

fromString: aString
        "Returns the argument as a Symbol."
    ^aString asSymbol! !



!Dictionary class methods !

dbData: aDictionary
        "By default the data read from the database goes as it is."
    ^aDictionary! !



!SystemDictionary methods !

loadSources
        "This method goes through a directory which has been
        written by the logAllChanges-method. It reads in all the
        sources in it, resulting an image identical to the one
        the logAllChanges was send to."
    | path |
    CursorManager execute change.
    path := Prompter prompt: 'Directory where to read the sources from?'
        default: Directory current fullDirName.
    path isNil ifFalse: [
        ((path at: path size) = $\) ifTrue: [
```

```
            path := path copyFrom: 1 to: (path size - 1)].

        "First file in all new pool dictionaries."
        ((Directory pathName: path) formatted: '*.dic') do: [ :aClass |
            (File pathNameReadOnly: path, '\', (aClass at: 1)) fileIn].

        "Then file in all the new classes and changes to
        the standard image."
        (File pathNameReadOnly: path, '\newclass.sml') fileIn.
        (File pathNameReadOnly: path, '\changes.sml') fileIn.

        "Finally file in the code for the new classes."
        ((Directory pathName: path) formatted: '*.cls') do: [ :aClass |
            (File pathNameReadOnly: path, '\', (aClass at: 1)) fileIn]].
    CursorManager normal change.!

logAllChanges
        "Write all of the modifications done to the standard image into
        one file and the new classes into files of their own."
    | classes changes fpath dialog |
    "dialog := DialogBox new fromResFile: 'vpmcustl.res'."
    fpath := Prompter prompt: 'The directory where to put the changes?'
        default: Directory current fullDirName.
    ((fpath isNil) or: [fpath size = 0]) ifTrue: [^nil].
    dialog := DialogBox new fromModule:
        (DynamicLinkLibrary open: 'vpmdbeng') id: 2.
    dialog showWindow.
    ((fpath at: fpath size) = $\) ifFalse: [
        fpath := fpath, '\'].
    classes := File newFile: fpath, 'newclass.sml'.
    changes := File newFile: fpath, 'changes.sml'.
    Object withAllSubclasses do: [ :class |
        class totallyNew
            ifTrue: [
                class fileOutOn: classes.
                classes nextPutAll: '!!'; cr; cr.
                class fileOut: fpath]
            ifFalse: [
                 class class fileOutChanges: changes.
                 class fileOutChanges: changes]].
    changes close.
    "classes nextPutAll: ' !!'; cr."
    classes close.
    dialog close! !


!Font class methods !

defaultFixedFont
        "Answer some good fixed-space font for column-sensitive
        purposes."
    ^Font face: 'System Monospaced' size: 8@12
         fixedWidth: true attributes: 0! !


!Magnitude class methods !

dbData: anObject
        "Objects of kind magnitude can be already converted in
        the internal representation to gain maximal efficiency."
    (anObject class)
        when: self do: [^anObject];
        when: String do: [^self fromString: anObject].
    ^anObject! !


!Magnitude methods !

dbData
        "The primitive classes are represented in the database typically
        as they are."
    ^self! !
```

```
!Character class methods !

fromString: aString
        "Converts a string into a character, if possible."
    (aString size > 0)
        ifTrue: [^aString at: 1]
        ifFalse: [^nil]! !



!Date class methods !

dbData: anObject
        "Dates come from the database as ISO strings or
        date instances."
    (anObject class)
        when: self do: [^anObject];
        when: String do: [^self fromISOString: anObject].!

fromISOString: aString
        "Answer a date represented in the receiver in the ISO-
        format YY-MM-DD. Edited by JuM 1990-05-24."

    |year month day temp|

    temp  := (aString replaceChar: $- byChar: $ ) asArrayOfSubstrings.
    year  := (temp at:1) asInteger.
    month := (temp at:2) asInteger.
    day   := (temp at:3) asInteger.
    ^self newDay: day
        month: (self nameOfMonth: month)
        year: year!

fromOwnString: aString
        "Answer a Date specified by aString.  aString
         contains first the day number then the month
         name and then the year separated with blanks."
    | aStream day month year |
    aStream := ReadStream on: aString.
    [aStream peekFor:  $ ] whileTrue: [].
    day :=  aStream upTo: $ .
    [aStream peekFor: $ ] whileTrue: [].
    month := (aStream upTo: $ ) asLowerCase.
    [aStream peekFor: $ ] whileTrue: [].
    year := aStream upTo: $ .
    month := MonthStrings at: month
        ifAbsent: [self errorInMonth].
    ^self
        newDay: day asInteger
        month: month
        year: year asInteger!

fromString: aString

        "Answer a Date representing the date described by the receiver.
         The following patterns are accepted: 'YY-MM-DD' and 'DD Mon YY'.
         Modified by JuM, 90-05-24"

    | pattern |
    pattern := Pattern new: '*-*-*'.
    (pattern match: aString index:1) notNil
        ifTrue:   [^self fromISOString: aString]
        ifFalse: [^self fromOwnString: aString]! !



!Date methods !

asISOstring
        "Answer the receiver as a string of format YYYY-MM-DD."
    ^(self year printString), '-',
        (self monthIndex printString), '-',
        (self dayOfMonth printString)!

dbData
        "The dates are stored to the database as ISO strings."
```

```
    ^self asISOstring!

printOn: aStream
        "Append the ASCII representation of the receiver
         to aStream  in the form: dd mmm  yyyy. (The form
         yyyy is satisfied only for positive year numbers
         of 4 digits). Modified by JuM 90-06-15."
    | dayNumber nameOfMonth |
    dayNumber := self dayOfMonth.
    dayNumber printOn: aStream.
    aStream space.
    nameOfMonth := #('Jan' 'Feb' 'Mar' 'Apr' 'May'
        'Jun' 'Jul' 'Aug' 'Sep' 'Oct' 'Nov' 'Dec').
    aStream
        nextPutAll: (nameOfMonth at: self monthIndex);
        space.
    self year printOn: aStream! !


!Float class methods !

dbData: aNumber
        "Real numbers come as fixed point units from the database."
    aNumber class
        when: String do: [^self fromString: aNumber];
        when: SmallInteger do: [^self fromInteger: aNumber];
        when: LargePositiveInteger do: [^self fromInteger: aNumber];
        when: LargeNegativeInteger do: [^self fromInteger: aNumber].
    ^self new: aNumber! !


!Float methods !

* aNumber
        "Answer the result of multiplying
         the receiver by aNumber."
    <primitive: 49>
    aNumber species == Float
        ifTrue: [self class noMathChip: [:result |
            FloatLibrary multiply: self by: aNumber result: result.
            ^result]]
        ifFalse: [^self * aNumber asFloat]!

+ aNumber
        "Answer sum of the receiver and aNumber."
    <primitive: 41>
    aNumber species == Float
        ifTrue: [self class noMathChip: [:result |
            FloatLibrary add: self to: aNumber result: result.
            ^result]]
        ifFalse: [^self + aNumber asFloat]!

- aNumber
        "Answer the difference between
         the receiver and aNumber."
    <primitive: 42>
    aNumber species == Float
        ifTrue: [self class noMathChip: [:result |
            FloatLibrary subtract: aNumber from: self result: result.
            ^result]]
        ifFalse: [^self - aNumber asFloat]!

/ aNumber
        "Answer the result of dividing
         the receiver by aNumber."
    <primitive: 50>
    aNumber species == Float
        ifTrue: [self class noMathChip: [:result |
            FloatLibrary divide: self by: aNumber result: result.
            ^result]]
        ifFalse: [^self / aNumber asFloat]!

< aNumber
        "Answer true if the receiver is less
         than aNumber, else answer false."
```

```
    <primitive: 43>
    aNumber species == Float
        ifTrue: [self class noMathChip: [:result |
            ^FloatLibrary less: self than: aNumber]]
        ifFalse: [^self < aNumber asFloat]!

= aNumber
        "Answer true if the receiver is equal
         to aNumber, else answer false."
    <primitive: 47>
    aNumber species == Float
        ifTrue: [self class noMathChip: [:result |
            ^FloatLibrary equal: self to: aNumber]]
        ifFalse: [
            (aNumber isKindOf: Number)
                ifTrue: [^self = aNumber asFloat]
                ifFalse: [^false]]! !


!Fraction class methods !

dbData: aFixed
        "A fixed can be converted into a fraction quite easily."
    ^(self numerator: (aFixed * 100) truncated denominator: 100) * 1! !


!Integer class methods !

dbData: anObject
        "The return of the value is done in a special way to allow the growing
        of the receiver."
    (anObject class = String)
        ifTrue: [^self fromString: anObject]
        ifFalse: [^anObject]!

fromString: aString
        "Makes a new integer according to its string representation."
    | result |
    (aString size = 0) ifTrue: [^0].
    result := 0.
    aString do: [:each |
        (each isDigit) ifTrue: [
            result := result * 10 + each digitValue]
        ifFalse: [
            self userError: 'ARITHMETICS ERROR'
                what: 'invalid characters in image ', aString]].
    ^result! !


!Integer methods !

contains: anInteger
        "Answer true if the receiver has at least those bits
        set as the argument."
    ^(self bitAnd: anInteger) = anInteger!

printOn: aStream
        "Append the ASCII representation (radix 10)
         of the receiver to aStream."
    self printOn: aStream base: 10! !


!Time class methods !

dbData: aString
        "Time values are exhanged in the string format."
    ^self fromString: aString!

fromString: aString
        "Returns a time according to the given string."
    | help seconds |
    help := (aString replaceChar: $: byChar: $ ) asArrayOfSubstrings.
    seconds := help inject: 0 into: [:result :figure |
```

```
        result := result * 60 + (figure asInteger)].
    (3 - help size) timesRepeat:
        [seconds := seconds * 60].
    ^self fromSeconds: seconds! !


!Time methods !

dbData
        "Time-values are stored into database as strings."
    ^self printString! !


!NotificationManager methods !

startUp
        "Private - Start a Smalltalk/V session."
    | font previousScreenSize |
    Font initialize.
    font := PM memPS currentFont.
    Font setSysFont: font.
    CursorManager initialize.
    Bitmap initialize.
    previousScreenSize := Display extent.
    Screen initialize.
    FileDialog initialize.
    Database init.
    Smalltalk isRunTime
        ifTrue: [self startUpApplication]
        ifFalse: [
            previousScreenSize = Display extent
                ifFalse: [
                    self initialize.
                    TextFont := ListFont := font.
                    TextWindow initializeTranscript]
                ifTrue: [self resume]].!

startUpApplication
        "Place startup code for your application here in place
         of 'TextWindow initializeTranscript'.
         For example,
            self initialize.
            Puzzle15 new open."
    WindowLabelPrefix := 'WorktimeManager:'.
    self initialize.    "Destroy all windows that were saved in the image."
    WorktimeManager new open! !


!PMWindowLibraryDLL methods !

sendDlgItemMsg: aWindow item: anInt msg: anInt2 mp1Struct: aByteObject mp2: ulong
    <api: WINSENDDLGITEMMSG handle short short struct long short>
    ^self invalidArgument!

sendDlgItemMsg: aWindow item: anInt msg: anInt2 mp1Struct: aByteObject1 mp2Struct:
                  aByteObject2
    <api: WINSENDDLGITEMMSG handle short short struct struct short>
    ^self invalidArgument! !


!DictionaryInspector methods !

add
        "Private - Add a new key to the dictionary
         by prompting for a new key expression."
    | key |
    key := Prompter
        prompt: 'new key expression'
        defaultExpression: String new.
    key isNil
        ifTrue: [^self].
    (object includesKey: key)
        ifTrue: [
```

```
            Menu message: 'key already in dictionary'.
            ^self].
    object at: key put: nil.
    self addKey: key.!

addKey: aKey
        "Private - add a new key value at its own place to the list pane
        and make that the current item."
    | size index |
    instList add:
        (Association key: aKey value: aKey printString).
    size := instList size.
    index := 1.
    [index > size
        or: [(instList at: index) key = aKey]]
                whileFalse: [index := index + 1].
    instIndex := index.
    self
        changed: #instVarList:
            with: #restoreSelected:
            with: instIndex;
        changed: #instance: ;
        enableInspectItem.! !



!TextWindow methods !

compressChanges
        Smalltalk compressChanges!

compressSources
        Smalltalk compressSources!

fileMenu
        "Private - Answer the File menu."
    | menu |
    menu := super fileMenu.
    self = Transcript ifTrue: [
        menu appendItem: 'Graphics Demo' selector: #openDemo.
        menu appendItem: 'About V...' selector: #aboutV.
        menu appendSeparator.
        menu appendItem: 'Compress Changes' selector: #compressChanges.
        menu appendItem: 'Compress Sources' selector: #compressSources.
        menu appendItem: 'File Changes Out' selector: #fileOutChanges.
        menu appendItem: 'Load Sources' selector: #loadSources].
    ^menu!

fileOutChanges
        Smalltalk logAllChanges!

loadSources
        Smalltalk loadSources! !



!DialogBox methods !

exportFromMLE: control
        "This method returns the data from an MLE as a string."
    | result mode1 mode2 |
    result := String new: (PMWindowLibrary
        sendDlgItemMsg: handle
        item: control
        msg: MlmQueryTextLength
        mp1: 0
        mp2: -1).
    (result size > 0)
        ifTrue: [
            PMWindowLibrary
                sendDlgItemMsg: handle
                item: control
                msg: MlmSetImportExport
                mp1Struct: result
                mp2: result size.
            "The strings are used to create pointers to simple
            integers. Sigh."
```

```
            mode1 := String
                with: 0 asCharacter
                with: 0 asCharacter
                with: 0 asCharacter
                with: 0 asCharacter.
            mode2 := String
                with: (result size \\ 256) asCharacter
                with: (result size // 256) asCharacter
                with: 0 asCharacter
                with: 0 asCharacter.
           PMWindowLibrary
                sendDlgItemMsg: handle
                item: control
                msg: MlmExport
                mp1Struct: mode1
                mp2Struct: mode2].
        ^result!

importToMLE: control string: aString
        "This method imports the data in the given string to a
        multiline-edit field and deletes the old text in the control."
    | temp mode oldLen |
    oldLen := PMWindowLibrary
        sendDlgItemMsg: handle
        item: control
        msg: MlmQueryTextLength
        mp1: 0
        mp2: -1.
    PMWindowLibrary
        sendDlgItemMsg: handle
        item: control
        msg: MlmDelete
        mp1: 0
        mp2: oldLen.
    temp := aString, Cr, Lf.
    "The string is used to create a pointer to zero."
    mode := String
        with: 0 asCharacter
        with: 0 asCharacter
        with: 0 asCharacter
        with: 0 asCharacter.
    PMWindowLibrary
        sendDlgItemMsg: handle
        item: control
        msg: MlmSetImportExport
        mp1Struct: temp
        mp2: temp size.
    PMWindowLibrary
        sendDlgItemMsg: handle
        item: control
        msg: MlmImport
        mp1Struct: mode
        mp2: temp size.!

processInput
        "Make the receiver modal to its owner window.
         This method doesn't return until close has been
         sent to the receiver."
    | cursor |
    Processor currentProcessIsRecursive ifTrue: [
        ^self
        "self error: 'Cannot do modal dialog during recursion.'"].
    owner disable.
    cursor := Cursor.
    CursorManager normal change.
    sem := Semaphore new.
    [CurrentProcess makeUserIF. Notifier run] fork.
    sem wait.
    CurrentProcess makeUserIF.
    cursor change.! !
```

14. APPENDIX F: THE WORKTIME MANAGER

14.1. WorktimeManager

```
ApplicationWindow subclass: #WorktimeManager
  instanceVariableNames:
    'dBase '
  classVariableNames: ''
  poolDictionaries:
    'PMConstants ' !


!WorktimeManager class methods ! !



!WorktimeManager methods !

buildMenuBar
        "Adds the own menus to the standard application menus."
     menuWindow addMenu: self dialogsMenu owner: self.
     menuWindow addMenu: self reportsMenu owner: self.
     super buildMenuBar!

close
        "Close the database connections before closing everything else.
        If the system is in application mode, exit the whole system as well."
    dBase close.
    super close.
    Smalltalk isRunTime
        ifTrue: [Smalltalk exit]!

dailyReport
        "Prints out the report of the hours done and the explanations."
    | report  interval |
    interval := ReportIntervalDialog prompt.
    interval notNil ifTrue: [
        report := ReportWindow new.
        report openRows: 25 cols: 79 label: 'Daily Report of'.
        self dailyReportTo: report of: interval]!

dailyReportTo: report of: dates
        "Prints out the report of the hours done and the explanations."
    | query |
    query := Relation openIn: dBase.
    query select: 'WDATE, TASK, WMIN/60, INFO' from: 'ACCOUNT';
        where: 'WDATE BETWEEN :firstDay AND :lastDay'.
    query bindTo: dates.
    report show: 'Detailed hours report from ', (dates at: #firstDay) asISOstring,
        ' to ', (dates at: #lastDay) asISOstring.
    report cr; cr.
    report contents: query.
    report
        columnTitle: 1 to: 'Date';
        columnTitle: 2 to: 'Project';
        columnTitle: 3 to: 'Hours';
        columnWidth: 3 to: 6;
        columnTitle: 4 to: 'Explanation';
        columnWidth: 4 to: 48.
    report show.
    query select: 'SUM(WMIN)/60' from: 'ACCOUNT';
        where: 'WDATE BETWEEN :firstDay AND :lastDay'.
    query bindTo: dates.
    report cr.
    report show: 'The total sum of hours at the time is '.
    report  show: (query row:1 col:1) asString.
    report cr.
    query close!

dialogsMenu
        "Returns the choices characteristic to this particular application."
```

```
    ^Menu new
        appendItem: '~Months Accounts  Alt+M' selector: #monthEdit accelKey: $m
                accelBits: AfAlt | AfChar;
        appendSeparator;
        appendItem: 'All ~Accounts      Alt+A' selector: #openAccounts accelKey: $a
                accelBits: AfAlt | AfChar ;
        appendItem: '~Tasks     Alt+T' selector: #openTasks accelKey: $t accelBits:
                AfAlt | AfChar ;
        appendItem: '~Work itemsAlt+W' selector: #openWorks accelKey: $w accelBits:
                AfAlt | AfChar ;
        appendItem: '~Projects  Alt+P' selector: #openProjects accelKey: $p
                accelBits: AfAlt | AfChar ;
        appendSeparator;
        appendItem: 'List tasks' selector: #listTasks;
        appendItem: 'List works' selector: #listWorks;
        appendItem: 'List projects' selector: #listProjects;
        title: '~Edit / List'!

header: aPane
        "Returns a comment string as the initial contents of the background
        pane."
    aPane contents: 'Worktime Manager v2.0 by JuM (c) Nokia Edacom Oy 1991'!

hoursReport
        "Prints out the report of the hours done under work-numbers."
    | report  interval |
    interval := ReportIntervalDialog prompt.
    interval notNil ifTrue: [
        report := ReportWindow new.
        report openRows: 12 cols: 58 label: 'Hours Report'.
        self hoursReportTo: report of: interval].!

hoursReportTo: aReport of: dates
        "Prints the hours done under work-numbers to the given report."
    | query |
    query := Relation openIn: dBase.
    query select: 'WORKS. NR, WORKS. NAME, SUM (ACCOUNT.WMIN)/60'
        from: 'WORKS, TASKS, ACCOUNT';
        where: 'WORKS. NR = TASKS. WORKNR AND TASKS. ID = ACCOUNT. TASK ',
            'AND ACCOUNT. WDATE BETWEEN :firstDay AND :lastDay';
        groupBy: 'WORKS.NR, WORKS.NAME'.
    query bindTo: dates.
    aReport show: 'Hours Report from ', (dates at: #firstDay) asISOstring,
        ' to ', (dates at: #lastDay) asISOstring.
    aReport cr; cr.
    aReport contents: query.
    aReport
        columnTitle: 1 to: 'WorkNr';
        columnWidth: 1 to: 6;
        columnTitle: 2 to: 'Name';
        columnTitle: 3 to: 'Hours'.
    aReport show.
    query select: 'SUM (ACCOUNT. WMIN)/60' from: 'ACCOUNT';
        where: 'ACCOUNT. WDATE BETWEEN :firstDay AND :lastDay ',
            'AND ACCOUNT. TASK IN ',
                '(SELECT TASKS. ID FROM TASKS WHERE TASKS. WORKNR IN ',
                    '(SELECT WORKS. NR FROM WORKS))'.
    query bindTo: dates.
    aReport cr.
    aReport show: 'The total sum of hours at the time is '.
    aReport  show: (query row:1 col:1) asString.
    aReport cr.
    query close!

init
        "Set up the instance variables etc."
    dBase := Database openAsSysadmin: 'JOBS'!

initSize
        "Returns the correct initial size."
    ^Rectangle origin: 32 @ 360 corner: 548 @ 480!

listProjects
        "Show the contents of the projects-table in a separate window."
    (Relation openIn: dBase) select: '*' from: 'PROJECTS'; orderBy: 'NR'; show!

listTasks
```

```
        "List the tasks out into a window."
    (Relation openIn: dBase) select: '*' from: 'TASKS'; orderBy: 'ID'; show!

listWorks
        "Show the contents of the works-table in a separate window."
    (Relation openIn: dBase) select: '*' from: 'WORKS'; orderBy: 'NR'; show!

monthEdit
        "Open and launch a new month editor."
    | monthEditor |
    monthEditor := AccountsInspector new.
    monthEditor owner: self.
    monthEditor openIn: dBase!

monthReport
        "Makes both the hours and daily reports as is customary with
        month reporting."
    | report interval |
    interval := ReportIntervalDialog prompt.
    interval notNil ifTrue: [
        report := ReportWindow new.
        report openRows: 40 cols: 79 label: 'Month Report'.
        self hoursReportTo: report of: interval.
        report cr.
        self dailyReportTo: report of: interval]!

open
        "Initializes the window and makes it visible."
    self label: 'Application'.
    self addSubpane: (
        TextPane new
            owner: self;
            when: #getContents perform: #header:;
            framingBlock: [:box| box];
            textFont: Font defaultFixedFont
    ).
    self init.
    ^super open!

openAccounts
        "Opens a modal accounts dialog."
    AccountsDialog new openOn: (AccountTable openIn: dBase).!

openProjects
        "Opens a modal projects- dialog."
    ProjectsDialog new openOn: (ProjectTable openIn: dBase).!

openTasks
        "Opens a modal tasks dialog."
    TasksDialog new openOn: (TaskTable openIn: dBase).!

openWorks
        "Opens a modal works dialog."
    WorksDialog new openOn: (WorkTable openIn: dBase).!

reportsMenu
        "Returns the reports menu."
    ^Menu new
        appendItem: '~Hours report     Alt+H' selector: #hoursReport
                    accelKey: $h accelBits: AfAlt | AfChar ;
        appendItem: '~Daily report     Alt+D' selector: #dailyReport
                    accelKey: $d accelBits: AfAlt | AfChar ;
        appendSeparator;
        appendItem: '~Month report     Alt+M' selector: #monthReport
                accelKey: $m accelBits: AfAlt | AfChar;
        title: '~Reports'.! !
```

14.2. AccountTuple

```
Tuple subclass: #AccountTuple
  instanceVariableNames: ''
  classVariableNames:
    'Tasks '
```

```
  poolDictionaries: ''  !


!AccountTuple class methods !

keyFields
        "The unique values are distinquished by the date and task-id."
    ^#(date task)!

new
        "Returns a new account-tuple with initial values."
    | result |
    result := super new.
    result
        directAt: #date put: (Date today);
        directAt: #task put: '';
        directAt: #info put: '';
        directAt: #duration put: (Time fromSeconds: 0).
    ^result!

setTaskSelection: aRelation
        "Private - tell which are the possible values of the project-id."
    Tasks := (aRelation asArray collect: [:i | i at: 1]) asSortedCollection! !



!AccountTuple methods !

dbAssociationAt: aKey
        "This method returns those column-name-value-associations which
        differ from the tuple's own representation."
    aKey
        when: #date do: [
            ^Association key: 'wdate' value: ((self at: #date) dbData)];
        when: #duration do: [
            ^Association key: 'wmin' value: ((self at: #duration) asSeconds / 60)].
    "otherwise"
    ^super dbAssociationAt: aKey!

dbAssociationPut: aPair
        "Converts those column-name-value pairs which differ from
        the tuple's standard representation."
    (aPair key)
        when: 'wdate' do: [
            self directAt: #date put: (Date dbData: aPair value).
            ^self];
        when: 'wmin' do: [
            self directAt: #duration put: (Time fromSeconds: (60 * aPair value)).
            ^self].
    "otherwise"
    super dbAssociationPut: aPair!

taskChoices
        "Return the allowed values for the task-field."
    ^Tasks! !
```

14.3. AccountTable

```
KeyedTable subclass: #AccountTable
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !


!AccountTable class methods !

deleteStatement
        "Private - returns the delete statement by which an account row
        is removed from the table."
    ^'DELETE FROM ACCOUNT ',
    'WHERE WDATE = :kDate AND TASK = :kTask'!

insertStatement
```

```
        "Private - returns the insert statement by which new account rows
        are added to the table."
    ^'INSERT INTO ACCOUNT ',
    'VALUES (:WDATE, :TASK, :INFO, :WMIN)'!

openIn: inDatabase
        "Opens the account-table in the Jobs-database for reading and writing
        and returns a handle to it."
    | result |
    result := super openIn: inDatabase ofName: 'ACCOUNT'.
    result init: inDatabase.
    ^result!

rowClass
        "The row class of this table is AccountTuple"
    ^AccountTuple!

selectNextStatement
        "Private - returns the select statement by which an account row
        is selected from the table."
   ^'SELECT * FROM ACCOUNT WHERE ',
    '(WDATE = :kDate AND TASK > :kTask) OR (WDATE > :kDate) ',
    'ORDER BY WDATE, TASK ASC'!

selectPrevStatement
        "Private - returns the select statement by which an account row
        is selected from the table according to the date and task ids."
    ^'SELECT * FROM ACCOUNT WHERE ',
    '(WDATE = :kDate AND TASK < :kTask) OR (WDATE < :kDate) ',
    'ORDER BY WDATE DESC, TASK DESC'!

selectThisStatement
        "Private - returns the select statement by which an account row
        is selected from the table."
    ^'SELECT * FROM ACCOUNT ',
    'WHERE WDATE = :kDate AND TASK = :kTask'!

updateStatement
        "Private - returns the update statement by which existing account rows
        are modified in the table."
    ^'UPDATE ACCOUNT ',
    'SET WDATE = :wdate, TASK = :task, INFO = :info, WMIN = :wmin ',
    'WHERE WDATE = :kDate AND TASK = :kTask'! !



!AccountTable methods !

init: inDatabase
        "Initalizes the AccountTuple-class."
    | tasks |
    tasks := Relation openIn: inDatabase.
    tasks select: 'ID' from: 'TASKS';
        where: 'WORKNR IN (SELECT NR FROM WORKS)'.
    AccountTuple setTaskSelection: tasks.
    tasks close.!

startTuple
        "With accounts we're more interested of the last ones."
    | lastToDate |
    lastToDate := AccountTuple new.
    lastToDate at: #task put: '9999'.
    ^self prevTo: lastToDate dbKey! !
```

14.4. AccountsDialog

```
TableDialog subclass: #AccountsDialog
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''  !


!AccountsDialog class methods !
```

```
itemIds
    | itemIds |
    itemIds := super itemIds.
    itemIds
         at: 257 put: #date;
         at: 'date' put: 257;
         at: 258 put: #task;
         at: 'task' put: 258;
         at: 259 put: #duration;
         at: 'duration' put: 259;
         at: 260 put: #info;
         at: 'info' put: 260.
    ^itemIds! !


!AccountsDialog methods !

getLayOut
        "Loads the dialog from resources."
    "self fromResFile: 'res\accounts.res'."
    self fromModule: (DynamicLinkLibrary open: 'worktime') id: 1.
    self setTextLimit: (self itemIds at: 'info') to: 200.! !
```

14.5. AccountDialog

```
ModelessTupleDialog subclass: #AccountDialog
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''   !


!AccountDialog class methods !

itemIds
        "Private - initializes the mapping between tuple components and
        dialog controls."
    | result |
    result := super itemIds.
    result
         at: 257 put: #date;
         at: 'date' put: 257;
         at: 258 put: #task;
         at: 'task' put: 258;
         at: 259 put: #duration;
         at: 'duration' put: 259;
         at: 260 put: #info;
         at: 'info' put: 260.
    ^result! !


!AccountDialog methods !

getLayOut
        "Loads the dialog from resources."
    "self fromResFile: 'res\account.res'."
    self fromModule: (DynamicLinkLibrary open: 'worktime') id: 6.
    self setTextLimit: (self itemIds at: 'info') to: 200.! !
```

14.6. NewAccountDialog

```
TupleDialog subclass: #NewAccountDialog
  instanceVariableNames: ''
  classVariableNames:
    'ItemIds '
  poolDictionaries: ''  !
```

```
!NewAccountDialog class methods !

itemIds
        "Private - Initialize the mapping between tuple components and
        dialog controls."
    | itemIds |
    itemIds := super itemIds.
    itemIds
         at: 257 put: #date;
         at: 'date' put: 257;
         at: 258 put: #task;
         at: 'task' put: 258.
    ^itemIds! !


!NewAccountDialog methods !

getLayOut
        "Loads the dialog from resources."
    "self fromResFile: 'res\newaccnt.res'."
    self fromModule: (DynamicLinkLibrary open: 'worktime') id: 7.! !
```

14.7. AccountsInspector

```
TableInspector subclass: #AccountsInspector
  instanceVariableNames:
    'month '
  classVariableNames: ''
  poolDictionaries:
    'PMConstants '  !


!AccountsInspector class methods ! !



!AccountsInspector methods !

buildMenuBar
        "Private - add the owner's reports to the menu."
    super buildMenuBar.
    menuWindow addMenu: owner reportsMenu owner: self.!

dailyReport
        "Prints out the report of the hours done and the explanations."
    | report |
    report := ReportWindow new.
    report openRows: 25 cols: 79 label: 'Daily Report of ', month name.
    owner dailyReportTo: report of: month asInterval.!

dialogClass
        "Returns the class by which single accounts are handled."
    ^AccountDialog!

dialogSize
        "Private - Answer the size of the child dialog."
    ^363 @ 331!

hoursReport
        "Prints out the report of the hours done and the explanations."
    | report |
    report := ReportWindow new.
    report openRows: 12 cols: 58 label: 'Hours Report of ', month name.
    owner hoursReportTo: report of: month asInterval.!

keyDialogClass
        "Returns the class by which single accounts are handled."
    ^NewAccountDialog!

monthReport
        "Prints out the report of the hours done and the explanations."
    | report |
```

```
    report := ReportWindow new.
    report openRows: 20 cols: 79 label: 'Monthreport of ', month name.
    owner hoursReportTo: report of: month asInterval.
    owner dailyReportTo: report of: month asInterval.!

newMonth
        "Get the new month from the user."
    | resp |
    resp := month.
    (MonthDialog new openOn: resp) isNil
        ifFalse: [
            self newMonth: resp.
            self setInstList.
            self changed: #instVarList: ].!

newMonth: aTuple
        "Sets the scope in the table to include the accounts in the
        given month."
    month := aTuple.
    table select: '*' from: 'ACCOUNT';
        where: 'WDATE BETWEEN :firstDay AND :lastDay'.
    table bindTo: month asInterval.
    self setLabel!

openIn: inDatabase
        "Before the inspector is opened the accounts of the current month
        are selected into the table."
    | firstDay lastDay |
    table := AccountTable openIn: inDatabase.
    self newMonth: MonthTuple new.
    super openOn: table.
    self setLabel!

setLabel
        "Set the label of the window."
    self label: (month at: #month) asString, ' ', (month at: #year) asString!

tableMenu
        "The select-month function is added to the menu."
    | menu |
    menu := super tableMenu.
    menu
        appendSeparator;
        appendItem: 'Select ~Month     Alt+M' selector: #newMonth
            accelKey: $m accelBits: AfAlt  | AfChar.
    ^menu! !
```

14.8. TaskTuple

```
Tuple subclass: #TaskTuple
  instanceVariableNames: ''
  classVariableNames:
    'Works '
  poolDictionaries: '' !


!TaskTuple class methods !

keyFields
        "Returns a list of the keyfields."
    ^#(ident)!

new
        "Creates an empty task-instance."
    | result |
    result := super new.
    result
        directAt: #ident put: '';
        directAt: #workNr put: 0;
        directAt: #workName put: '';
        directAt: #info put: '';
        directAt: #totalHours put: (Fixed new: 0.0);
        directAt: #thisMonthsHours put: (Fixed new: 0.0).
```

```
    ^result!

setWorkSelection: aCollection
        "Sets the work-number selection according to the argument which
        is a collection of two-element arrays with worknumber and names in."
    Works := Dictionary new: aCollection size.
    aCollection do: [:each |
        Works at: (each at: 1) put: (each at: 2)].!

virtualFields
        "Tells which fields come from other tables."
    ^#(totalhours thismonthshours workname)!

works
    ^Works! !



!TaskTuple methods !

dbAssociationAt: aKey
        "The names differ in the tuple and in the database."
     (aKey = #ident)
        ifTrue: [^Association key: 'id' value: (self directAt: #ident) dbData]
        ifFalse: [^super dbAssociationAt: aKey]!

dbAssociationPut: anAssociation
        "The names differ in the tuple and in the database."
     (anAssociation key)
        when: 'id'       do: [^self at: #ident put: anAssociation value];
        when: 'name' do: [^self at: #workName put: anAssociation value].
    super dbAssociationPut: anAssociation!

workname: newName
        "When a new workname is selected, the worknumber must be
        updated accordingly."
    self at: #workNr put: (Works at: newName).
    self hasChanged: #workNr!

worknameChoices
        "Returns the selection of available worknames."
    ^Works keys asOrderedCollection asArray! !
```

14.9. TaskTable

```
KeyedTable subclass: #TaskTable
  instanceVariableNames:
    'hours '
  classVariableNames: ''
  poolDictionaries: ''    !


!TaskTable class methods !

deleteStatement
        "Private - returns the delete statement by which an account row
        is removed from the table."
    ^'DELETE FROM TASKS ',
    'WHERE ID = :kIdent'!

insertStatement
        "Private - returns the insert statement by which new rows
        are added to the table."
    ^'INSERT INTO TASKS VALUES (:Id, :WorkNr, :Info)'!

openIn: inDatabase
        "Opens the tasks-table in the Jobs-database for reading and writing
        and returns a handle to it."
    | result |
    result := super openIn: inDatabase.
    result init: inDatabase.
    result select: 'ID, WORKNR, NAME, INFO' from: 'TASKS, WORKS';
        where: 'WORKNR = NR'; orderBy: 'ID'.
```

```
    ^result!

rowClass
        "The row class of this table is TaskTuple"
    ^TaskTuple!

selectNextStatement
    ^'SELECT ID, WORKNR, NAME, INFO FROM TASKS, WORKS ',
    'WHERE WORKNR = NR AND ID > :kIdent ORDER BY ID'!

selectPrevStatement
    ^'SELECT ID, WORKNR, NAME, INFO FROM TASKS, WORKS ',
    'WHERE WORKNR = NR AND ID < :kIdent ORDER BY ID DESC'!

selectThisStatement
    ^'SELECT ID, WORKNR, NAME, INFO FROM TASKS, WORKS ',
    'WHERE WORKNR = NR AND ID = :kIdent'!

updateStatement
    ^'UPDATE TASKS ',
    'SET ID = :Id, WORKNR = :workNr, INFO = :info ',
    'WHERE ID = :kIdent'! !



!TaskTable methods !

at: aKey put: aTuple
        "Take care that the account rows follow the task item they belong to.
        Terminate the operation if it doesn't succeed."
    (aKey at: #kId) = (aTuple at: #id)
        ifFalse: [
            hours update: 'ACCOUNT' set: 'ID = :1'; where: 'ID = :2'.
            hours bindTo: (Array with: (aTuple at: #id) with: (aKey at: #kId)).
            hours execute ifFalse: [^false]].
    ^super at: aKey put: aTuple.!

delete: aKey
        "Checks that there are no dangling account rows before allows
        the task to be removed."
    (0.0 < (self getValue at: #totalhours)) ifTrue: [
        ^self holdOn: 'There are working periods connected to this task. Cannot
                    remove it.']
    ifFalse: [^super delete: aKey]!

getValue
        "Private - overrides the default method. Data is needed also
        from the works-table."
    | result taskId firstDay lastDay temp |
    result := super getValue.
    taskId := result at: #ident.
    hours select: 'SUM(WMIN)/60' from: 'ACCOUNT';
        where: 'TASK = :1'.
    hours bindTo: taskId.
    temp := hours row: 1 col: 1.
    temp isNil
        ifTrue: [result at: #totalhours put: (Fixed new: 0.0)]
        ifFalse: [result at: #totalhours put: (Fixed dbData: temp)].
    lastDay := Date today.
    firstDay := lastDay firstDayOfMonth.
    hours select: 'SUM(WMIN)/60' from: 'ACCOUNT';
        where: 'TASK = :1 AND WDATE BETWEEN :2 AND :3'.
    hours bindTo: (Array with: taskId with: firstDay with: lastDay).
    temp := hours row: 1 col: 1.
    temp isNil
        ifTrue: [result at: #thisMonthsHours put: (Fixed new: 0.0)]
        ifFalse: [result at: #thisMonthsHours put: (Fixed dbData: temp)].
    ^result!

init: inDatabase
        "Initalizes the instance variables and the TaskTuple-class."
    | works |
    works := Relation openIn: inDatabase.
    works select: 'NAME, NR' from: 'WORKS';
        orderBy: 'NAME'.
    TaskTuple setWorkSelection: works.
    works close.
```

```
    hours := Table openIn: inDatabase.! !
```

14.10. TasksDialog

```
TableDialog subclass: #TasksDialog
  instanceVariableNames: ''
  classVariableNames:
    'ItemIds '
  poolDictionaries: ''   !


!TasksDialog class methods !

itemIds
    | itemIds |
    itemIds := super itemIds.
    itemIds
        at: 263 put: #ident;
        at: 'ident' put: 263;
        at: 264 put: #workname;
        at: 'workname' put: 264;
        at: 265 put: #totalhours;
        at: 'totalhours' put: 265;
        at: 266 put: #thismonthshours;
        at: 'thismonthshours' put: 266;
        at: 267 put: #info;
        at: 'info' put: 267.
    ^itemIds! !


!TasksDialog methods !

getLayOut
        "Loads the dialog from resources."
    self fromModule: (DynamicLinkLibrary open: 'worktime') id: 2.
    "self fromResFile: 'res\tasks.res'."! !
```

14.11. WorkTuple

```
Tuple subclass: #WorkTuple
  instanceVariableNames: ''
  classVariableNames:
    'Bosses Projects '
  poolDictionaries: ''   !


!WorkTuple class methods !

keyFields
        "Returns the database key field"
    ^#(worknumber)!

new
        "CredirectAte a new worktuple instance."
    | result |
    result := super new.
    result
        directAt: #workNumber put: 0;
        directAt: #workName put: '';
        directAt: #bossName put: '';
        directAt: #projectName put: '';
        directAt: #projectNumber put: 0;
        directAt: #totalHours put: (Fixed new: 0.0);
        directAt: #thisMonthsHours put: (Fixed new: 0.0).
    ^result!

setProjectSelection: aCollection
        "Intializes the possible project names and numbers."
    Projects := Dictionary new: aCollection size.
```

```
    Bosses := Dictionary new: aCollection size.
    aCollection do: [:each |
        Projects at: (each at: 1) put: (each at: 2).
        Bosses at: (each at: 1) put: (each at: 3)]!

virtualFields
        "Tells which fields come from other tables."
    ^#(bossname projectname totalhours thismonthshours)! !


!WorkTuple methods !

dbAssociationAt: aKey
        "This method returns those column-name-value-associations which
        differ from the tuple's own representation."
    aKey
        when: #worknumber do: [
            ^Association key: 'nr' value: (self at: #workNumber) dbData];
        when: #workname do: [
            ^Association key: 'name' value: (self at: #workName) dbData];
        when: #projectnumber do: [
            ^Association key:  'projnr' value: (self at: #projectNumber) dbData].
    ^nil!

dbAssociationPut: aPair
        "Converts those column-name-value pairs which differ from
        the tuple's standard representation. Notice how the qualified
        names are used."
    (aPair key)
        when: 'works.name' do: [
            self directAt: #workName put: aPair value];
        when: 'works.projnr' do: [
            self directAt: #projectNumber put: aPair value];
        when: 'works.nr' do: [
            self directAt: #workNumber put: aPair value];
        when: 'projects.boss' do: [
            self directAt: #bossName put: aPair value];
        when: 'projects.name' do: [
            self directAt: #projectName put: aPair value].
    ^self!

projectname: newName
        "When the project name changes, the number and bossname have
        to be updated respectively."
    self directAt: #projectName put: newName.
    self directAt: #projectNumber put: (Projects at: newName).
    self directAt: #bossName put: (Bosses at: newName).
    self hasChanged: #projectNumber.
    self hasChanged: #bossName.!

projectnameChoices
        "Returns the possible values for project-names."
    ^Projects keys asSortedCollection asArray! !
```

14.12. WorkTable

```
KeyedTable subclass: #WorkTable
  instanceVariableNames:
    'hoursView '
  classVariableNames: ''
  poolDictionaries: ''    !


!WorkTable class methods !

deleteStatement
        "Private - returns the delete statement by which an account row
        is removed from the table."
    ^'DELETE FROM WORKS ',
    'WHERE NR = :kWorkNumber'!

insertStatement
```

```
        "Private - returns the insert statement by which new rows
        are added to the table."
    ^'INSERT INTO WORKS VALUES (:Nr, :Name, :ProjNr)'!

openIn: inDatabase
        "Opens the Works-table in the Jobs-database for reading and writing
        and returns a handle to it."
    | result |
    result := super openIn: inDatabase.
    result select: 'WORKS.NR, WORKS.NAME, WORKS.PROJNR, ',
            'PROJECTS.BOSS, PROJECTS. NAME '
        from: 'WORKS, PROJECTS';
        where: 'WORKS.PROJNR = PROJECTS.NR ';
        orderBy: 'WORKS.NR'.
    result init: inDatabase.
    ^result!

rowClass
        "This table contains worktuples"
    ^WorkTuple!

selectNextStatement
    ^'SELECT WORKS.NR, WORKS.NAME, WORKS.PROJNR, ',
    'PROJECTS.BOSS, PROJECTS.NAME ',
    'FROM WORKS, PROJECTS WHERE WORKS. NR > :kWorkNumber ',
    'AND WORKS.PROJNR = PROJECTS.NR ',
    'ORDER BY WORKS.NR'!

selectPrevStatement
    ^'SELECT WORKS.NR, WORKS.NAME, WORKS.PROJNR, ',
    'PROJECTS.BOSS, PROJECTS.NAME ',
    'FROM WORKS, PROJECTS WHERE WORKS. NR < :kWorkNumber ',
    'AND WORKS.PROJNR = PROJECTS.NR ',
    'ORDER BY WORKS.NR DESC'!

selectThisStatement
    ^'SELECT WORKS.NR, WORKS.NAME, WORKS.PROJNR, ',
    'PROJECTS.BOSS, PROJECTS.NAME ',
    'FROM WORKS, PROJECTS WHERE WORKS. NR = :kWorkNumber ',
    'AND WORKS.PROJNR = PROJECTS.NR '!

updateStatement
    ^'UPDATE WORKS ',
    'SET NR = :Nr, NAME = :Name, PROJNR = :ProjNr ',
    'WHERE NR = :kWorkNumber'! !



!WorkTable methods !

getValue
        "Private - overrides the default method. Data is needed also
        from the account-table for the hour-sums."
    | result totalHours monthsHours workNr |
    result := (self class rowClass) dbData: cursor namedValue.
    workNr := result at: #workNumber.

    hoursView select: 'SUM(WMIN)/60' from: 'ACCOUNT';
        where: 'TASK IN (SELECT ID FROM TASKS WHERE ',
            'WORKNR = :1)'.
    hoursView bindTo: (Array with: workNr).
    totalHours := hoursView row: 1 col: 1.
    totalHours isNil
        ifTrue: [totalHours := Fixed new: 0.0]
        ifFalse: [totalHours := Fixed dbData: totalHours].
    result at: #totalHours put: totalHours.

    hoursView select: 'SUM(WMIN)/60' from: 'ACCOUNT';
        where: ' WDATE BETWEEN :2 AND :3 AND ',
            'TASK IN (SELECT ID FROM TASKS WHERE ',
            'WORKNR = :1)'.
    hoursView bindTo: (
        Array with: workNr
            with: (Date today firstDayOfMonth)
            with: (Date today)).
    monthsHours := hoursView row: 1 col: 1.
    monthsHours isNil
```

```
            ifTrue: [monthsHours := Fixed new: 0.0]
            ifFalse: [monthsHours := Fixed dbData: monthsHours].
        result at: #thisMonthsHours put: monthsHours.
        ^result!

    init: inDatabase
            "Initalizes the instance variables and the WorkTuple class."
         | projects |
        projects := Relation openIn: inDatabase.
        projects select: 'NAME, NR, BOSS' from: 'PROJECTS';
            orderBy: 'NAME'.
        WorkTuple setProjectSelection: (projects asArray).
        projects close.

        hoursView := Relation openIn: inDatabase.! !
```

14.13. WorksDialog

```
TableDialog subclass: #WorksDialog
  instanceVariableNames: ''
  classVariableNames:
    'ItemIds '
  poolDictionaries: ''   !


!WorksDialog class methods !

itemIds
    | itemIds |
    itemIds := super itemIds.
    itemIds
        at: 272 put: #bossname;
        at: 'bossname' put: 272;
        at: 271 put: #projectname;
        at: 'projectname' put: 271;
        "at: 270 put: #projectnumber;
        at: 'projectnumber' put: 270;"
        at: 269 put: #worknumber;
        at: 'worknumber' put: 269;
        at: 264 put: #workname;
        at: 'workname' put: 264;
        at: 265 put: #totalhours;
        at: 'totalhours' put: 265;
        at: 266 put: #thismonthshours;
        at: 'thismonthshours' put: 266.
    ^itemIds! !



!WorksDialog methods !

getLayOut
        "Loads the dialog from resources."
    self fromModule: (DynamicLinkLibrary open: 'worktime') id: 3.
    "self fromResFile: 'res\works.res'."! !
```

14.14. ProjectTuple

```
Tuple subclass: #ProjectTuple
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''  !


!ProjectTuple class methods !

keyFields
        "This table has only one keyfield."
    ^#(projectnumber)!
```

```
new
        "Creates a new instance of project-tuple."
    | result |
    result := super new.
    result
        directAt: #bossname put: '';
        directAt: #projectname put: '';
        directAt: #projectnumber put: 0;
        directAt: #totalhours put: (Fixed new: 0.0);
        directAt: #thismonthshours put: (Fixed new: 0.0).
    ^result!

virtualFields
        "Tells which fields come from other tables."
    ^#(totalhours thismonthshours)! !


!ProjectTuple methods !

dbAssociationAt: aKey
        "This method returns those column-name-value-associations which
        differ from the tuple's own representation."
    aKey
        when: #projectnumber do: [
            ^Association key: 'nr' value: (self at: #projectNumber) dbData];
        when: #bossname do: [
            ^Association key: 'boss' value: (self at: #bossName) dbData];
        when: #projectname do: [
            ^Association key: 'name' value: (self at: #projectName) dbData].!

dbAssociationPut: aPair
        "Converts those column-name-value pairs which differ from
        the tuple's standard representation."
    (aPair key)
        when: 'name' do: [
            self directAt: #projectName put: aPair value];
        when: 'boss' do: [
            self directAt: #bossName put: aPair value];
        when: 'nr' do: [
            self directAt: #projectNumber put: aPair value].
    ^self! !
```

14.15. ProjectTable

```
KeyedTable subclass: #ProjectTable
  instanceVariableNames:
    'accountTable '
  classVariableNames: ''
  poolDictionaries: ''  !


!ProjectTable class methods !

deleteStatement
        "Private - returns the delete statement by which an account row
        is removed from the table."
    ^'DELETE FROM PROJECTS ',
    'WHERE NR = :kProjectNumber'!

insertStatement
        "Private - returns the insert statement by which new rows
        are added to the table."
    ^'INSERT INTO PROJECTS (NR, NAME, BOSS) VALUES (:Nr, :Name, :Boss)'!

openIn: inDatabase
        "Opens the Projects-table in the Jobs-database for reading and writing
        and returns a handle to it."
    | result |
    result := super openIn: inDatabase.
    result select: '*' from: 'PROJECTS';
        orderBy: 'NR'.
    result init: inDatabase.
```

```
    ^result!

rowClass
        "The row class of this table is TaskTuple"
    ^ProjectTuple!

selectNextStatement
    ^'SELECT * FROM PROJECTS WHERE NR > :kProjectNumber ',
    'ORDER BY NR'!

selectPrevStatement
    ^'SELECT * FROM PROJECTS WHERE NR < :kProjectNumber ',
    'ORDER BY NR DESC'!

selectThisStatement
    ^'SELECT * FROM PROJECTS WHERE NR = :kProjectNumber'!

updateStatement
    ^'UPDATE PROJECTS ',
    'SET NR = :Nr, NAME = :Name, BOSS = :Boss ',
    'WHERE NR = :kProjectNumber'! !


!ProjectTable methods !

getValue
        "Private - overrides the default method. Data is needed also
        from the account-table for the hour-sums."
    | result totalHours monthsHours projNr |
    result := (self class rowClass) dbData: cursor namedValue.
    projNr := result at: #projectNumber.

    accountTable select: 'SUM(WMIN)/60' from: 'ACCOUNT';
        where: 'TASK IN (SELECT ID FROM TASKS WHERE ',
            'WORKNR IN (SELECT NR FROM WORKS WHERE ',
            'PROJNR = :1))'.
    accountTable bindTo: (Array with: projNr).
    totalHours := accountTable row: 1 col: 1.
    totalHours isNil
        ifTrue: [totalHours := Fixed new: 0.0]
        ifFalse: [totalHours := Fixed dbData: totalHours].
    result at: #totalHours put: totalHours.

    accountTable select: 'SUM(WMIN)/60' from: 'ACCOUNT';
        where: ' WDATE BETWEEN :2 AND :3 AND ',
            'TASK IN (SELECT ID FROM TASKS WHERE ',
            'WORKNR IN (SELECT NR FROM WORKS WHERE ',
            'PROJNR = :1))'.
    accountTable bindTo: (
        Array with: projNr
            with: (Date today firstDayOfMonth)
            with: (Date today)).
    monthsHours := accountTable row: 1 col: 1.
    monthsHours isNil
        ifTrue: [monthsHours := Fixed new: 0.0]
        ifFalse: [monthsHours := Fixed dbData: monthsHours].
    result at: #thisMonthsHours put: monthsHours.
    ^result!

init: inDatabase
        "Opens a view to the accountTable when the projects table is opened."
    accountTable := Relation openIn: inDatabase.! !
```

14.16. ProjectsDialog

```
TableDialog subclass: #ProjectsDialog
  instanceVariableNames: ''
  classVariableNames:
    'ItemIds '
  poolDictionaries: ''    !
```

```
!ProjectsDialog class methods !

itemIds
    | itemIds |
    itemIds := super itemIds.
    itemIds
         at: 272 put: #bossname;
         at: 'bossname' put: 272;
         at: 271 put: #projectname;
         at: 'projectname' put: 271;
         at: 270 put: #projectnumber;
         at: 'projectnumber' put: 270;
         at: 265 put: #totalhours;
         at: 'totalhours' put: 265;
         at: 266 put: #thismonthshours;
         at: 'thismonthshours' put: 266.
    ^itemIds! !


!ProjectsDialog methods !

getLayOut
        "Loads the dialog from resources."
    self fromModule: (DynamicLinkLibrary open: 'worktime') id: 4.
    "self fromResFile: 'res\projects.res'."! !
```

14.17. MonthTuple

```
Tuple subclass: #MonthTuple
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''    !


!MonthTuple class methods !

new
        "Returns a new instance of a year-month pair."
    | result |
    result := super new.
    result
        directAt: #year put: (Date today year);
        directAt: #month put: (Date today monthName) asSymbol.
    ^result! !


!MonthTuple methods !

asInterval
        "Returns the first and the last day of the receiver in a tuple."
    | result help |
    result := Tuple new.
    help := Date newDay: 1 month: (self at: #month) year: (self at: #year).
    result at: #firstDay put: help.
    result at: #lastDay put: (help addDays: (help daysInMonth - 1)).
    ^result!

monthChoices
        "Return an array of possible month-names."
    | result |
    result := Array new: 12.
    1 to: 12 do: [:nr | result at: nr put: (Date nameOfMonth: nr) asSymbol].
    ^result!

name
        "Returns a pretty image of the receiver."
    ^(self at: #month) asString, ' ', (self at: #year) asString! !
```

14.18. MonthDialog

```
TupleDialog subclass: #MonthDialog
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !


!MonthDialog class methods !

itemIds
        "Return the binding between dialog controls and symbolic names."
    | itemIds |
    itemIds := super itemIds.
    itemIds
        at: 'year' put: 281;
        at: 281 put: #year;
        at: 'month' put: 282;
        at: 282 put: #month.
    ^itemIds! !



!MonthDialog methods !

getLayOut
        "Read the layout from the resources."
    "self fromResFile: 'res\monthdlg.res'."
    self fromModule: (DynamicLinkLibrary open: 'worktime') id: 5.! !
```

14.19. ReportIntervalDialog

```
TupleDialog subclass: #ReportIntervalDialog
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''    !


!ReportIntervalDialog class methods !

itemIds
        "Private - Returns the binding between tuple fields and dialog
        controls."
    | itemIds |
    itemIds := super itemIds.
    itemIds
        at: 'firstday' put: 283;
        at: 283 put: #firstday;
        at: 'lastday' put: 284;
        at: 284 put: #lastday.
    ^itemIds!

prompt
        "Prompt an interval of two days from the user. Return the result,
        (nil, if cancel was pressed.)"
     ^self new openOn: (MonthTuple new asInterval)! !



!ReportIntervalDialog methods !

getLayOut
        "Get the dialog from a resource file."
    "self fromResFile: 'res\interval.res'."
    self fromModule: (DynamicLinkLibrary open: 'worktime') id: 8.! !
```

15. APPENDIX G: DBENGINE

15.1. The interface

```
// ------------------------------------------------------------------------
//
//  SYSTEM:           OOPoS Proto
//
//  UNIT:             dbEngine
//
//  AUTHOR:           $Author:   JuM  $
//
//  $Workfile:   dbengine.h  $
//  $Revision:   2.6  $ $Date:   31 Jan 1991 07:11:18  $ $Author:   JuM  $
//
// ------------------------------------------------------------------------
//
//  ABSTRACTS:        This is the interface of the low-level SQL-database
//                    server designed especially for the high-level
//                    object-oriented programming tools Smalltalk/V PM
//                    and Choreographer.
//
//                    This layer has three objectives. For one, there are
//                    some technical requirements set by Smalltalk/V PM
//                    which concern the calling conventions of external
//                    modules. These limitations dictate the simplified
//                    scheme of the procedures to be used: only parameters
//                    of few primitive types are allowed, and no pointers
//                    are passed as parameters (with one expection).
//                    The result of an operation is returned as function
//                    result. Consequently, this scheme makes it impossible
//                    to use the native GTI-interface of SQLBase.
//
//                    The second reason for the existence of this module
//                    is that it hides the details of the used SQL-solution.
//                    The module has been designed so that it should be
//                    easy to implement on top of other SQL-servers, too.
//
//                    The third objective is to simplify and clarify the
//                    usage of SQL from an application program. For that
//                    reason this module provides automatically allocated
//                    bind-variables and select-buffers, encompanied with
//                    a minimal operation set for accessing the SQL server.
//                    The penalty to be paid for this is some overhead at
//                    runtime, but that cannot be regarded as an alien feature
//                    for environments like Smalltalk or Choreographer.
//
//                    Refer to the comments in this source about how to
//                    use the interface from an application.
//
// ------------------------------------------------------------------------
//
//  $Log:   D:/jumsoft/dbengine/v/dbengine.h  $
//
//                Rev 2.6   31 Jan 1991 07:11:18   JuM
//              Added a new error code for the case that
//              an invalid handle is given as the cursor.
//
//                Rev 2.5   25 Jan 1991 03:21:48   JuM
//              dbeColStatus changed to dbeColIsNull.
//
//                Rev 2.4   10 Jan 1991 23:14:48   JuM
//                 Changes made:
//                   DbeBack, dbeToEnd and dbeTableRows REMOVED from the module,
//                 since
//                   they wouldn't work in all cases or they returned unpredictable
//                   results. The 'enable fetch backwards' call is no more made in
//                   execute.
//                   DbeCompStore changed so that it now stores the compiled
//                 statement
//                   if the restore-attemp returns ANY error code.
```

```
//
//
//                    Copyright (c) 1990,1991 by Nokia Data
//
// ----------------------------------------------------------------------------

#ifndef DBENGINE_H
#define DBENGINE_H

#include <jumstd.h>

//  On architecture and design guidelines
//  -------------------------------------
//  This module has been designed with the object-oriented ideas in mind.
//  In practice it means that all of the operations have a common first
//  parameter, 'the object', to which 'messages are sent' when function
//  calls are made. The functions fall into two gathegories (with few
//  expections), which are called IMPULSEs and ATTRIBUTEs respectively.
//  The former change the status of the object without returning any data,
//  while the latter return various aspects of it without changing its
//  status. (For more info on this division see the OOPoS documentation.)
//  All function-names start with the prefix 'dbe' for database-engine.
//
//  The basic unit is called a 'connection', which represents one physical
//  access link to the database. Once connected, it may be attached to
//  a single SQL-statement and simple operations can be performed. Since
//  one SQL-connection can only fetch data from the database (as a query)
//  or write to it (as an insert/update/delete statement) but not both
//  at the same time, a group of co-operative connections is needed
//  for more sophisticated database maintenace. The task of combining
//  a set of connections together to form more expressive and more natural
//  database objects is left to the higher layers, however. As such, this
//  module may be seen as the C-implementation of the primitive 'connection'
//  class.
//
//  The interface has been grouped into three blocks. The first includes
//  operations for connecting to the database and performing such sql
//  statements which imply no data transfer between the database and
//  the application. The second group concentrates on writing to
//  the database via bind-variables while the third one includes operations
//  for making queries and managing select-buffers.

// ----------------------------------------------------------------------
// The main types

typedef unsigned short  connection;
#define CONSTRUCTOR connection ATTRIBUTE

#define NO_CONNECTION   255     // Returned from a constructor when no success
#define INVALID_PROGRAM_VARIABLE_NAME 0 // Returned from dbeBindNumber when
                                        // no success.

//  These are the primitive types in which data can be stored into
//  the database. All types in SQL-Table definitions are mapped into
//  these, so that
//      char, char (1)                      -> CHARACTER
//      char (n) | n>1                      -> STRING
//      smallint, dec (n, 0) | n <= 4       -> SHORT_INTEGER
//      integer, dec (n, 0) | 5 <= n <= 9   -> LONG_INTEGER
//      dec (n, 0) | n >= 10                -> VERY_LONG_INTEGER
//      number, real, float (n), double precision,
//          dec (n, m) | m > 0              -> REAL
//      date                                -> DATE
//      time                                -> TIME
//      datetime                            -> TIMEPOINT

//  The data can be transferred between the program and the database
//  in the textual format (as STRING) as long as the contents of the
//  string meet the semantics of the data type. In addition, some of
//  of the simple types can be set and get directly in their internal
//  representation (char, short, long) for efficiency. See the routines
//  reserved for that.

typedef enum {
    CHARACTER, STRING,
    SHORT_INTEGER, LONG_INTEGER, VERY_LONG_INTEGER, REAL,
    DATE, TIME, TIMEPOINT
```

```
} SQLDataType;

// ------------------------------------------------------------------
//  The error management is performed by returning a status code from
//  each of the IMPULSEs. If the value of it differs from OK, a textual
//  explanation can be acquired by the connection handle. Also the status
//  can be requeried by giving the handle; this can be done also after
//  CONSTRUCTORs, which return no error code. The ATTRIBUTEs are regarded
//  as fool-proof; no status code is returned but a suitable null value
//  if no reasonable value can be acquired becouse of an error-situation.

//  The error may come from one of two sources: either from the underlying
//  database implementation or from this layer itself. The more-significant
//  word of the 2-word status code tells that; it is 0 when SQL is the
//  source and 1 otherwise. When asking the error description that
//  distiction is transparent, however. Even when a constructor failed,
//  the status may be queried with the returned connection.

status ATTRIBUTE dbeConnectionStatus (connection c);
string ATTRIBUTE dbeErrorDescription (connection c);

#define MEMORY_EXHAUSTED            0x00010001
#define UNKNOWN_BINDVAR_NAME        0x00010002
#define UNKNOWN_SELBUF_NAME         0x00010003
#define UNBOUND_PROGRAM_VARIABLES   0x00010004
#define LONG_DATA_NOT_SUPPORTED     0x00010005 // ...in this version
#define FEATURE_NOT_IMPLEMENTED     0x00010006
#define STATEMENT_NOT_COMPILED      0x00010007
#define TOO_MANY_BINDINGS           0x00010008
#define UNCONTIGUOUS_BIND_NUMBERING 0x00010009
#define STATEMENT_NOT_FOUND         0x0001000A
#define INVALID_BIND_NUMBER         0x0001000B
#define INVALID_CURSOR_HANDLE       0x0001000C

// ------------------------------------------------------------------
//  The string management is done in a little bit original way due to
//  the limitations set by Smalltalk/V PM. When passed as in-parameters,
//  no null-termination is assumed and thus the length of the string
//  has to be given always. Respectively, attributes which should return a
//  string value return really a pointer to it. The actual value has
//  to be then fetched by first getting the length of the string for
//  allocating a suitable buffer and then the data with the two operations
//  defined below. Only strings smaller to 64 kB are implemented, due to
//  the limitations of the segment-architecture of 80286. To a C-programmer
//  the two operations would seem more familiar named as "strlen" and
//  "strncpy". Note that the stringCopy returns no status code and is
//  thus defined as "void-attribute".

unsigned ATTRIBUTE stringLength (string str);
void     ATTRIBUTE copyString   (string to, string from, int ofLength);

// This attribute returns a pointer to a string which contains
// a description of the module's version

string ATTRIBUTE dbeVersion (void);

// ------------------------------------------------------------------
//  Connection and transaction management.

//  A connection can be created by one of
//  two ways. With the first one the database name, user id and
//  password have to be given. More connections to the same database
//  can then be made by duplicating this first connection. After
//  usage the connections must be disabled. Any uncommitted
//  transactions will be automatically rolled back in disconnecting,
//  so be sure to commit before doing it.

CONSTRUCTOR dbeConnect
(
    string  dbName,
    int     dbNameLen,
    string  userId,
    int     userIdNameLen,
    string  password,
    int     passwordLen
);
CONSTRUCTOR dbeDuplicate  (connection c);
```

```
IMPULSE     dbeDisconnect (connection c);

boolean ATTRIBUTE dbeIsConnected (connection c);

// ------------------------------------------------------------------
// The transaction management.

IMPULSE     dbeCommit   (connection c);
IMPULSE     dbeRollback (connection c);

// ------------------------------------------------------------------
// The locking scheme borrows heavily from the underlying implementation.
// All writing operations lock the affected pages automatically until
// the transaction is ended. Attempts to read these pages block, and
// writing fails. The pages can also be locked via reading, either the
// current page or all of the pages from the beginning of the transaction
// (Isolation levels lockCurrent and lockAll.) Such pages can be read
// by other applications but attempts to update them stay to wait that
// the locks are released. If the isolation level is readOnly, no writing
// is possible by that application. Note that the isolation level
// is not per cursor but per user: thus, all cursors are at the same level.
// Notice that the default value for isolationLevel is lockAll, which
// gives very poor degree of concurrency and also relatively poor
// performance.

typedef enum {readOnly, lockNothing, lockCurrent, lockAll} lockingMode;

IMPULSE dbeSetIsolationLevel (connection c, lockingMode m);

// ------------------------------------------------------------------
//  SQL-statement management. Single isolated SQL-statements can be
//  performed with the operation dbeDoSQL, which compiles, executes
//  and commits an atomic statement. This is convinient with such
//  statements only which do not have any binding to program data.

//  When more complicated statements are used, they have to be
//  compiled and executed separately. The statements can be anonymous,
//  in which case the have to be compiled each time before execution
//  after connection to the database, or they can be named, in which
//  case they can be stored to the database and retrieved later without
//  recompilation. There's also an option that the dbengine generates
//  a unique name for the statement and stores it after the compilation.
//  In later compilations the statement is checked and if it hasn't been
//  changed, the stored version is used instead. The name is generated
//  by calculating the CRC-32 checksum and appending it in hexadeximal
//  string to the first three characters to the statement preceded by
//  an underscore. Although there is a possibility of name clashes
//  in theory the wide value space of CRC-32 should in practice guarantee
//  that they never really happen.

//  Thus, there are four leves of persistence depeneding on whether
//  DoSQL, Compile-Execute, CompStore-Execute or Name-Store-Execute/
//  Name-Restore-Execute technique is used. Notice that storing a
//  statement also makes it executable right away.

IMPULSE     dbeDoSQL        (connection c, string SQLStatement, int strLen);

IMPULSE     dbeCompile      (connection c, string SQLStatement, int strLen);
IMPULSE     dbeCompStore    (connection c, string SQLStatement, int strLen);

IMPULSE     dbeSetName      (connection c, string toThis, int nameLen);
IMPULSE     dbeStoreStmnt   (connection c, string SQLStatement, int strLen);
IMPULSE     dbeRestoreStmnt (connection c);

IMPULSE     dbeExecute      (connection c);

// A Statement can be stored only if it has an unique name and
// be executed only if it has been compiled.

string  ATTRIBUTE dbeName       (connection c);
boolean ATTRIBUTE dbeIsCompiled (connection c);

// ------------------------------------------------------------------
//  The bind-variables may be referred either by numbers or by names,
//  depending on which is the used practice in the SQL statement.
//  When the name is used, it's attached to a number by the attribute
//  dbeBindNumber and the binding is then done the same way as with
```

```
//  a number. All of the variables in a SQL-statement must be either
//  named or numbered. When numbers are used, the numbering
//  starts from 1 and mustn't contain gaps. (That is, if a bind variable
//  is removed from a statement the succeeding bind-variables have to be
//  changed as well to make the numbering contiguous.) When names are
//  used, the returned number must be handled as a macig-cookie, expect
//  for testing the value INVALID_PROGRAM_VARIABLE_NAME.

//  These operations may be performed only if the attached SQLStatement
//  contains bind-variables.

int ATTRIBUTE dbeBindNumber (connection c, string ofName, int nameLen);

IMPULSE dbeSetVar (connection c, int ofNumber, string toThis, int valLen);

IMPULSE dbeSetVarChar (connection c, int ofNumber, char toThis);
IMPULSE dbeSetVarInt  (connection c, int ofNumber, int toThis);
IMPULSE dbeSetVarLong (connection c, int ofNumber, long toThis);

//  The columns of a query are referred as select-buffers.
//  The columns are numbered from 1..dbeNbrofCols. These operations
//  are valid only if the attached SQLStatement begins by 'SELECT...'
//  Otherwise they may have unpredictable results.

int         ATTRIBUTE dbeNbrofCols   (connection c);
SQLDataType ATTRIBUTE dbeColDataType (connection c, int colNbr);
int         ATTRIBUTE dbeColDataLen  (connection c, int colNbr);
string      ATTRIBUTE dbeColName     (connection c, int colNbr);
boolean     ATTRIBUTE dbeColIsNull   (connection c, int colNbr);

string ATTRIBUTE dbeGetCol     (connection c, int colNbr);
char   ATTRIBUTE dbeGetColChar (connection c, int colNbr);
int    ATTRIBUTE dbeGetColInt  (connection c, int colNbr);
long   ATTRIBUTE dbeGetColLong (connection c, int colNbr);

// ---------------------------------------------------------------------
//  Once a select-statement has been executed, a (virtual) table
//  (result set) defined by it is created with a line-pointer.
//  The pointer may be moved back and forth in the table.
//  If no row is pointed becouse of there is none in the table or
//  the pointer is taken beyond the boundaries of the table, the attribute
//  dbePoints returns FALSE. Notice that this is regarded as normal
//  behaviour and thus no error code is returned. Also immediately after
//  execute no row is pointed. The recommended algorithm for iterating
//  through all rows in the table is like this:
//
//      checkStatus (dbeToStart (con));
//      while (dbePoints (con)) {
//          ...handle the row...
//          checkStatus (dbeForth (con));
//      }
//
//  The algorithm works also with tables of zero rows.
//  The checkStatus should react to any error code differing from OK,
//  which all are unexpected and possibly severe errors.

IMPULSE dbeForth   (connection c);
IMPULSE dbeToStart (connection c);
boolean ATTRIBUTE dbePoints (connection c);

#endif
```

15.2. The implementation

```
// ---------------------------------------------------------------------------
//
//  SYSTEM:           OOPoS Proto
//
//  UNIT:             dbEngine
//
//  AUTHOR:           JuM
//
//  $Workfile:   dbengine.c  $
```

```
//  $Revision:   2.9  $ $Date:   31 Jan 1991 07:10:10  $ $Author:   JuM  $
//
// ------------------------------------------------------------------------
//
//  ABSTRACTS:        This is the implementation of the low-level
//                    SQL-database server designed especially for
//                    the high-level object-oriented programming tools
//                    Smalltalk/V PM and Choreographer.
//
//                    Refer to the comments in the header file about how to
//                    use the interface from an application.
//
// ------------------------------------------------------------------------
//
//  $Log:   D:/jumsoft/dbengine/v/dbengine.c  $
//
//                   Rev 2.9   31 Jan 1991 07:10:10   JuM
//               Added the checking of the given cursor handle
//               to each impluse. This should remove most of
//               the traps detected earlier.
//
//                   Rev 2.8   25 Jan 1991 03:21:14   JuM
//               Support for null values added in select.
//
//                   Rev 2.7   24 Jan 1991 04:17:08   JuM
//               DbeColDataLen returns now exactly the space
//               required. Some bugs corrected.
//
//                   Rev 2.6   23 Jan 1991 21:39:14   JuM
//               A new isolation mode (lockNothing) added. Something
//               obsolete taken away from dbeExecute.
//
//                   Rev 2.5   15 Jan 1991 02:10:26   JuM
//               Updates to MS-C 6.0. Uses no longer CRTLIB.
//               New commutils with crc-calculation. Updates
//               to PVCS from CCS.
//
//                   Rev 2.4   10 Jan 1991 23:11:16   JuM
//                    Bugs corrected:
//                      A silly bug corrected in dbeCompStore.
//
//
//                      Copyright (c) 1990,1991 by Nokia Data
//
// ------------------------------------------------------------------------

#include <stdlib.h>
#include <string.h>

// OS/2 is used for concurrency control, no PM is used.

#include <os2.h>

// The SQLBase stuff

#include <sqlsrv.h>
#include <errsql.h>

// The checksum calculation

#include <checksum.h>

// Own spec

#include <dbengine.h>

// ------------------------------------------------------------------
// Cursor characteristics used in the SQLConnect

#define DEF_CURSOR_WORK_SIZE    0       // Use the system default size
#define MAX_CURSOR_WORK_SIZE    0xffff  // For dynamic increase limit
#define DEF_CACHE_PAGES         0       // Use default nbrof cache pages*/
#define USE_RECOVERY            TRUE    // Always enable recovery
#define DEF_OUTSIZE             4096    // Maximal data transfer volume
#define DEF_INSIZE              4096    // sizes.

// Some constants for declarative API-calls
```

```
#define NO_SCALE                0
#define NULL_TERMINATED         0

// The internal error-codes are defined in the header file.
// error codes since they are handled via the same routine. Check this.
// The textual representation of these are found from a proprietary
// dictionary file.

#define internalErrorCode(x) (x & 0x00010000)

// The actual error messages are arranged into an array according to
// the least significant byte of the internal-errorcode. In older
// versions they were read from an external file but that seemed not
// to work in a single-thread DLL (Gpi-trap resulted from using the
// stdio-routines.)

const char* errorDescs [] = {
    "No errors detected.",
    "Run out of memory!",
    "This SQL statement contains nosuch bindvariable!",
    "This SQL statement contains nosuch select expression!",
    "There are unbound program variables in the statement!",
    "Sorry, long data fields are not supported in this version.",
    "Sorry, this feature has not yet been implemented.",
    "The SQL statement has not been compiled!",
    "The SQL statement has not this many bind variables!",
    "The program variable numbering must be contiguous!",
    "No precompiled statement found for this name!",
    "Invalid bind-variable number! Possibly name not found.",
    "Invalid cursor handle. Possibly already disconnected."
};

#define NBROF_INTERNAL_ERRORS 13
#define UNKNOWN_ERROR_CODE  "Unknown error code!"
#define COULDNT_FETCH_ERROR "Some SQL Error: couldn't fetch the description!"

// This is needed by the memory allocator
#define NOSEGMENT       0

// ---------------------------------------------------------------------
// The internal connection table table and associated types.
// Separate dictionaries for both bind variables and select
// buffers are provided.

typedef struct {
    string          biName;
    unsigned char*  biValue;
    SQLDataType     biType;
    unsigned char   biLen;
    unsigned char   strLen;
    unsigned char   biStatus;
} bufferItem, *bufferItemPtr;

typedef bufferItem      bufferArray [SQLMSLI];
typedef bufferArray*    bufferAPtr;

typedef struct {
    SQLTCUR             ulCursor;           // the underlying cursor handle
    char                connId  [128];      // userId and password
    char                name    [SQLMBNL + 1];  // distinctive cursor name
    boolean             compiled;           // contains a compiled statement
    boolean             points;             // points to any row
    int                 maxNbrofBindVars;
    bufferAPtr          bindVars;
    int                 nbrofSelectBufs;
    bufferAPtr          selectBufs;
    char                errorBuffer [SQLMERR];
    status              latestStatus;
    char                workBuffer  [20];
} cursorItem, *cursorItemPtr;

// ---------------------------------------------------------------------
// Global variables; they are local to this module and thus declared
// as "static". Although this module is re-entrant, it's not advisable
// to use the same cursor simulteaneously from many threads. Distinct
// cursor values can, however, be called from concurrent threads.
// The global variables are protected by RAM-semaphores.
```

```
static cursorItemPtr    ConnTB [SQLMICU];
static connection       LastConn    = 0;
static status           GlobStatus  = OK;
static SEL              currentSeg  = 0;
static unsigned         currentOffs;

static unsigned long    lastConnSem   = 0;
static unsigned long    currentSegSem = 0;
static unsigned long    globStatusSem = 0;
#define enterCriticalSection(x) DosSemRequest (&x, SEM_INDEFINITE_WAIT)
#define exitCriticalSection(x)  DosSemClear (&x)

// The pointer structure to the bind- and select buffers is pretty
// complicated. It's made a little bit more readable by these macros

#define ucurs(connection)       ConnTB [connection]->ulCursor
#define cursPtr(connection)     ConnTB [connection]
#define cursField(conn,field)   ConnTB [conn]->field
#define curStatus(cur)          cur->latestStatus
#define connStatus(conn)        ConnTB [conn]->latestStatus
#define column(cursor,nbr)      (*ConnTB [cursor]->selectBufs)[nbr - 1]
#define bindv(cursor,nbr)       (*ConnTB [cursor]->bindVars)[nbr - 1]

// ------------------------------------------------------------------
// The version

string ATTRIBUTE dbeVersion (void)
{
    return "dbEngine, $Revision:   2.9  $ $Date:   31 Jan 1991 07:10:10  $ $Author:
                JuM  $";
}; // dbeVersion

// ------------------------------------------------------------------
// This module has an own memory management, since the malloc seems
// not to work properly. The memory is allocated in chunks by the
// DosAllocSeg-command. The pieces of memory requested are handed
// to the caller and their number is kept in the first word of the
// segment. Each time a piece of memory is deallocated, the counter
// is decremented and when it's zero, the whole segment is freed.
// See functions allocate and deallocate for details.

#define SEGMENT_SIZE    0x1000

void* allocate (unsigned blockSize)
{
    PCH     result;
    PCH     counter;

    enterCriticalSection (currentSegSem);
    if (currentSeg == NOSEGMENT || currentOffs + blockSize > SEGMENT_SIZE) {
        // Must allocate a new segment
        if (DosAllocSeg (SEGMENT_SIZE, &currentSeg, SEG_NONSHARED) == OK) {
            counter = MAKEP (currentSeg, 0);
            *counter = 1;                   // The first piece is used
            currentOffs = sizeof (unsigned); // Leave room for the counter
            result =  MAKEP (currentSeg, currentOffs);
            currentOffs += blockSize;
        } else
            result =  NULLPTR;
    } else {
        // Use the current segment
        counter = MAKEP (currentSeg, 0);
        (*counter)++;
        result = MAKEP (currentSeg, currentOffs);
        currentOffs += blockSize;
    };
    exitCriticalSection (currentSegSem);
    return result;
}; // allocate

void deallocate (void* aBlock)
{
    PCH     counter;
    SEL     seg;

    enterCriticalSection (currentSegSem);
```

```
    // Decrease the count of pieces in the segment, if 0 then deallocate
    counter = MAKEP (SELECTOROF (aBlock), 0);
    if (*counter > 0) {
        (*counter)--;
        if (*counter == 0) {
            seg = SELECTOROF (aBlock);
            DosFreeSeg (seg);
            if (seg == currentSeg)
                currentSeg = NOSEGMENT;
        };
    };
    exitCriticalSection (currentSegSem);
}; // deallocate

// The rounding of allocated space is still used to enable the using of
// the same buffer for many (different length) bind variable values.
static int allocSize (unsigned char ofLen)
{
    int     result;

    if (ofLen <= 16)
        return 16;
    else {
        for (result = 0x100; ((result & ofLen) == 0); result >>= 1);
        if (ofLen > (unsigned char) result)
            result <<= 1;
        return result;
    };
}; // allocSize

// ------------------------------------------------------------------
// The help functions
// ------------------

unsigned ATTRIBUTE stringLength (string str)
{
    return strlen (str);
}; // stringLength

void ATTRIBUTE copyString (string to, string from, int ofLength)
{
    memcpy (to, from, ofLength);
}; // copyString

// ------------------------------------------------------------------
// The error management. Most errors from the internal operations
// are stored inside the cursor record, and an error returned from
// a dbe-operation is also there. Only the constructors save the
// result into a global variable.

#define begin(connection) ConnTB [connection]->latestStatus = OK
#define checkCur(c) if (ConnTB [c] == NULL) return INVALID_CURSOR_HANDLE

LOCALPROC setGlobalStatus (status toThis)
{
    enterCriticalSection (globStatusSem);
    if (GlobStatus == OK) GlobStatus = toThis;
    exitCriticalSection (globStatusSem);
};

status ATTRIBUTE dbeConnectionStatus (connection c)
{
    if (c == NO_CONNECTION || cursPtr (c) == NULLPTR)
        return GlobStatus;
    else
        return connStatus (c);
};
LOCALPROC fetchOwnErrorDecl (status errorCode, string toBuf)
{
    int     errIx = (short) errorCode;

    if (errIx < NBROF_INTERNAL_ERRORS)
        strcpy (toBuf, errorDescs [errIx]);
    else
        strcpy (toBuf, UNKNOWN_ERROR_CODE);
}; // fetchOwnErrorDecl;
```

```
string ATTRIBUTE dbeErrorDescription (connection c)
{
    if (c != NO_CONNECTION) {
        if (!internalErrorCode (connStatus (c)))
            sqlerr ((short) connStatus (c), cursField (c, errorBuffer));
        else
            fetchOwnErrorDecl (connStatus (c), cursField (c, errorBuffer));
    } else {
        if (!internalErrorCode (GlobStatus))
            return (COULDNT_FETCH_ERROR);
        else
            return ((string) errorDescs [(short) GlobStatus]);
    };
    return cursField (c, errorBuffer);
}; // dbeErrorDescription;

LOCALPROC signalError (cursorItemPtr cur, status errorCode)
{
    cur->latestStatus = errorCode;
    if (errorCode == MEMORY_EXHAUSTED)
        setGlobalStatus (errorCode);
}; // signalError

// ------------------------------------------------------------------
// The general data-buffer management. These routines are used buy both
// select-buffer and bind-variable management.

  // This procedure is called when a buffer item is allocated or
  // attached to data the first time
LOCALPROC setupBufferItem (
    cursorItemPtr   toCursor,
    bufferItemPtr   theBufItem,
    string          name,
    unsigned char   dataLength,
    SQLDataType     dataType
) {
    theBufItem->biType = dataType;
    if ((theBufItem->biName = allocate (strlen (name) + 1)) == NULL)
        signalError (toCursor, MEMORY_EXHAUSTED);
    else {
        strcpy (theBufItem->biName, name);
        if ((theBufItem->biValue = allocate (dataLength)) == NULL)
            signalError (toCursor, MEMORY_EXHAUSTED);
        else
            theBufItem->biLen = dataLength;
    };
}; // setupBufferItem

LOCALPROC setBufferValue (
    cursorItemPtr   cur,
    bufferItemPtr   toBuf,
    unsigned char*  toThis,
    unsigned char   ofLen
) {
    if (toBuf->biValue == NULL ||
        allocSize (toBuf->biLen) != allocSize (ofLen))
    {
        // if the new data is larger or very much smaller than already
        // allocated space deallocate old and allocate new area
        if (toBuf->biLen > 0)
            deallocate (toBuf->biValue);
        if ((toBuf->biValue = allocate (allocSize (ofLen))) == NULL) {
            signalError (cur, MEMORY_EXHAUSTED);
            return;
        };
    };
    toBuf->biLen = ofLen;
    memcpy (toBuf->biValue, toThis, ofLen);
}; // setBufferValue

LOCALPROC setBufferName (
    cursorItemPtr   cur,
    bufferItemPtr   toBuf,
    unsigned char*  toThis,
    unsigned char   ofLen
) {
    if (toBuf->biName == NULL)
```

```
        if ((toBuf->biName = allocate (SQLMBNL + 1)) == NULL) {
            signalError (cur, MEMORY_EXHAUSTED);
            return;
        };
    memset (toBuf->biName, '\0', SQLMBNL + 1);
    strncpy (toBuf->biName, toThis, ofLen);
}; // setBufferName

LOCALPROC clearBufferItem (bufferItemPtr theBufItem)
{
    theBufItem->biValue = NULL;
    theBufItem->biLen   = 0;
    theBufItem->strLen  = 0;
    theBufItem->biName  = NULL;
}; // clearBufferItem

LOCALPROC releaseBufferItem (bufferItemPtr theBufItem)
{
    if (theBufItem-> biName != NULLPTR) {
        deallocate (theBufItem-> biName);
        theBufItem-> biName = NULLPTR;
    };
    if (theBufItem-> biValue != NULLPTR) {
        deallocate (theBufItem->biValue);
        theBufItem-> biValue = NULLPTR;
    };
}; // releaseBufferItem

// ------------------------------------------------------------------
// The select buffer management. Data is transferred from the database
// to the program via these. They are allocated after compilation and
// bound immediately. After each fetch the data is automatically
// transferred to them by SQLBase.

LOCALPROC allocSelectBuf (
    cursorItemPtr   toCursor,
    unsigned char   ofNumber,
    string          ofName,
    unsigned char   ofLength,
    SQLDataType     ofType
) {
    setupBufferItem (
        toCursor,
        &((*toCursor->selectBufs) [ofNumber - 1]),
        ofName,
        ofLength,
        ofType
    );
}; // allocSelectBuf

LOCALPROC allocSelectBuffers (cursorItemPtr ofCursor, int nbrofColumns)
{
    ofCursor->selectBufs = allocate (nbrofColumns * sizeof (bufferItem));
    if (ofCursor->selectBufs == NULL)
        signalError (ofCursor, MEMORY_EXHAUSTED);
    else
        memset (ofCursor->selectBufs, '\0',
            nbrofColumns * sizeof (bufferItem));
    ofCursor->nbrofSelectBufs = nbrofColumns;
}; // allocSelectBuffers

// The following routines perform the data conversions between
// program data types and SQLBase's own internal types.
// See the header file of comments on the algorithm. It's mainline is
// to transfer any data converted into strings to avoid problems with
// proprietary internal formats of SQLBase and Smalltalk. There is
// a serious drawback, however: almost all of the numeric data is
// transferred by converting it back and forth to the ASCII-representation.
// If efficiency turns out to be a problem (as it always does), an effort
// of providing means for transferring them directly in the internal
// representation of SQLBase should be taken to serious consideration.

static SQLDataType infereSuitableDataType (
    cursorItemPtr   cur,
    unsigned char   dbDataType,
    unsigned char   dataLength,
    unsigned char   precision,
```

```
    unsigned char   scale,
    unsigned char   *strLen     // out: the max length as a string
) {
    switch (dbDataType) {

        case SQLDCHR:
            *strLen = dataLength;
            if (dataLength == 1)
                return CHARACTER;
            else
                return STRING;
            break;

        case SQLDNUM:
            if (scale == 0) { // this is an integer
                *strLen = precision;
                if (precision == 0) { // SQLBase couldn't decide
                    *strLen = 8;
                    return REAL;
                } else if (precision <= 4)
                    return SHORT_INTEGER;
                else if (precision <= 9)
                    return LONG_INTEGER;
                else
                    return VERY_LONG_INTEGER;
            } else {
                // All decimals etc. are regarded as reals
                // one pos for dec point
                *strLen = (unsigned char) (precision - scale +  1);
                return REAL;
            };

        case SQLDDAT: {
            *strLen = 20; // Like in: "17-Mar-1963 10:05:00"
            return TIMEPOINT;
        };

        case SQLDDTE: {
            *strLen = 11;
            return DATE;
        };

        case SQLDTIM: {
            *strLen = 8;
            return TIME;
        };

        case SQLDLON:
            signalError (cur, LONG_DATA_NOT_SUPPORTED);

        default: {
            *strLen = dataLength;
            return STRING;
        };

    }; // switch
}; // infereSuitableDataType

LOCALPROC setupOneSelectBuf (cursorItemPtr cur, unsigned char colNbr)
{
    short           result;
    unsigned char   dataType, dataLen, nameLen, strLen, prec, scale;
    char            pDataType;
    SQLDataType     internalType;
    char            colName [128];

    // Get the accurate description of the column in question
    result = sqldes (
        cur->ulCursor,
        colNbr,
        &dataType,
        &dataLen,
        colName,
        &nameLen,
        &prec,
        &scale
    );
```

```
    colName [nameLen] = '\0';
    if (result == OK) {
        // ...and set up a suitable buffer for it
        internalType = infereSuitableDataType (cur, dataType, dataLen,
            prec, scale, &strLen);
        (*cur->selectBufs) [colNbr - 1]. strLen = strLen;
        // decide the actual transfer type and transfer buffer size
        switch (internalType) {
            case CHARACTER:
                pDataType = SQLPSCH;
                dataLen = sizeof (char);
                break;
            case SHORT_INTEGER:
                pDataType = SQLPSIN;
                dataLen = sizeof (int);
                break;
            case LONG_INTEGER:
                pDataType = SQLPSLO;
                dataLen = sizeof (long);
                break;
            default:
                pDataType = SQLPSTR;
                dataLen++;  // Make room for the terminating null
                break;
        }; // switch

        allocSelectBuf (cur, colNbr, colName, dataLen, internalType);
        // Setup the select buffer
        result = sqlssb (
            cur->ulCursor,
            colNbr,
            pDataType,
            (*cur->selectBufs) [colNbr - 1]. biValue,
            dataLen,
            NO_SCALE,
            SQLNPTR,
            &((*cur->selectBufs) [colNbr - 1]. biStatus)
        );
        if (result != OK)
            signalError (cur, (status) result);
    } else
        signalError (cur, (status) result);
}; // setupOneSelectBuf

LOCALPROC setupSelectBufs (cursorItemPtr cur)
{
    short           result;
    unsigned char   nsi, loop;

    result = sqlnsi (cur->ulCursor, &nsi);
    if (result == OK) {
        allocSelectBuffers (cur, nsi);
        for (loop = 1; loop <= nsi; loop++) {
            setupOneSelectBuf (cur, loop);
            if (curStatus (cur) != OK) break; // some error deteceted
        };
    } else
        signalError (cur, (status) result);
}; // setupSelectBufs

// ------------------------------------------------------------------
//  Bind-buffer management. The bind buffers are allocated partly
//  after compilation and partly when they are set the first time.
//  Each bind variable has to be set before 'execute' is performed.
//  In each 'execute' the data currently in the bind-buffers is
//  transferred and converted automatically by the SQLBase.

  // The main array is allocated after the compilation...
LOCALPROC setupBindVars (cursorItemPtr ofCursor, int thisMany)
{
    int     loop;

    if (thisMany > 0) {
        ofCursor->bindVars = allocate (thisMany * sizeof (bufferItem));
        if (ofCursor->bindVars == NULL)
            signalError (ofCursor, MEMORY_EXHAUSTED);
        else
```

```
            for (loop = 1; loop <= thisMany; loop++)
                clearBufferItem (&(*ofCursor->bindVars) [loop - 1]);
    } else
        ofCursor-> bindVars = NULLPTR;
    ofCursor->maxNbrofBindVars = thisMany;
}; // setupBindVars

  // ...and each buffer when it's set the first time
static bufferItemPtr allocBindVar (
    cursorItemPtr   toCursor,
    int             ofNumber,
    SQLDataType     ofType,
    unsigned char*  withData,
    unsigned char   ofSize
) {
    bufferItemPtr   theBuf;

    // Add the new item to last
    if (toCursor->maxNbrofBindVars >= ofNumber) {
        theBuf = &((*toCursor->bindVars) [ofNumber - 1]);
        theBuf->biType = ofType;
        setBufferValue (toCursor, theBuf, withData, ofSize);
        return theBuf;
    } else {
        signalError (toCursor, UNCONTIGUOUS_BIND_NUMBERING);
        return NULL;
    };
}; // allocBindVar

LOCALPROC setBindValue (
    cursorItemPtr   cur,
    int             ofNumber,
    SQLDataType     ofType,
    unsigned char*  fromThis,
    unsigned char   ofBytes
) {
    bufferItemPtr   bindVar;
    short           result;
    char            pDataType;

    if (ofNumber == INVALID_PROGRAM_VARIABLE_NAME) {
        signalError (cur, UNKNOWN_BINDVAR_NAME);
        return;
    };

    // allocate the buffer if not yet done and move the data to it
    bindVar = allocBindVar (cur, ofNumber, ofType, fromThis, ofBytes);
    if (curStatus(cur) != OK) return;

    // decide the type and inform the database
    switch (ofType) {
        case CHARACTER:
            pDataType = SQLPUCH;
            break;
        case SHORT_INTEGER:
            pDataType = SQLPSIN;
            break;
        case LONG_INTEGER:
            pDataType = SQLPSLO;
            break;
        default:
            pDataType = SQLPBUF;
            break;
    }; // switch

    if (bindVar->biName == NULL)
        // Bind by the number
        result = sqlbnn (
            cur->ulCursor,
            (unsigned char) ofNumber,
            bindVar->biValue,
            bindVar->biLen,
            NO_SCALE,
            pDataType
        );
    else
        // Bind by the name
```

```
        result = sqlbnd (
            cur->ulCursor,
            bindVar->biName,
            NULL_TERMINATED,
            bindVar->biValue,
            bindVar->biLen,
            NO_SCALE,
            pDataType
        );
    if (result != OK)
        signalError (cur, (status) result);
}; // setBindValue

// ------------------------------------------------------------------
// Some cursor management

LOCALPROC initCursor (cursorItemPtr cur)
{
    cur->maxNbrofBindVars = 0;
    cur->bindVars = NULL;
    cur->nbrofSelectBufs = 0;
    cur->selectBufs = NULL;
    cur->compiled = FALSE;
    cur->points = FALSE;
    cur->connId [0] = '\0';
    cur->name [0] = '\0';
    cur->errorBuffer [0] = '\0';
    cur->workBuffer [0] = '\0';
    cur->latestStatus = OK;
}; // initCursor

LOCALPROC clearCursor (cursorItemPtr cur)
{
    int     loop;

    for (loop = 1; loop <= cur->maxNbrofBindVars; loop++)
        releaseBufferItem (&(*cur->bindVars) [loop - 1]);
    if (cur-> bindVars != NULLPTR) {
        deallocate (cur-> bindVars);
        cur-> bindVars = NULLPTR;
    };
    cur->maxNbrofBindVars = 0;
    for (loop = 1; loop <= cur->nbrofSelectBufs; loop++)
        releaseBufferItem (&(*cur->selectBufs) [loop - 1]);
    if (cur-> selectBufs != NULLPTR) {
        deallocate (cur-> selectBufs);
        cur-> selectBufs = NULLPTR;
    };
    cur->nbrofSelectBufs = 0;
}; // clearCursor

// ------------------------------------------------------------------
// Connection and transaction management.

LOCALPROC tryToConnect (cursorItemPtr byThis)
{
    short result;

    result = sqlcon (
        &byThis->ulCursor,
        byThis->connId,
        strlen (byThis->connId),
        DEF_CURSOR_WORK_SIZE,
        DEF_CACHE_PAGES,
        USE_RECOVERY,
        DEF_OUTSIZE,
        DEF_INSIZE
    );
    if (result == OK)
        LastConn++;
    else
        signalError (byThis, (status) result);
}; // tryToConnect

static connection newConnection (string theName)
{
    connection  newConn;
```

```
    enterCriticalSection (lastConnSem);
    newConn = LastConn;
    if ((ConnTB [newConn] = allocate (sizeof (cursorItem))) != NULL) {
        initCursor (ConnTB [newConn]);
        strcpy (ConnTB [newConn]->connId, theName);
        tryToConnect (ConnTB [newConn]);
        exitCriticalSection (lastConnSem);
        return newConn;
    } else {
        setGlobalStatus (MEMORY_EXHAUSTED);
        exitCriticalSection (lastConnSem);
        return NO_CONNECTION;
    };
}; // newConnection

CONSTRUCTOR dbeConnect
(
    string  dbName,
    int     dbNameLen,
    string  userId,
    int     userIdLen,
    string  password,
    int     passwordLen
) {
    char        combinedName [256];

    strncpy (combinedName, dbName, dbNameLen);
    combinedName [dbNameLen] = '\0';
    if (userIdLen > 0 || passwordLen > 0) {
        strcat (combinedName, "/");
        strncat (combinedName, userId, userIdLen);
        strcat (combinedName, "/");
        strncat (combinedName, password, passwordLen);
    };
    return newConnection (combinedName);
}; // dbeConnect

CONSTRUCTOR dbeDuplicate (connection c)
{
    begin (c);
    return newConnection (ConnTB [c]->connId);
}; // dbeDuplicate

IMPULSE dbeDisconnect (connection c)
{
    short result;

    checkCur(c);
    begin (c);
    result = sqldis (ucurs(c));
    if (result != OK)
        signalError (cursPtr (c), (status) result);
    else {
        clearCursor (ConnTB [c]);
        deallocate (ConnTB [c]);
        ConnTB [c] = NULL;
        setGlobalStatus ((status) OK);
        // Decrease the first-free ptr to avoid running out of handles
        enterCriticalSection (lastConnSem);
        if (c == LastConn - 1)
            for (; ConnTB [LastConn - 1] == NULL; LastConn--);
        exitCriticalSection (lastConnSem);
    };
    return (status) result;
}; // dbeDisconnect

boolean ATTRIBUTE dbeIsConnected (connection c)
{
    return (c < LastConn && ConnTB [c] != NULL);
};

// ------------------------------------------------------------------
// The transaction management.

IMPULSE     dbeCommit (connection c)
{
```

```
    short   result;

    begin (c);
    result = sqlcmt (ucurs(c));
    if (result != OK)
        signalError (cursPtr (c), (status) result);
    return connStatus (c);
}; // dbeCommit

IMPULSE     dbeRollback (connection c)
{
    short   result;

    begin (c);
    result = sqlrbk (ucurs(c));
    if (result != OK)
        signalError (cursPtr (c), (status) result);
    return connStatus (c);
}; // dbeRollback

// ------------------------------------------------------------------
// The locking scheme

IMPULSE dbeSetIsolationLevel (connection c, lockingMode m)
{
    short   result;

    checkCur (c);
    begin (c);
    switch (m) {
        case readOnly: {
            result = sqlsil (ucurs(c), SQLILRO);
            break;
        };
        case lockNothing: {
            result = sqlsil (ucurs(c), SQLILRL);
            break;
        };
        case lockCurrent: {
            result = sqlsil (ucurs(c), SQLILCS);
            break;
        };
        case lockAll: {
            result = sqlsil (ucurs(c), SQLILRR);
            break;
        };
    };
    if (result != OK)
        signalError (cursPtr (c), (status) result);
    return connStatus (c);
}; // dbeSetIsolationLevel

// ------------------------------------------------------------------
//  SQL-statement management.

IMPULSE dbeDoSQL (connection c, string SQLStatement, int strLen)
{
    // This procedure is performed by help of the other dbe-level routines
    // to get advantage of their possibly better error management than
    // what the sqlcex-routine could offer.

    checkCur (c);
    dbeCompile (c, SQLStatement, strLen);
    if (connStatus (c) == OK) {
        dbeExecute (c);
        if (connStatus (c) == OK) {
            dbeCommit (c);
        };
    };
    return connStatus (c);
}; // dbeDoSQL

// ------------------------------------------------------------------

LOCALPROC setUpCompiledStatement (connection c)
{
    short           result;
```

```
    char            commandType;
    unsigned char   nbv;

    if (cursField (c, compiled)) // Clean away the old buffers
        clearCursor (cursPtr (c));
    cursField (c, compiled) = TRUE;

    // Allocate and initialize the bind- and select buffers
    result = sqlnbv (ucurs(c), &nbv);
    if (result == OK)
        setupBindVars (cursPtr (c), nbv);
    else {
        signalError (cursPtr (c), (status) result);
        return;
    };

    result = sqlcty (ucurs(c), &commandType);
    if (result == OK) {
        if (commandType == SQLTSEL)
            setupSelectBufs (cursPtr (c));
    } else
        signalError (cursPtr (c), (status) result);
}; // setUpCompiledStatement

IMPULSE dbeCompile (connection c, string SQLStatement, int strLen)
{
    short           result;

    checkCur (c);
    begin (c);
    // Compile in the new statement
    result = sqlcom (ucurs(c), SQLStatement, strLen);
    if (result == OK)
        setUpCompiledStatement (c);
    else
        signalError (cursPtr (c), (status) result);
    return connStatus (c);
}; // dbeCompile

LOCALPROC generateName (string SQLStatement, int strLen, string result)
{
    int     starts;
    char    help [12];     // for instance _sel_6bf30a91

    strcpy (result, "_");
    for (starts = 0; SQLStatement [starts] == ' '; starts++);
    strncat (result, &SQLStatement [starts], 3);
    strcat (result, "_");
    ultoa (crc32 (SQLStatement, (long) strLen), help, 0x10);
    strcat (result, help);
}; // generateName

IMPULSE dbeCompStore (connection c, string SQLStatement, int strLen)
{
    char        genName [SQLMBNL];
    short       result;

    checkCur (c);
    generateName (SQLStatement, strLen, genName);
    // Try to restore the statement first
    if (sqlret (ucurs(c), genName, NULL_TERMINATED) == OK)
        setUpCompiledStatement (c);
    else {
        // Hasn't been compiled before or is invalidated by changes
        // in the database structure. Store and restore the statement.
        begin (c);
        result = sqlsto (ucurs(c), genName, NULL_TERMINATED,
            SQLStatement, strLen);
        if (result == OK) {
            result = sqlret (ucurs(c), genName, NULL_TERMINATED);
            if (result == OK)
                setUpCompiledStatement (c);
            else if (result != DBOECNF)
                signalError (cursPtr (c), (status) result);
        } else
            signalError (cursPtr (c), (status) result);
    };
```

```
    return connStatus (c);
}; // dbeCompStore

LOCALPROC checkBindings (connection c)
{
    int         loop;

    for (loop = 1; loop <= cursField (c, maxNbrofBindVars); loop++) {
        if (bindv (c,loop). biValue == NULL)
            signalError (cursPtr (c), UNBOUND_PROGRAM_VARIABLES);
    };
}; // checkBindings

IMPULSE dbeExecute (connection c)
{
    short   result;

    checkCur (c);
    begin (c);
    if (dbeIsCompiled (c)) {
        checkBindings (c);
        if (connStatus (c) == OK) {
            result = sqlexe (ucurs(c));
            if (result != OK)
                signalError (cursPtr (c), (status) result);
            else
                cursField (c, points) = FALSE;
        };
    } else
        signalError (cursPtr (c), STATEMENT_NOT_COMPILED);
    return connStatus (c);
}; // dbeExecute

boolean ATTRIBUTE dbeIsCompiled (connection c)
{
    return cursField (c, compiled);
}; // dbeIsCompiled

IMPULSE dbeSetName (connection c, string toThis, int nameLen)
{
    int     safeLen = min(nameLen, sizeof (cursField (c, name)) - 1);
    short   result;

    checkCur (c);
    begin (c);
    strncpy (cursField (c, name), toThis, safeLen);
    cursField (c, name) [safeLen] = '\0';
    result = sqlscn (ucurs (c), cursField (c, name), NULL_TERMINATED);
    if (result != OK)
        signalError (cursPtr (c), (status) result);
    return connStatus (c);
}; // dbeSetName

string ATTRIBUTE dbeName (connection c)
{
    return cursField (c, name);
}; // dbeName

IMPULSE dbeStoreStmnt (connection c, string SQLStatement, int strLen)
{
    short       result;

    checkCur (c);
    begin (c);
    result = sqlsto (ucurs(c), cursField (c, name), NULL_TERMINATED,
        SQLStatement, strLen);
    if (result == OK)
        return dbeRestoreStmnt (c);
    else
        signalError (cursPtr (c), (status) result);
    return connStatus (c);
}; // dbeStoreStmnt

IMPULSE dbeRestoreStmnt (connection c)
{
    short       result;
```

```
    checkCur (c);
    begin (c);
    result = sqlret (ucurs(c), cursField (c, name), NULL_TERMINATED);
    if (result == OK)
        setUpCompiledStatement (c);
    else if (result == DBOECNF)
        signalError (cursPtr (c), STATEMENT_NOT_FOUND);
    else
        signalError (cursPtr (c), (status) result);
    return connStatus (c);
}; // dbeRestoreStmnt

// ------------------------------------------------------------------
// Bind-variable management.

// This function returns a number of the place for named bind data.
// It's either the first free place or the one already allocated for
// that name.

int ATTRIBUTE dbeBindNumber (connection c, string ofName, int nameLen)
{
    int             loop, safeLen;
    char            locName [SQLMBNL + 1];

    safeLen = min (nameLen, sizeof (locName));
    memset (locName, '\0', sizeof (locName));
    // case insensitive comparison
    for (loop = 0; loop < safeLen; loop++)
        locName [loop] = (char) toupper (ofName [loop]);
    loop = 1;
    do {
        if (bindv (c, loop). biName == NULL) {
            // a free place located
            setBufferName (cursPtr (c), &bindv (c, loop), locName,
                (unsigned char) safeLen);
            return loop;
        } else {
            if (strcmp (bindv (c, loop). biName, locName) == 0)
                // found the correct name!
                return (int) loop;
            else {
                if (loop == cursField (c, maxNbrofBindVars)) {
                    // All places are already named and no match was found
                    signalError (cursPtr (c), UNKNOWN_BINDVAR_NAME);
                    return INVALID_PROGRAM_VARIABLE_NAME;
                } else
                    loop++;
            }; // else
        }; // else
    } while (TRUE); // do
}; // dbeBindNumber

IMPULSE dbeSetVar (connection c, int ofNumber, string toThis, int valLen)
{
    checkCur (c);
    begin (c);
    setBindValue (cursPtr (c), ofNumber, STRING,
        (unsigned char*) toThis, (unsigned char) valLen);
    return connStatus (c);
}; // dbeSetVar

IMPULSE dbeSetVarChar (connection c, int ofNumber, char toThis)
{
    char        local;

    checkCur (c);
    begin (c);
    local = toThis;
    setBindValue (cursPtr (c), ofNumber, CHARACTER,
        (unsigned char*) &local, (unsigned char) sizeof (char));
    return connStatus (c);
}; // dbeSetVarChar

IMPULSE dbeSetVarInt  (connection c, int ofNumber, int toThis)
{
    int         local;
```

```
    checkCur (c);
    begin (c);
    local = toThis;
    setBindValue (cursPtr (c), ofNumber, SHORT_INTEGER,
        (unsigned char*) &local, (unsigned char) sizeof (int));
    return connStatus (c);
}; // dbeSetVarInt

IMPULSE dbeSetVarLong (connection c, int ofNumber, long toThis)
{
    long        local;

    checkCur (c);
    begin (c);
    local = toThis;
    setBindValue (cursPtr (c), ofNumber, LONG_INTEGER,
        (unsigned char*) &local, (unsigned char) sizeof (long));
    return connStatus (c);
}; // dbeSetVarLong

// ------------------------------------------------------------------
//  Select-buffer (column) management

int ATTRIBUTE dbeNbrofCols (connection c)
{
    return cursField (c, nbrofSelectBufs);
}; // dbeNbrofCols

SQLDataType ATTRIBUTE dbeColDataType (connection c, int colNbr)
{
    return column (c, colNbr). biType;
}; // dbeColDataType

int ATTRIBUTE dbeColDataLen  (connection c, int colNbr)
{
    return column (c, colNbr). strLen;
}; // dbeColDataLen

string ATTRIBUTE dbeColName (connection c, int colNbr)
{
    return column (c, colNbr). biName;
}; // dbeColName

boolean ATTRIBUTE dbeColIsNull (connection c, int colNbr)
{
    unsigned char   valLen;         // If this is 0, data is null
    char            fetchStatus;    // This is received already
    short           retCode;

    if (column (c, colNbr). biStatus == OK) {
        retCode = sqlgfi (ucurs(c), (unsigned char) colNbr,
            &valLen, &fetchStatus);
        if (retCode == OK)
            return (valLen == 0);
        else {
            signalError (cursPtr (c), (status) retCode);
            return TRUE;
        };
    } else
        return TRUE;
}; // dbeColIsNull

string ATTRIBUTE dbeGetCol (connection c, int colNbr)
{
    switch (column (c, colNbr). biType) {

        default:
            return (column (c, colNbr). biValue);

        case CHARACTER:
            cursField (c, workBuffer) [0] = dbeGetColChar (c, colNbr);
            cursField (c, workBuffer) [1] = '\0';
            return cursField (c, workBuffer);

        // When binary integers have been used to boost the transfer
        // and ascii representation is still used the conversion to string
        // must be done here
```

```
        case SHORT_INTEGER:
            itoa ((int) *(column (c, colNbr). biValue),
                cursField (c, workBuffer), 10);
            return cursField (c, workBuffer);

        case LONG_INTEGER:
            ltoa ((status) *(column (c, colNbr). biValue),
                cursField (c, workBuffer), 10);
            return cursField (c, workBuffer);
    };
}; // dbeGetCol

char ATTRIBUTE dbeGetColChar (connection c, int colNbr)
{
    char    result;
    if (column (c, colNbr). biType == CHARACTER) {
        result = (char) *(column (c, colNbr). biValue);
        // This is one of the peculiarties of SQLBase. Digits, when
        // saved into char(1)-fields, are returned from range 0-9
        // rather than from '0'-'9'!
        if (result < ' ')
            result += '0';
        return result;
    } else
        return '\0';
}; // dbeGetColChar

int ATTRIBUTE dbeGetColInt  (connection c, int colNbr)
{
    if (column (c, colNbr). biType == SHORT_INTEGER ||
        column (c, colNbr). biType == LONG_INTEGER ||
        column (c, colNbr). biType == CHARACTER
    )
        return (int) *(column (c, colNbr). biValue);
    else
        return atoi (dbeGetCol (c, colNbr));
}; // dbeGetColInt

long ATTRIBUTE dbeGetColLong (connection c, int colNbr)
{
    if (column (c, colNbr). biType == SHORT_INTEGER ||
        column (c, colNbr). biType == LONG_INTEGER
    )
        return (long) *(column (c, colNbr). biValue);
    else
        return atol (dbeGetCol (c, colNbr));
}; // dbeGetColLong

// ------------------------------------------------------------------------
// Traversal in the result set

IMPULSE dbeForth (connection c)
{
    short   result;

    checkCur (c);
    begin (c);
    result = sqlfet (ucurs(c));
    if (result == OK)
        cursField (c, points) = TRUE;
    else {
        cursField (c, points) = FALSE;
        if (result != FETEEOF)
            signalError (cursPtr (c), (status) result);
    };
    return connStatus (c);
}; // dbeForth

IMPULSE dbeToStart (connection c)
{
    checkCur (c);
    begin (c);
    if (cursField (c, compiled))
        // Can be executed
        if (cursField (c, points))
            // Has been progressed already. Set back to the start
            // by re-executing the statement.
```

```
            dbeExecute (c);
    if (connStatus (c) == OK)
        return dbeForth (c);
    else
        return (connStatus (c));
}; // dbeToStart

boolean ATTRIBUTE dbePoints (connection c)
{
    return cursField (c, points);
}; // dbePoints
```

15.3. Test programs

```
// ------------------------------------------------------------------------
//
//  SYSTEM:           OOPoS Proto
//
//  UNIT:             dbEngine
//
//  AUTHOR:           $Author:   JuM  $
//
//  $Workfile:   dbetest1.c  $
//  $Revision:   2.4  $ $Date:   10 Jan 1991 23:50:08  $ $Author:   JuM  $
//
// ------------------------------------------------------------------------
//
// ABSTRACTS: Connecting to the database and performing isolated SQL-
//  statements. Do not let the "own"-routines fool You, they are just
//  shorthand notation to strip off a few parameters (mainly the string
//  length).
//
// ------------------------------------------------------------------------
//
//  $Log:   D:/jumsoft/dbengine/v/dbetest1.c  $
//
//                 Rev 2.4   10 Jan 1991 23:50:08   JuM
//              Initial revision.
//
//
//                    Copyright (c) 1990,1991 by Nokia Data
//
// ------------------------------------------------------------------------

#include <stdio.h>
#include <string.h>

#include "dbengine.h"

static void checkErc (connection c)
{
    if (dbeConnectionStatus (c) != OK) {
        printf ("SQLERROR %s\n", dbeErrorDescription (c));
    };
};

CONSTRUCTOR myConnect (string toDb)
{
    return dbeConnect (toDb, strlen (toDb), "", 0, "", 0);
};

IMPULSE myDoSQL (connection cur, string theStatement)
{
    return dbeDoSQL (cur, theStatement, strlen (theStatement));
};

int main (void)
{
    connection  myCur;

    checkErc (myCur = myConnect ("JOBS"));
    myDoSQL (myCur,
        "INSERT INTO ACCOUNT (WDATE, TASK, WMIN, INFO) VALUES \
```

```
            ('12-Oct-90', 'EVE', 240, 'Making dbEngine re-entrant')");
        checkErc (myCur);
        dbeDisconnect (myCur);
        return 0;
};
```

```
// ------------------------------------------------------------------------
//
//  SYSTEM:           OOPoS Proto
//
//  UNIT:             dbEngine
//
//  AUTHOR:           $Author:   JuM  $
//
//  $Workfile:   dbetest2.c  $
//  $Revision:   2.5  $ $Date:   25 Jan 1991 03:22:28  $ $Author:   JuM  $
//
// ------------------------------------------------------------------------
//
// ABSTRACTS: Selecting items from the database and fetching them to
// the application.
//
// ------------------------------------------------------------------------
//
//  $Log:   D:/jumsoft/dbengine/v/dbetest2.c  $
//
//                Rev 2.5   25 Jan 1991 03:22:28   JuM
//             Changed from printing Accounts to Tasks
//             and added explicit notes of null-values.
//
//                Rev 2.4   10 Jan 1991 23:50:10   JuM
//             Initial revision.
//
//
//                   Copyright (c) 1990,1991 by Nokia Data
//
// ------------------------------------------------------------------------

#include <stdio.h>
#include <string.h>

#include "dbengine.h"

static void checkErc (connection c)
{
    if (dbeConnectionStatus (c) != OK) {
        printf ("SQLERROR %s\n", dbeErrorDescription (c));
    };
};

CONSTRUCTOR myConnect (string toDb)
{
    return dbeConnect (toDb, strlen (toDb), "", 0, "", 0);
};

IMPULSE myCompile (connection cur, string theStatement)
{
    return dbeCompile (cur, theStatement, strlen (theStatement));
};

// Output functions

static void printHeading (connection cur)
{
    char        ownBuf [256];
    int         loop, i, nbrofCols, first = TRUE;

    nbrofCols = dbeNbrofCols (cur);
    for (loop = 1; loop <= nbrofCols; loop++) {
        if (first) first = FALSE; else printf (" ");
        strcpy (ownBuf, dbeColName (cur, loop));
        printf (ownBuf);
        for (i = 0; i < (dbeColDataLen (cur, loop) - strlen (ownBuf)); i++)
            printf (" ");
    };
    printf ("\n-------------------------------------------------------------------------
                --------\n");
};

static void printTuple (connection cur)
{
    char        ownBuf [256];
    int         loop, i, nbrofCols, len, first = TRUE;
```

```
    nbrofCols = dbeNbrofCols (cur);
    for (loop = 1; loop <= nbrofCols; loop++) {
        if (first) first = FALSE; else printf (" ");
        len = dbeColDataLen (cur, loop);
        if (dbeColIsNull (cur, loop))
            strcpy (ownBuf, "NULL");
        else {
            strncpy (ownBuf, dbeGetCol (cur, loop), len);
            ownBuf [len] = '\0';
        };
        printf ("%s", ownBuf);
        for (i = 0; i < (len - strlen (ownBuf)); i++)
            printf (" ");
    };
    printf ("\n");
};

static void printTasks (connection cur)
{
    myCompile (cur,
        "SELECT * FROM TASKS");
    checkErc (cur);
    dbeExecute (cur);
    checkErc (cur);
    printHeading (cur);
    dbeToStart (cur);
    checkErc (cur);
    while (dbePoints (cur)) {
        printTuple (cur);
        dbeForth (cur);
        checkErc (cur);
    };
};

int main (void)
{
    connection  myCur;

    checkErc (myCur = myConnect ("JOBS"));
    printTasks (myCur);
    dbeDisconnect (myCur);
    return 0;
};
```

```
// ------------------------------------------------------------------------
//
//  SYSTEM:           OOPoS Proto
//
//  UNIT:             dbEngine
//
//  AUTHOR:           $Author:   JuM  $
//
//  $Workfile:   dbetest3.c  $
//  $Revision:   2.4  $ $Date:   10 Jan 1991 23:50:12  $ $Author:   JuM  $
//
// ------------------------------------------------------------------------
//
// ABSTRACTS: Selecting items interactively from the database according
//  to a bind variable.
//
// ------------------------------------------------------------------------
//
//  $Log:   D:/jumsoft/dbengine/v/dbetest3.c  $
//
//                Rev 2.4   10 Jan 1991 23:50:12   JuM
//              Initial revision.
//
//
//                    Copyright (c) 1990,1991 by Nokia Data
//
// ------------------------------------------------------------------------

#include <stdio.h>
#include <string.h>

#include "dbengine.h"

static boolean checkErc (connection c)
{
    if (dbeConnectionStatus (c) != OK) {
        printf ("SQLERROR %s\n", dbeErrorDescription (c));
        return FALSE;
    } else
        return TRUE;
};

CONSTRUCTOR myConnect (string toDb)
{
    return dbeConnect (toDb, strlen (toDb), "", 0, "", 0);
};

IMPULSE myDoSQL (connection cur, string theStatement)
{
    return dbeDoSQL (cur, theStatement, strlen (theStatement));
};

IMPULSE myCompile (connection cur, string theStatement)
{
    return dbeCompile (cur, theStatement, strlen (theStatement));
};

// Output functions

static void printHeading (connection cur)
{
    char        ownBuf [256];
    int         loop, nbrofCols;

    nbrofCols = dbeNbrofCols (cur);
    for (loop = 1; loop <= nbrofCols; loop++) {
        strcpy (ownBuf, dbeColName (cur, loop));
        printf (" %s", ownBuf);
    };
    printf ("\n-------------------------------------------------------------------------
                --------\n");
};

static void printTuple (connection cur)
{
    char        ownBuf [256];
    int         loop, nbrofCols;
```

```
    nbrofCols = dbeNbrofCols (cur);
    for (loop = 1; loop <= nbrofCols; loop++) {
        strcpy (ownBuf, dbeColName (cur, loop));
        strcat (ownBuf, " => ");
        strcat (ownBuf, dbeGetCol (cur, loop));
        printf ("%s\n", ownBuf);
    };
    printf ("\n");
};

static boolean selectDay (connection cur)
{
    char        response [64];

    printf ("Give the date (stop with '!') => ");
    scanf ("%s", response);
    if (response [0] != '!') {
        dbeSetVar (cur, 1, response, strlen (response));
        checkErc (cur);
        return TRUE;
    } else
        return FALSE;
};

static void printAccount (connection cur)
{
    status  result;

    printf ("The info on that date is:\n");
    dbeExecute (cur);
    dbeToStart (cur);
    if (checkErc (cur))
        while (dbePoints (cur)) {
            printTuple (cur);
            checkErc (cur);
            dbeForth (cur);
            checkErc (cur);
        };
};

int main (void)
{
    connection  myCur;

    if (!checkErc (myCur = myConnect ("JOBS")))
        return 1;
    myCompile (myCur, "SELECT TASK, WMIN/60, INFO FROM ACCOUNT \
        WHERE WDATE = :1");
    if (checkErc (myCur)) {
        do {
            if (selectDay (myCur))
                printAccount (myCur);
            else
                break;
        } while (TRUE);
        dbeDisconnect (myCur);
    };
    return 0;
};
```

```
// ------------------------------------------------------------------------
//
//  SYSTEM:           OOPoS Proto
//
//  UNIT:             dbEngine
//
//  AUTHOR:           $Author:   JuM  $
//
//  $Workfile:   dbetest4.c  $
//  $Revision:   2.4  $ $Date:   10 Jan 1991 23:50:14  $ $Author:   JuM  $
//
// ------------------------------------------------------------------------
//
// ABSTRACTS: A program for making a worktime-account transaction with
//  checking of the values. It compiles several SQL-statements by using
//  the same cursor. The statements include both using bind-variable and
//  queries.
//
// ------------------------------------------------------------------------
//
//  $Log:   D:/jumsoft/dbengine/v/dbetest4.c  $
//
//                Rev 2.4   10 Jan 1991 23:50:14   JuM
//             Initial revision.
//
//
//                   Copyright (c) 1990,1991 by Nokia Data
//
// ------------------------------------------------------------------------

#include <stdio.h>
#include <string.h>
#include <time.h>

#include "dbengine.h"

static boolean checkErc (status retCode, connection c)
{
    if (retCode != OK) {
        printf ("SQL-VIRHE: %s\n", dbeErrorDescription (c));
        return FALSE;
    } else
        return TRUE;
};

CONSTRUCTOR myConnect (string toDb)
{
    return dbeConnect (toDb, strlen (toDb), "", 0, "", 0);
};

IMPULSE myCompile (connection cur, string theStatement)
{
    return dbeCompile (cur, theStatement, strlen (theStatement));
};

// Input functions

static void getDate (connection cur)
{
    struct tm*  tP;
    time_t      help;
    char        wdate [12];

    time (&help);
    tP = localtime (&help);
    printf ("T•n••n on %d.%d.19%d\n", tP->tm_mday, tP->tm_mon+1, tP->tm_year);
    _strdate (wdate);
    checkErc (dbeSetVar (cur, 1, wdate, strlen (wdate)), cur);
};

static void getAccountInfo (connection cur)
{
    int     wMin;
    long    wHour;
    char    info [128];

    memset (info, '\0', sizeof (info));
```

```
    printf ("Anna tuntim••r• ja lyhyt kuvaus aiheesta!\n");
    scanf ("%ld %s", &wHour, info);
    wMin = (int) (wHour * 60);
    checkErc (dbeSetVarInt (cur, 2, wMin), cur);
    checkErc (dbeSetVar (cur, 4, info [1], strlen (info), cur);
};

static boolean checkProjId (connection cur, string projid)
{
    checkErc (myCompile (cur,
        "SELECT TASKS. ID, TASKS. INFO, PROJECTS. NAME, WORKS. NAME \
         FROM TASKS, WORKS, PROJECTS \
         WHERE TASKS. ID = :1 \
             AND TASKS. WORKNR = WORKS. NR \
             AND WORKS. PROJNR = PROJECTS. NR"), cur);
    checkErc (dbeSetVar (cur, 1, projid, strlen (projid)), cur);
    checkErc (dbeExecute (cur), cur);
    checkErc (dbeToStart (cur), cur);
    if (dbePoints (cur)) {
        printf ("Tehty projektia %s; %s,\njoka kuuluu luokkaan %s / %s.\n",
            dbeGetCol (cur, 1), dbeGetCol (cur, 2), dbeGetCol (cur, 3),
            dbeGetCol (cur, 4));
        return TRUE;
    } else {
        printf ("VIRHE: %s-projektia ei ole avattu!\n", projid);
        return FALSE;
    };
};

static void printTotal (connection cur, string projid)
{
    float   total;

    checkErc (myCompile (cur,
        "SELECT SUM (WMIN) \
         FROM ACCOUNT \
         WHERE TASK = :1"), cur);
    checkErc (dbeSetVar (cur, 1, projid, strlen (projid)), cur);
    checkErc (dbeExecute (cur), cur);
    checkErc (dbeToStart (cur), cur);
    if (dbePoints (cur)) {
        sscanf (dbeGetCol (cur, 1), "%f", &total);
        printf ("Merkitty. T•t• projektia on tehty kaikkiaan %.2f tuntia.\n",
            total/60);
    };
};

int main (int argc, char* argv [])
{
    connection  insCur;
    char        projId [20];

    if (argc < 2) {
        printf ("Kutsu ohjelmaa antamalle projektitunnus komentorivill•!\n");
        return 1;
    } else
        strcpy (projId, argv [1]);

    if (dbeConnectionStatus (insCur = myConnect ("JOBS")) == OK) {
        if (checkProjId (insCur, projId)) {
            checkErc (myCompile (insCur,
                "INSERT INTO ACCOUNT (WDATE, WMIN, TASK, INFO) \
                    VALUES (:1, :2, :3, :4)"), insCur);
            checkErc (dbeSetVar (insCur, 3, projId, strlen (projId)), insCur);
            getDate (insCur);
            getAccountInfo (insCur);
            checkErc (dbeExecute (insCur), insCur);
            printTotal (insCur, projId);
            checkErc (dbeDisconnect (insCur), insCur);
        };
    };
    return 0;
};
```

```
// ------------------------------------------------------------------------
//
//  SYSTEM:           OOPoS Proto
//
//  UNIT:             dbEngine
//
//  AUTHOR:           $Author:   JuM  $
//
//  $Workfile:   dbetest5.c  $
//  $Revision:   2.4  $ $Date:   10 Jan 1991 23:50:16  $ $Author:   JuM  $
//
// ------------------------------------------------------------------------
//
// ABSTRACTS: An interactive database-editor for going through the database,
//  selecting items by key and updating, inserting and removing them to/from
//  the database.
//
//  This simple program contains the essentials of interactive database
//  maintenance. The idea is to have three cursors for reading: two
//  for accessing the items in it sequentilly either in ascending or
//  descending order and one for accessing them randomly by the key value.
//  The current key value is also kept in an internal variable, and the
//  writing, which is performed through yet another cursor is done by
//  always compiling a new statement before the execution.
//
// ------------------------------------------------------------------------
//
//  $Log:   D:/jumsoft/dbengine/v/dbetest5.c  $
//
//                Rev 2.4   10 Jan 1991 23:50:16   JuM
//              Initial revision.
//
//
//                    Copyright (c) 1990,1991 by Nokia Data
//
// ------------------------------------------------------------------------

// As a new feature with the version 2 dbEngine this application also
// uses locking an asynchronous writes implemented by threads.

#include <stdio.h>
#include <string.h>
#include <conio.h>

#include "dbengine.h"

static  char    currFName [30];
static  char    currSName [30];

static boolean checkErc (connection c)
{
    if (dbeConnectionStatus (c) != OK) {
        printf ("SQLERROR %s\n", dbeErrorDescription (c));
        return FALSE;
    } else
        return TRUE;
};

static void clearScreen (void)
{
    printf ("%c%s", 0x1b, "[2J");
    printf ("Selaa tietoja numeron•pp•imill•, lopeta painamalla !.\n\n");
};

CONSTRUCTOR myConnect (string toDb)
{
    return dbeConnect (toDb, strlen (toDb), "", 0, "", 0);
};

IMPULSE myCompile (connection cur, string theStatement)
{
    return dbeCompile (cur, theStatement, strlen (theStatement));
};

static void setVar (connection cur, int varNr, string toThis)
{
    dbeSetVar (cur, varNr, toThis, strlen (toThis));
```

```
    checkErc (cur);
};

static void printTuple (connection cur, connection phoneCur)
{
    printf ("%s %s\n", dbeGetCol (cur, 1), dbeGetCol (cur, 2));
    printf ("   %-24s, %s %s\n", dbeGetCol (cur, 3), dbeGetCol (cur, 4),
        dbeGetCol (cur, 5));
    setVar (phoneCur, 1, dbeGetCol (cur, 1));
    setVar (phoneCur, 2, dbeGetCol (cur, 2));
    dbeExecute (phoneCur);
    dbeToStart (phoneCur);
    while (dbePoints (phoneCur) && checkErc (phoneCur)) {
        switch (dbeGetColInt (phoneCur, 2)) {
            case 0:
                printf ("   puh kotiin: ");
                break;
            case 1:
                printf ("   puh t”ihin: ");
                break;
            default:
                printf ("   NMT tms.  : ");
        };
        printf ("%s\n", dbeGetCol (phoneCur, 1));
        dbeForth (phoneCur);
    };
    printf ("\n");
};

static void getName (void)
{
    printf ("Forename => ");
    scanf ("%s", currFName);
    printf ("Surname => ");
    scanf ("%s", currSName);
    printf ("\n");
};

static void updCurrName (connection cur)
{
    if (dbePoints (cur)) {
        strcpy (currFName, dbeGetCol (cur, 1));
        strcpy (currSName, dbeGetCol (cur, 2));
    } else {
        strcpy (currFName, " ");
        strcpy (currSName, " ");
    };
};

static void findData (connection cur, connection phoneCur)
{
    clearScreen ();
    printf ("Give the name to be searched! Wildcards _ and % are allowed.\n");
    getName ();
    dbeSetVar (cur, 1, currFName, strlen (currFName));
    checkErc (cur);
    dbeSetVar (cur, 2, currSName, strlen (currSName));
    checkErc (cur);
    dbeExecute (cur);
    checkErc (cur);
    dbeToStart (cur);
    checkErc (cur);
    updCurrName (cur);
    clearScreen ();
    while (dbePoints (cur)) {
        printTuple (cur, phoneCur);
        dbeForth (cur);
        checkErc (cur);
    };
}; // findData

static void fetchNext (connection cur, connection phoneCur)
{
    dbeSetVar (cur, 1, currSName, strlen (currSName));
    checkErc (cur);
    dbeSetVar (cur, 2, currSName, strlen (currSName));
    checkErc (cur);
```

```
    dbeSetVar (cur, 3, currFName, strlen (currFName));
    checkErc (cur);
    dbeExecute (cur);
    checkErc (cur);
    dbeToStart (cur);
    while (dbePoints (cur) &&
        (strcmp (dbeGetCol (cur, 1), currFName) == 0) &&
        (strcmp (dbeGetCol (cur, 2), currSName) == 0))
            dbeForth (cur);
    updCurrName (cur);
    clearScreen ();
    if (dbePoints (cur))
        printTuple (cur, phoneCur);
    else
        printf ("No more records!\n");
};

static void addPhoneNbr (connection phoneCur)
{
    char        nbr [40];
    int         location;
    connection  writeCur;

    printf ("Uusi puhelinnumero => ");
    scanf ("%s", nbr);
    printf ("Kotiin = 0, T”ihin = 1, NMT tms. = 2 => ");
    scanf ("%d", &location);
    if (checkErc (writeCur = dbeDuplicate (phoneCur))) {
        myCompile (writeCur,
            "INSERT INTO PHONE \
            (FNAME, SNAME, PHONE, LOCATION) \
            VALUES (:1, :2, :3, :4)");
        setVar (writeCur, 1, currFName);
        setVar (writeCur, 2, currSName);
        setVar (writeCur, 3, nbr);
        dbeSetVarInt (writeCur, 4, location);
        checkErc (writeCur);
        dbeExecute (writeCur);
        checkErc (writeCur);
    };
    dbeDisconnect (writeCur);
};

int main (void)
{
    connection  readSeqAscCur, readSeqDescCur, readRandCur, phoneCur;
    boolean     endLoop = FALSE;

    // Connect the cursors to the database
    if (!checkErc (readSeqAscCur = dbeConnect ("PEOPLE", 6, "SYSADM", 6,
        "SYSADM", 6)))
            return 1;
    if (!checkErc (readSeqDescCur = dbeDuplicate (readSeqAscCur)))
            return 2;
    if (!checkErc (readRandCur = dbeDuplicate (readSeqDescCur)))
            return 3;
    if (!checkErc (phoneCur = dbeDuplicate (readSeqDescCur)))
            return 4;
    dbeSetIsolationLevel (readRandCur, lockCurrent);

    // Compile the queries
    myCompile (readSeqAscCur,
        "SELECT FNAME, SNAME, STREET, ZIP, CITY \
        FROM ADDRESS \
        WHERE (SNAME > :1) OR (SNAME = :2 AND FNAME > :3) \
        ORDER BY SNAME ASC, FNAME ASC");
    checkErc (readSeqAscCur);

    myCompile (readSeqDescCur,
        "SELECT FNAME, SNAME, STREET, ZIP, CITY \
        FROM ADDRESS \
        WHERE (SNAME < :1) OR (SNAME = :2 AND FNAME < :3) \
        ORDER BY SNAME DESC, FNAME DESC");
    checkErc (readSeqDescCur);

    myCompile (readRandCur,
        "SELECT FNAME, SNAME, STREET, ZIP, CITY \
```

```
            FROM ADDRESS \
            WHERE FNAME LIKE :1 AND SNAME LIKE :2 \
            ORDER BY SNAME ASC, FNAME ASC");
        checkErc (readRandCur);

        myCompile (phoneCur,
            "SELECT PHONE, LOCATION \
            FROM PHONE \
            WHERE FNAME = :1 AND SNAME = :2 \
            ORDER BY LOCATION");
        checkErc (phoneCur);

        strcpy (currFName, " ");
        strcpy (currSName, " ");
        clearScreen ();
        do {
            switch (getch ()) {
                case '4': // Arrow left
                    fetchNext (readSeqDescCur, phoneCur);
                    break;
                case '6': // Arrow right
                    fetchNext (readSeqAscCur, phoneCur);
                    break;
                case '8': // Arrow up
                    findData (readRandCur, phoneCur);
                    break;
                case 'P':
                    addPhoneNbr (phoneCur);
                    break;
                case '!':
                    endLoop = TRUE;
                    break;
                default:
                    clearScreen ();
            };
        } while (!endLoop);
        dbeDisconnect (readSeqAscCur);
        checkErc (readSeqAscCur);
        dbeDisconnect (readSeqDescCur);
        checkErc (readSeqDescCur);
        dbeDisconnect (readRandCur);
        checkErc (readRandCur);
        dbeDisconnect (phoneCur);
        checkErc (phoneCur);
        return 0;
};
```

```
// ------------------------------------------------------------------------
//
//  SYSTEM:           OOPoS Proto
//
//  UNIT:             dbEngine
//
//  AUTHOR:           $Author:   JuM  $
//
//  $Workfile:   dbetest6.c  $
//  $Revision:   2.4  $ $Date:   10 Jan 1991 23:50:18  $ $Author:   JuM  $
//
// ------------------------------------------------------------------------
//
// ABSTRACTS: Storing and retrieving statements. This program makes the
//  hours-report by help of a few precompiled statements.
//
// ------------------------------------------------------------------------
//
//  $Log:   D:/jumsoft/dbengine/v/dbetest6.c  $
//
//                Rev 2.4   10 Jan 1991 23:50:18   JuM
//              Initial revision.
//
//
//                    Copyright (c) 1990,1991 by Nokia Data
//
// ------------------------------------------------------------------------

#include <stdio.h>
#include <string.h>
#include <time.h>

#include "dbengine.h"

static boolean checkErc (connection c)
{
    if (dbeConnectionStatus (c) != OK) {
        printf ("SQLERROR %s\n", dbeErrorDescription (c));
        return FALSE;
    } else
        return TRUE;
};

CONSTRUCTOR myConnect (string toDb)
{
    return dbeConnect (toDb, strlen (toDb), "", 0, "", 0);
};

IMPULSE myCompile (connection cur, string theStatement)
{
    return dbeCompStore (cur, theStatement, strlen (theStatement));
};

static void printHourReport (connection cur, string startDate, string endDate)
{
    printf ("Tuntiraportti ajalta %s - %s\n\n", startDate, endDate);
    printf ("%-10s%-30s%18s\n", "Ty”numero", "Ty”n nimi", "Tehdyt tunnit");
    printf ("-------------------------------------------------------\n");

    myCompile (cur,
            "SELECT WORKS. NR, WORKS. NAME, SUM (ACCOUNT.WMIN)/60 \
             FROM WORKS, TASKS, ACCOUNT \
             WHERE WORKS. NR = TASKS. WORKNR AND TASKS. ID = ACCOUNT. TASK \
               AND ACCOUNT. WDATE BETWEEN :1 AND :2 \
             GROUP BY WORKS.NR, WORKS.NAME");

    dbeSetVar (cur, 1, startDate, strlen (startDate));
    checkErc (cur);
    dbeSetVar (cur, 2, endDate, strlen (endDate));
    checkErc (cur);
    dbeExecute (cur);
    checkErc (cur);

    dbeToStart (cur);
    while (dbePoints (cur)) {
        printf ("%-10s%-30s%18s\n", dbeGetCol (cur, 1),
            dbeGetCol (cur, 2), dbeGetCol (cur, 3));
```

```
        dbeForth (cur);
    };

    myCompile (cur,
            "SELECT SUM (ACCOUNT. WMIN)/60 \
             FROM ACCOUNT \
             WHERE ACCOUNT. WDATE BETWEEN :1 AND :2 \
             AND ACCOUNT. TASK IN \
                 (SELECT TASKS. ID FROM TASKS WHERE TASKS. WORKNR IN \
                     (SELECT WORKS. NR FROM WORKS))");

    dbeSetVar (cur, 1, startDate, strlen (startDate));
    checkErc (cur);
    dbeSetVar (cur, 2, endDate, strlen (endDate));
    checkErc (cur);
    dbeExecute (cur);
    checkErc (cur);

    dbeToStart (cur);
    if (dbePoints (cur)) {
        printf ("%58s\n", "----------");
        printf ("%58s\n", dbeGetCol (cur, 1));
    };
};

int main (int argc, char* argv [])
{
    connection  cur;
    char        startDate [20], endDate [20];

    if (argc < 3) {
        printf ("Kutsun syntaksi: tuntirap ensimm•inen-pvm viimeinen-pvm\n");
        return 1;
    } else {
        strcpy (startDate, argv [1]);
        strcpy (endDate, argv [2]);
    };

    cur = myConnect ("JOBS");
    if (dbeConnectionStatus (cur) != OK)
        checkErc (cur);
    else {
        dbeSetIsolationLevel (cur, lockCurrent);
        printHourReport (cur, startDate, endDate);
        dbeCommit (cur); // To leave the stored statements in
        dbeDisconnect (cur);
    };
    return 0;
};
```

```
// ------------------------------------------------------------------------
//
//  SYSTEM:           OOPoS Proto
//
//  UNIT:             dbEngine
//
//  AUTHOR:           $Author:   JuM  $
//
//  $Workfile:   dbetest7.c  $
//  $Revision:   2.4  $ $Date:   10 Jan 1991 23:50:22  $ $Author:   JuM  $
//
// ------------------------------------------------------------------------
//
// ABSTRACTS: Making a detailed hours-report by using named bind-variables
//
// ------------------------------------------------------------------------
//
//  $Log:   D:/jumsoft/dbengine/v/dbetest7.c  $
//
//                Rev 2.4   10 Jan 1991 23:50:22   JuM
//             Initial revision.
//
//
//                   Copyright (c) 1990,1991 by Nokia Data
//
// ------------------------------------------------------------------------

#include <stdio.h>
#include <string.h>
#include <time.h>

#include "dbengine.h"

static boolean checkErc (connection c)
{
    if (dbeConnectionStatus (c) != OK) {
        printf ("SQLERROR %s\n", dbeErrorDescription (c));
        return FALSE;
    } else
        return TRUE;
};

CONSTRUCTOR myConnect (string toDb)
{
    return dbeConnect (toDb, strlen (toDb), "", 0, "", 0);
};

static void prepareStatement (connection cur, string ofSQL)
{
    dbeCompStore (cur, ofSQL, strlen (ofSQL));
    checkErc (cur);
};

static int bindNbr (connection cur, string ofName)
{
    int result;

    result = dbeBindNumber (cur, ofName, strlen (ofName));
    if (result == INVALID_PROGRAM_VARIABLE_NAME)
        checkErc (cur);
    return result;
};

static void printDetailReport (connection cur, string startDate, string endDate)
{
    printf ("Yksil”ity tuntiraportti ajalta %s - %s\n\n", startDate, endDate);
    printf ("%-10s %-12s%6s %-20s\n", "Pvm", "Projekti", "Tunnit", "Selitys");
    printf ("------------------------------------------------------------------------
                ------\n");

    prepareStatement (cur,
            "SELECT WDATE, TASK, WMIN/60, INFO \
             FROM ACCOUNT \
             WHERE WDATE BETWEEN :first_date AND :last_date");

    dbeSetVar (cur, bindNbr (cur, "first_date"), startDate,
        strlen (startDate));
```

```
    checkErc (cur);
    dbeSetVar (cur, bindNbr (cur, "last_date"), endDate, strlen (endDate));
    checkErc (cur);
    dbeExecute (cur);
    checkErc (cur);

    dbeToStart (cur);
    while (dbePoints (cur)) {
        printf ("%-10s %-12s%6s %-47.47s\n", dbeGetCol (cur, 1),
            dbeGetCol (cur, 2), dbeGetCol (cur, 3), dbeGetCol (cur, 4));
        dbeForth (cur);
    };

    prepareStatement (cur,
            "SELECT SUM (ACCOUNT. WMIN)/60 \
             FROM ACCOUNT \
             WHERE ACCOUNT. WDATE BETWEEN :first_date AND :last_date \
             AND ACCOUNT. TASK IN \
                 (SELECT TASKS. ID FROM TASKS WHERE TASKS. WORKNR IN \
                     (SELECT WORKS. NR FROM WORKS))");

    dbeSetVar (cur, bindNbr (cur, "first_date"), startDate,
        strlen (startDate));
    checkErc (cur);
    dbeSetVar (cur, bindNbr (cur, "last_date"), endDate, strlen (endDate));
    checkErc (cur);
    dbeExecute (cur);
    checkErc (cur);

    dbeToStart (cur);
    if (dbePoints (cur)) {
        printf ("%29s\n", "-----------");
        printf ("%29s\n", dbeGetCol (cur, 1));
    };
};

int main (int argc, char* argv [])
{
    connection  cur;
    char        startDate [20], endDate [20];

    if (argc < 3) {
        printf ("Kutsun syntaksi: p•iv•rap ensimm•inen-pvm viimeinen-pvm\n");
        return 1;
    } else {
        strcpy (startDate, argv [1]);
        strcpy (endDate, argv [2]);
    };

    cur = myConnect ("JOBS");
    if (checkErc (cur))
    {
        printDetailReport (cur, startDate, endDate);
        dbeDisconnect (cur);
    };
    return 0;
};
```