xplain2sql 4.1.0 manual

2011-December-20

Xplain Technology Ltd
by Berend de Boer

Contents

Introduction

This document is a manual for compiling and using xplain2sql. xplain2sql is an Xplain to SQL converter. Xplain is a data definition and data manipulation language, just like SQL. However, Xplain has many advantages. It's a very clean, concise and orthogonal language. Unlike SQL, in Xplain there's usually just one solution to a problem, not dozens. Xplain is also easier to learn. And it supports both aggregation and generalization. Both the *is-a* and *has-a* using an elegant definition language. Inheritance in data models is still something one seldom encounters. With Xplain this is natural and easy. Xplain therefore is a perfect complement to an all object-oriented language like Eiffel. By the way, in chapter **7** Xplain is treated in more detail and some examples are given.

There's one problem however, you can draw Xplain diagrams and write Xplain code, but there's no[1] way to execute it. In the Xplain book (see (ter Bekke, 1992)) some guidelines to convert Xplain to SQL are given. But wouldn't it be really useful to be able to convert Xplain to SQL automatically?

And that was the start of a long list of tools. The first such a tool was written in Turbo Pascal 7.0. Later versions were written in Delphi. As they mostly only did the data definition part, they were named ddl2sql. These converters have been used on serious business systems, for example to generate the Microsoft SQL Server 6.5 code on the new administrative system for Urk, Fish Auction, the largest flat-fish auction in Europe.

When I wanted to write some serious programs on Unix, I started to look for a language that compiled to native code, was cross-platform, object-oriented, and not C++. I knew Eiffel a long time, having used design-by-contract for 5 years. But I hadn't used Eiffel. I was surprised by the activity and tools that were available at that moment. Especially SmallEiffel (now called SmartEiffel) attracted me as I could use familiar Emacs, a compiler that wasn't buried behind layers of a supposedly fancy IDE. It was time to learn Eiffel, the language. And there was the **Eiffel Struggle 1999**. It was time to write the greatest Xplain converter of all times, which converted the entire Xplain language to an SQL dialect of your choice.

With xplain2sql two goals are achieved:

1. Write database code in an object-oriented data definition language.
2. Write it once, and port to any SQL dialect you want.

So here it is, xplain2sql. This document explains how to compile and install it, how to use it. Also it gives a short introduction to the Xplain language itself (see

[1] There is also an Xplain product. But compared to todays tools, this is quite primitive. It's also a one-tier system, so therefore not accessible from modern RAD tools like Delphi.

chapter 7). More information can be found on my Xplain page at `http://www.pobox.com/~berend/xplain/`.

And of course, this document has been created with Emacs and compiled to PDF with ConTEXt.

Acknowledgements

It has been a gratifying experience to know that so many people use this product for so many different purposes. So thanks to all people who use this product and have told me something about how they use it. Many users have also sent me suggestions for improvement. Especially I want to thank:

- Martin van Dinther for the very sensible suggestion to translate A to varchars instead of chars. Also thanks for your suggestions on improving the Oracle output.
- Mark Hissink for trying the Microsoft Access output.
- Jacco Eerland for suggesting a fix so the middleware code would compile with Delphi 7.
- John Wiggings for suggestions to improve PostgreSQL output.

License and Support

License

xplain2sql is an Open Source program. It's available under the MIT License. You can find more details in the `LICENSE` file, provided with the xplain2sql distribution.

Mailinglist

You can subscribe to the Xplain mailing list. Subject of this mailing list is Xplain the language and related tools like xplain2sql. Go to `http://groups.yahoo.com/group/Xplain` to read current message and to subscribe.

Paid support

If your company is interested in Xplain or xplain2sql, you can ask me to provide a one or two day course in Xplain and/or xplain2sql. Prices are 1200 NZD a day, excluding VAT, travel and hotel expenses. Contact me at **berend@pobox.com**. If your company is not satisfied about the course, the fee is refunded (excluding the expenses).

1 Compilation and installation

This chapter explains how to compile and install xplain2sql. If you want to avoid compiling xplain2sql, packages and binaries for several platforms are available at `http://www.pobox.com/~berend/xplain2sql/`.

1.1 Requirements

1.1.1 Supported platforms

xplain2sql compiles on both Unix (tested on Ubuntu 11.94 and FreeBSD 8.2) and Windows (tested on Windows 2000 sp2). Any platform with a Standard C compiler should be fine.

1.1.2 Supported compilers

I suggest you use the C code release of xplain2sql. It can be compiled with any Standard C compiler.

If you want to compile the Eiffel sources, things are only slightly more difficult. You can use either SmartEiffel 1.2r7 or ISE Eiffel 6.8 or later. Earlier SmartEiffel releases or the SmartEiffel 2.x fork are not supported.

1.1.3 Libraries

If you want to compile the Eiffel sources of xplain2sql, you need to have Gobo 3.9 or later installed on your system.

1.2 Compiling xplain2sql C source

Just unzip the archive:

```
unzip xplain2sql-3.0-csrc.zip
```

And run make:

```
make
```

If you use the free Borland C 5.5 compiler, make sure to use the special target:

```
make xplain2sql-bcc
```

1.3 Compiling xplain2sql Eiffel source

If you have the Gobo tools in your path, building it on any platform using SmartEiffel can be done with:

```
geant compile_se
```

For ISE Eiffel it is:

```
geant compile_ise
```

There is a `Makefile`, but this is for development only.

1.4 Testing the installation

If you have the full Eiffel sources, and you use a Unix platform, you can run a few test targets:

```
make test
```

or

```
make bank
```

Errors are written to `test.err` in the first example or `bank.err` in the second. There should only be warnings in `test.err`. `bank.err` should be empty.

On NT, you need to have some Unix utilities on your path like `rm` and `cat`. If you have the NT resource Kit, the POSIX utilities will do fine. You probably can also use the win32 port from Cygnus. And with a decent `make`, you should be able to execute these commands too. I've been not very lucky with makes on NT. Many of the NT makes seem to have problems with redirection inside a Makefile.

1.5 Installation

Installation is as simple as copying `plain2sql` or `xplain2sql.exe` to a directory somewhere in your path. On Unix systems this can be `/usr/local/bin`, on NT this can be `/winnt/system32`.

You can also copy the man file `xplain2sql.1` somewhere in your man path.

2 Using xplain2sql

2.1 Converting xplain files

To convert an Xplain ddl to some dialect of SQL, type:

```
xplain2sql -tsql test.ddl
```

The above example translates the Xplain `test.ddl` to Transact-SQL, writing it to standard output. `xplain2sql` exits with exit level 1 if a fatal error has occurred.

To generate output for MySQL, use:

```
xplain2sql -mysql test.ddl
```

To save output to `test.sql`, redirect the output as usual:

```
xplain2sql -tsql test.ddl > test.sql
```

In table **2.1** the SQL dialects xplain2sql supports are given. See also section **2.2** about known limitations.

In table **2.2** the other options xplain2sql supports are given. Options only have affect if the selected SQL dialect supports them.

2.2 Known limitations

The conversion is not (yet) optimal for all supported dialects. Sometimes Xplain is not fully supported, and sometimes limitations in the SQL dialect or a not yet finished Builder gives problems. This chapter also mentions some limitations in support for xplain2sql extensions described in the next chapter.

2.2.1 No full support for the Xplain language yet

The Xplain language is supported to a large degree, but still some things are missing. Most important are:

1. Names are case-sensitive. This can be changed fairly easily, but a consistent spelling helps reading Xplain source in the author's opinion.
2. Data types are not checked for correctness. When a string is assigned to an integer, xplain2sql relies on the SQL dialect to convert it. And that does not always succeed.
3. The *assert* command should work for all dialects, provided they support at least views, but no code is generated to check if the value of an assertion is in the specified range. For example:

Dialect	Remark
-ansi	Output ansi-92 sql.
-basic	Output basic sql statements.
-db26	Output DB/2 6 sql statements.
-db2 -db271	Output DB/2 7.1 sql statements (tested against DB/2 9.5.0).
-firebird -firebird21	Output FireBird 2.1 sql (tested against version 2.1.0.17798).
-interbase -interbase6	Output InterBase or FireBird 1.x sql (tested against version 1.0.3).
-mysql -mysql5	Output MySQL 5.0 (tested against version 5.1.54).
-mysql4	Output MySQL 4.0 or higher. Only data definition could be made to work more or less.
-mysql322	Output MySQL 3.22. Only data definition could be made to work more or less.
-oracle -oracle901	Output Oracle sql (tested on Oracle 9.0.1).
-pgsql -pgsql81	Output PostgreSQL 8.1 sql (tested against version 8.4.10).
-pgsql73	Output PostgreSQL 7.3 sql (tested against version 7.3.3).
-sqlite -sqlite3	Output SQLite version 3 (tested against version 3.2.7).
-tsql65	Output Microsoft Transact-SQL 6.5. Purging of columns doesn't work. And Microsoft SQL Server 6.5 is less stable than SQL Server 7.0, so a statement might generate a crash of the server.
-tsql70	Output Microsoft Transact-SQL 7.0.
-tsql -tsql2000	Output Microsoft SQL Server 2000 Transact-SQL (tested against SQL Server Express 2005).

Table 2.1 SQL dialects supported by xplain2sql

Option	Explanation
-attributenull	Attributes (columns) are normally not null except BLOB columns. With this option columns are null by default.
-noauto	Do not create auto-generated primary keys. Auto-generated primary keys are the default, so you don't have to care about primary keys in your applications.
-nodatabase	Do not output code for the Xplain database command. This makes it easier to create scripts that can be run in different databases. Without this option the Xplain database command will change to current database to the set database first.
-nosp	Don't create a stored procedure to insert rows.
-timestamp	Add the fieldname `ts_<type name>` of type 'timestamp' to every table. This is a Microsoft SQL Server specific option. Note that the 'timestamp' type has nothing todo with date or time. Timestamps are degenerate-degenerate field calls, but it's the only form of concurrency control we have with MS SQL Server.
-view	Create a view on every table. Could be useful if you want to have security only on views, and not on tables.
-xml	Generate an XML description of the generated SQL code. This XML description is very useful to generate wrapper classes for your favorite programming language.

Table 2.2 Options supported by xplain2sql

assert invoice *its* lines (1..*) = *count* invoice line *per* invoice.

will allow you to query the number of lines per invoice with:

get invoice *its* lines.

But there's no check that an invoice has at least one line. Such checks should be run at the end of a transaction, and very view databases allow such checks.

4. The level of support for the *init* (*default*) statements depends on the particular dialect. Constant values are most widely supported. Literal values (including expressions that evaluate to a constant such as 1 + 1) need trigger support in certain dialects. All other forms require support for triggers.
   Due to some parsing peculiarities there should be no comment between a type and its *init* (*default*) statements.
   case expressions are not supported at the moment.
5. The *check* constraint is not supported.
6. Make sure the *init* statement immediately follows the *type* statement, without intervening other statements. This is due to the fact that `xplain2sql` does not

build an AST, but generates output immediately while parsing. This will be corrected in a future release.
7. Purging of attributes is supported but not all dialects support the **alter table drop column** statement.
8. *purge* of attributes or inits is not yet supported.
9. *case* keyword in inheritance is not yet supported.
10. Because the date functions are not supported, any usage of a literal date is sql dialect specific.
11. Not all Xplain 5.8 string and mathematical functions are supported.
12. The *newline* keyword is not supported.
13. The *input* keyword is not supported.
14. Constants can be updated, although this is not allowed in Xplain.
15. The *cascade* keyword is not supported.

2.2.2 General conversion to SQL issues

There are six problems converting Xplain to SQL:

1. If a dialect doesn't support double quoted identifiers, don't try to use an sql keyword as name for a type or base. Currently they're not translated or escaped.
2. Xplain data definition support can be converted quite well to SQL. Only *init* construct which refer to an attribute, need trigger or stored procedure support. For a really good implementation you need to be able to write a 'before insert' trigger. Else a conversion is not possible or you need to do all your inserts through a stored procedure.
   InterBase should be able to support *init* quite well, but the implementation is severely lacking in this respect.
3. The *value* statement needs some temporary storage, local to the current transaction.
   InterBase and DB/2 don't have this, so they essentially only support single-user. The generated code need to be modified on the fly to support multiple users by creating tables with unique names or so.
   Microsoft SQL Server and PostgreSQL support temporary storage.
4. The *extend* statement also needs temporary storage, the problems and limitations are the same as for the *value* statement.
5. Full support for the *some* function requires the ability to select a single row, usually the first, from a statement returning multiple rows. Not all dialects have this ability. If the *some* statement selects a single instance, i.e.:

   get some t "1" *its* a.

   the conversion is safe. Also if you know the result is just a single instance, the conversion should also be good enough.
6. It is not clearly defined if Xplain is case-sensitive, but it seems it is. The code that xplain2sql generates is case-sensitive on a database that is case-sensitive

and case-insensitive on a database that is case-insensitive. This is considered a feature. For certain backends it is possible to generate case-sensitive code on a database that is case-insensitive, but a need for this option has not yet arisen. All relational database servers seem to be case-sensitive by default, except Microsoft SQL Server. With Microsoft SQL Server 2000, case-sensitivity can be set on a per database basis.

2.2.3 DB/2 6 limitations

xplain2sql has the following limitations when converting to DB/2 version 6 (but **see also 2.2.4**):

1. Auto–generated primary keys are not supported in the version 6 script. But it seems DB/2 version 6 had them too, so you might just want to use the 7.1 script against version 6 databases.
2. Not-literal *inits* are not supported.
3. There is no support for temporary tables. The SQL generated for the *value* statement is therefore not immediately useful in a multi-user environment.
4. For extends, one use either views (option `-extendview`) or subselects (option `extendinline`) instead of extends with tables.
5. Insert, update and delete stored procedures cannot be generated as DB/2 only has Java or C stored procedures.
6. No full support for the *some* statement.

2.2.4 DB/2 7.1 limitations

xplain2sql's support for DB/2 is quite complete. These are the current limitations when converting to DB/2:

1. DB/2 7.1 cannot alter a temporary table, so the value statement cannot use one. The SQL generated for the *value* statement is therefore not immediately useful in a multi-user environment.
   Inside procedures, values should work fine though.
2. Full support for the *some* statement is possible, but has not yet been implemented.
3. You need at least Fixpack 2 if you want to be able to specify instance identification values in insert. And note that the code generated by xplain2sql only works in the single-user case, i.e. when loading the data base with data.

DB/2 output uses the '@' character as statement terminator. Use this command to process xplain2sql generated scripts with db2:

```
db2 -td@ -vf myfile.sql
```

Also the `db2upd` function must be present in your `sqllib/function` directory. That can be simply done with:

```
ln -s /usr/IBMdb2/V7.1/function/db2udp .
```

Replace V7.1 with your DB/2 version of course.

You also need temporary table space in your databases. I create this with:

```
CREATE USER TEMPORARY TABLESPACE USR_TEMPSPC1
  MANAGED BY SYSTEM USING ('usr_tempspc1')
```

Certain SQL output needs access to a table which returns just one row. xplain2sql uses SYSIBM.SYSDUMMY1. This is a default table that is present on every installation and is to be used for exactly this purpose. Users need to have select permissions on this table.

2.2.5 FireBird 2.1 limitations

xplain2sql has the following limitations when converting to FireBird 2.1:

1. A large number (larger than 31 bits) which has a domain restriction will fail if the specified domain restriction exceeds 31 bits. Example:

   base salary (I20) (5000..*).

2. Complex *inits* cannot be supported by FireBird. A complex init initializes an attribute using the value of another attribute through an its list like:

   init default entry *its* entry description = group *its* group description.

   It is not possible to generate SQL code for this with FireBird. Currently these *init*s are not masked out for FireBird output and generate an exception. So don't use them in data definition scripts meant to be processed by FireBird.
3. No full support for the *some* function as it is not possible to specify that a select should return just a single row.
4. FireBird does not support the `insert into ... default values` clause. If there is nothing to insert, xplain2sql will generate incorrect code. That can happen if there is an *init* on all attributes of a type.
5. An *extend* inside a stored procedure does not work.
6. A procedure with no statements does not compile on FireBird.

2.2.6 InterBase 6/FireBird 1.x limitations

xplain2sql has the following limitations when converting to InterBase or FireBird 1.x:

1. It's just too easy to create a query that bombs InterBase. This generator tries to output the most safe code, but if the output looks complex, it probably will not make InterBase happy.
2. A large number (larger than 31 bits) which has a domain restriction will fail if the specified domain restriction exceeds 31 bits. Example:

   base salary (I20) (5000..*).

3. InterBase does not have support for temporary tables. The SQL generated for the *value* statement is therefore not immediately useful in a multi-user environment. Inside a procedure *value* should work perfectly.
4. Complex *inits* cannot be supported by InterBase. A complex init initializes an attribute using the value of another attribute through an its list like:

   init default entry *its* entry description = group *its* group description.

   It is not possible to generate SQL code for this with InterBase. Currently these *init*s are not masked out for InterBase output and generate an exception. So don't use them in data definition scripts meant to be processed by InterBase.
5. No full support for the *some* function.
6. InterBase does not support the `insert into ... default values` clause. If there is nothing to insert, xplain2sql will generate incorrect code. That can happen if there is an *init* on all attributes of a type.
7. InterBase does not have support for temporary tables. The SQL generated for the *extend* statement is therefore not immediately useful in a multi-user environment because everyone will write to the same table.
8. An *extend* with a function might be a bit slower than in other SQL dialects, because InterBase does not have a coalesce function. Therefore we need to do a second pass to update those cases where the calculated value is null and does not have the default value for that function.
9. An *extend* inside a stored procedure does not work.
10. A procedure with no statements does not compile on InterBase.

2.2.7 Microsoft SQL Server limitations

The Microsoft SQL Server implementation is very complete. On one hand this is due to me using it a lot, on the other, because it is absolutely the most easiest SQL dialect to work with. Many other dialects require special syntax inside stored procedures, special hacks, or just don't have the required functionality. Kudos to Microsoft for this one!

The only conversion artefact is that a non literal *init default* is translated to a column which can be null. This is due to the fact that in Transact SQL triggers only fire after the constraints have been satisfied (incorrectly I think). There will be an on update trigger in a future release to check that a user does not set such attributes to null.

2.2.8 MySQL 3 and 4 limitations

MySQL is a very strange 'database' among the other databases xplain2sql supports. It doesn't obey the ACID properties, nor do its authors think this is useful. Anyway, only the data definition part of Xplain can be converted to MySQL. Other statements like the *extend* statement, ask too much of what MySQL has to offer. As MySQL doesn't support subselects, Xplain functions can also not be translated.

For MySQL 4 you must make sure ANSI mode has been enabled.

You might find the **MySQL Gotchas** page useful.

2.2.9 MySQL 5 limitations

MySQL 5 has begun to look like a real database. That means the support of xplain2sql is far more complete than for earlier versions. There still remain a few issues though.

1. The Xplain system default functions systemdate and loginname are not supported. Output is incorrect in this case.
2. Self referencing queries might not work due to problems with temporary tables, see **Temporary Table Problems**.
3. Specifying a non literal *init default* will cause a column to allow null values. This is due to the fact that in MySQL triggers only fire after the constraints have been satisfied (incorrectly I think). There will be an on update trigger in a future release to check that a user does not set such attributes to null.

xplain2sql has been tested with MySQL 5.0.15. Due to certain changes with earlier 5.0 versions this is probably also the minimum 5.0 release xplain2sql supports.

2.2.10 Oracle limitations

1. Domains are not translated to subtypes. Support for this would require to output a package with all types at the beginning of the output. This requires a change in how `xplain2sql` emits its output. It probably will be possible in version 3.0.
2. *extend* command not really supported. Works slightly outside stored procedures.
3. Output for Booleans should include restriction that it can be only 'T' or 'F'.
4. There is no support for more complex inits (those referring to other attributes).
5. Updating a column using the value of an extend will fail if there are no rows in the table.
6. Don't mix inserts that provide an instance id with inserts that rely on a autogenerated key. The current Oracle output does not update the generator when an instance id is provided.

7. Procedures (an `xplain2sql` extension) are transformed to a function that returns a reference cursor (weak cursor). I'm not sure if that is correct. The goal is to produce output that can generate a result set that is recognized by ADO/ODBC drivers.
8. A procedure with no statements will not compile under Oracle.

This is an example of how to call an Xplain generated function that returns a result set:

```
variable c refcursor
exec :c := "sp_test"
print c
```

2.2.11 PostgreSQL 8 limitations

PostgreSQL 8 support is very complete. Remaining issues:

1. Stored procedures cannot return multiple record sets. I.e. two gets in a stored procedure will lead to invalid code.
2. Names with spaces still have major issues. For example PostgreSQL auto generates a sequences for serial primary keys, The sequence name is derived from the column name. But if the column name contains spaces, it is not possible to retrieve the value of the generated key.
   The use `-nospace` to suppress spaces is therefore recommended.
3. Should emit `create type` statements for domains.
4. Constants are parsed as float8 and do not map to numeric datatypes.
5. If you use an identifier that is actuall an SQL keyword, PostgreSQL will not accept this SQL. PostgreSQL quoting doesn't actually quote keywords.

Note that xplain2sql needs the 'plpgsql' language loaded in the database. This can be achieved by the **create language** statement or by using the `createlang` utility. An example of loading this language into database 'mydatabase' is:

```
createlang --pglib=/usr/lib/pgsql plpgsql mydatabase
```

Another option is to add this to your template database, so it will always be present.

If a stored procedures returns a resultset (i.e. you have a *get* statement in it), that result set can be accessed with:

```
select * from "sp_my procedure"();
```

This result set works fine with ODBC. The output is different from xplain2sql 2.0, which used an older technique. The new technique is far more practical.

If you use the `-pgsql71` switch, code for the 7.1 dialect will be emitted. For set returning functions this is slightly different from the 7.3 code. Cursors can be accessed with:

```
begin;
select "sp_my procedure"();
fetch all in "<unnamed cursor 1>";
commit;
```

For ODBC applications I could get a result set if I first executed this statement:

```
begin;
select "sp_my procedure"();
```

And next executing, using the same connection and statement handle, this:

```
fetch all in "<unnamed cursor 1>";
```

No commit statement until you have read all rows. Execute commit

```
commit;
```

before calling another set returning function. It's a bit unfortunate that the next unnamed cursor will be called `"<unnamed cursor 2>"`. So that makes a complex application extremely awkward. If someone knows how to generate named cursors, please.

If you want to create a procedure in xplain2sql that is to be used as a , use *trigger procedure* instead of just *procedure*. xplain2sql will emit the necessary code so the procedure can be used as a trigger.

2.2.12 PostgreSQL 7 limitations

Since PostgreSQL 7.0 subselects are accepted. With that, it moves to the forefront of supported SQL dialects. Remaining issues:

1. Stored procedures cannot return multiple record sets. I.e. two gets in a stored procedure will lead to invalid code.
2. Names with spaces don't really work in functions in Postgres 7.1. It seems quoted names are not handled properly by the plpgsql parser. Version 7.3 works fine here.

See **section 2.2.11** for a discussion on calling PostgreSQL SQL code from a client.

2.2.13 SQLite limitations

SQLite was one of the easiest versions to support. Support is quite complete, but be aware that SQLite does not actually guarantee all specified constraints.

Unsupported are:

1. No support for the extended Xplain command *procedure*, because SQLite does not support stored procedures.
2. *purge* of attributes is unsuported, because SQLite does not support this.
3. Specifying a non literal *init default* will cause a column to allow null values. This is due to the fact that in SQLite triggers only fire after the constraints have been satisfied (incorrectly I think). There will be an on update trigger in a future release to check that a user does not set such attributes to null.

3 Extensions to Xplain

This chapter describes the various enhancements xplain2sql has compared to the original Xplain specification.

3.1 More representations

Beside the standard Xplain representations, see table **3.1**, xplain2sql also has support for additional representations, see table **3.2**. Support for additional representations depends on the particular sql dialect.

Representation	Explanation
Ax	Character field of length x. Translated to a VARCHAR.
B	Boolean. Either translated to a Boolean if the dialect supports it, or to a CHAR(1) with 'T' and 'F' as allowed values.
Ix	Integer field of length x.
Rx,y	Float field with x positions before the decimal point and y after.

Table 3.1 Standard Xplain representations

Representation	Explanation
C	Fixed length character field.
D	Implemented as date and time field if dialect supports it, else it's the standard date field.
M	Money field.
P	Picture field.
T	Text/memo field (unlimited compared to character field).

Table 3.2 Additional representations supported by xplain2sql

3.2 Null/not null support

By default attributes (columns) are not null. This is basically an Xplain requirement as Xplain doesn't know Nulls. There is one exception: memo and picture columns are Null by default. This has to do with storage. Even the empty string takes up a full page in case of a memo column.

To change the default Null behavior xplain2sql supports two additional keywords : *optional* and *required*.

If you want a certain column to be null you specify the *optional* keyword in front of it. So if business town is not required, you write:

type department (A3) = department name, *optional* business_town.

On the other hand, if you want a certain column to be required, use the *required* keyword. So if a employee should have a picture:

type employee (A3) = name, home_town, salary, *required* photo.

Be aware that there is no real support for optional attributes, base or type. If you say:

get employee *its* department *its* department name.

employees without a department will be silently skipped.

3.3 Quoted names

It is sometimes useful to use an Xplain keyword in a name. This can be done by quoting the name inside the ‘ and the ’ characters like:

unique index t2 *its* ‘first *index*’ = a1.

In this example *index* is a reserved name. Because it is quoted, it is no longer rejected.

Quoted names are allowed wherever a name is allowed.

3.4 Unique support

If you want a certain column to have a unique value for a table, specify the *unique* keyword in front of it. So if a department name is unique, you write:

type department (A3) = *unique* department name, business_town.

3.5 Rename column heading support

When a *get* statement is translated to the SQL **select** statement, the column names (column headings) do no longer indicate how there were derived. So in SQL columns can have duplicate names, which is not possible in Xplain. In such a case you can use the *as* keyword to rename a column. Example:

get department *its* department name *as* name.

3.6 Index support

It is possible to specify the indexes that should be created with the keyword *index*. You can either create a normal *index*, a *unique index*, a *clustered index* or a *unique clustered index*.

Example:

index department *its* firstindex =
  department name.

unique index department *its* firstindex =
  department name.

unique clustered index department *its* '*some* name' =
  business_town, department name.

3.7 Including and using other script files

For more complex environments one often wants to split Xplain statements across several files. xplain2sql has two features to support this quite well:

1. xplain2sql can *include* another Xplain script. Example:

   database test.

   .include "myinclude.ddl"

   type something (I4) = name.

   A *.include* statement reads the contents of the included file as if it was typed right there. The definition of name for example is assumed to be in `myinclude.ddl`. The definition of name and the definition of something are converted to SQL, and therefore both present in the output.
2. xplain2sql can *use* another Xplain script. Example:

   database test.

   .use "myinclude.ddl"

   type something (I4) = name.

   A *.use* statement reads the definitions of the used file, but does not generate any output for this file. In the above example the definition of something is converted to SQL, but the definition of name is not. So only the definition of something is present in the output.

Other remarks:

- Note that these dot commands start with a dot, but do not end with a dot as all Xplain commands do!
- Both included and used files have one restriction: they should not have the database statement.
- Files can be included and used in any combination, and without depth restriction. xplain2sql currently does not detect or warn for circular references!

3.8 Literal SQL

It is possible to include literal (include) SQL in an Xplain file. That Xplain file however, will not be portable to a different dialect anymore.

Literal SQL starts with the '' character and stops with the '' character. Example:

type department (A3) = department name, business_town.

{
create *unique index* idx_department on
  department("department name");
}

First a create table statement will be generated and the create index statement will be directly behind it. Note that since release 0.8.1 you can specify indexes directly with enhanced Xplain, so a more portable form would be:

type department (A3) = department name, business_town.

unique index department *its* 'idx_department' = department name.

See also **section 3.6**.

3.9 Stored procedures support

The *procedure* statement in xplain2sql makes it possible to create SQL dialect independent stored procedures. A procedure is a collection of one or more Xplain data retrieval statements. Data definition statements are not supported and usually not allowed by SQL dialects either.

The following example wraps a get statement inside the stored procedure 'retrieve names':

base short text (A60).
base long text (T).

```
type name (I9) =
  unique short text, long text.

procedure retrieve names =

get name.

end.
```

Procedures can take parameters as well. Parameters can be base or type names, with an optional role. An example of a base parameter is:

```
procedure retrieve names by text with short text =

get name
  where
    short text = ?short text.

end.
```

A parameter can be used by using a question mark followed by its name. The following example shows how a type parameter can be used to select a single instance:

```
procedure retrieve name instance with name =

get name ?name.

end.
```

It is possible to use literal SQL (see **section 3.8**) inside a procedure. The following example demonstrates this:

```
procedure 'delete unreferenced content' =

extend content with 'reference count' =
  count node
  per content.

update content "1" its 'reference count' = 1.

# delete content
#   where 'reference count' = 0.

# this code is extremely faster on TSQL
{
```

```
delete from [content]
  from [content]
  join [#content.reference count] on
    [#content.reference count].[id_content] = [content].[id_content]
  where
    [#content.reference count].[reference count] = 0
}

end.
```

The reason why the literal SQL appears in the procedure shown above is a fault in Xplain. Xplain translates the commented out *delete* to a **delete** with a subselect. But subselects in Transact-SQL perform much worse than a select with joins.

Currently, there is no support for more advanced constructs like *if-then-else* or loops. There are also no provisions to call an Xplain procedure from within Xplain. With literal SQL this can still be done, but not portably.

xplain2sql supports two special stored procedures:

1. If the procedure starts with "recompiled", the output on SQL Server includes a "with recompile" option so the plan is not cached. On other platforms the output is not changed.
2. If the stored procedure starts with "trigger", the output on PostgreSQL is a procedure that can be used as a trigger. The procedure has to conform to the PostgreSQL specification for triggers, so it has to return an explicit value, including null. On other platforms the output is not changed.

3.10 Enhanced auto-primary key support

If you use auto-primary generation, you sometimes need access to the last generated number. You can retrieve this number with *inserted* followed by the type name. Example:

```
insert customer *
  its name = '''Smith''.

value last id = inserted customer.

value last id.
```

The number is only valid, immediately after the insert. If the table has triggers, the number may become invalid even in that case.

4 Using xplain2sql in a legacy environment

xplain2sql has various features that allow it to work with existing databases, created by other tools than xplain2sql, or perhaps created by hand. This chapter discusses the features xplain2sql has to offer here.

In these environment you usually define the database again in Xplain, but perhaps only use the procedure output of Xplain in the database. So the existing database remains unmodified, but Xplain queries are executed against it.

4.1 Names with underscores

Names in legacy databases frequently contain underscores, something that Xplain sees as the end of a role. You can use quoting, an extension to Xplain, that xplain2sql implements. Quoting an identifier removes any special treatment and xplain2sql will just see it as a single unit, not a role and name for example. An identifier is quoted by prefixing it with an open single quote and suffixing it with a close single quote.

Quoting example:

type 'legacy_name' (I9) = a, b.

get 'legacy_name' *its* 'a'.

4.2 Primary keys

xplain2sql automatically generates a primary key column consisting of 'id_' followed by the table name. xplain2sql puts the 'id_' in front because some output dialects have limits on the length of identifiers, and with 'id_' in front the 'id_' gets preserved.

But in legacy systems you frequently find things like table name followed by '_id'. You can use the `-pkformat` option to generate the primary key columns in that case. Example using the Unix shell:

```
xplain2sql -pgsql -pkformat '$$s_id' test.ddl
```

This will change the default format of xplain2sql to use the `\$s_id` format. Note that because of shell escaping a double `\$\$s` is used, in other environments `\$s` might suffice.

4.3 Sequences

When the format of the primary key is different, the sequence names (used in PostgreSQL for example) are different as well. Use the `-sequenceformat` option to change the format. Example:

```
xplain2sql -pgsql -sequenceformat '$$s_seq' test.ddl
```

`-sequenceformat` takes one or two parameters. If only one `\$\$s` is present this is assumed to be the primary key name of the table. If two are present, they are replaced by the table namd and primary key name respectively.

4.4 Use and include

Use `.use` to include files without generating SQL code.

4.5 Creating views to map xplain2sql output more closely

You can create views in SQL to map tables to something that more closely resembles the output of xplain2sql. ...

4.6 Calling functions

xplain2sql allows you to call arbitrary functions using the $ syntax. Example:

```
get subcomponent its
    component,
    component its description,
    $'getcomponent_weight' (component) as 'total_weight',
    quantity,
    $'getcomponent_price' (component) as 'cost'
  where
    master_component = ?component
  per
    component its description.
```

This calls two functions, `getcomponent_weight` and `getcomponent_price`. Parameters are normal Xplain expressions and should be valid. But they are passed as is, it is up to you to make sure the number of arguments and the argument type is correct.

4.7 Injection of arbitrary SQL code in expressions

It is possible to inject arbitrary SQL between statements as explained in **section 3.8**. But xplain2sql also supports injection of SQL everywhere an Xplain expression is expected, for those times you really need to modify the output.

The following example shows a procedure that returns a certain quote. It returns a valid until date using a call to a PostgreSQL function to format it properly. However, the valid until date is the date of the quote plus a specific number of days, passed in as the interval parameter.

```
procedure document quote with quote, interval =

get quote ?quote its
    $'to_char'('date_quote' + {cast(a_interval as interval)}, "Month DD, YYYY")
as 'valid_until'.

end.
```

Xplain nor xplain2sql has any functions to format dates (nor should it), but for legacy support it is sometimes easiest to work with the flow and existing code. Natively Xplain doesn't even have a date format, although xplain2sql has a not well-specified date type.

But xplain2sql doesn't have any date functions, and it would be impossible to support them among all the dialects. PostgreSQL has lots of date functions, so this piece of code calls the PostgreSQL cast function to cast a_interval, an integer, into a number of days using PostgreSQL's interval data type. Calling cast as a function as explained in **section 4.6** wouldn't work because something as `\$cast(a_interval as interval)` is not valid Xplain. That's where the SQL injection comes into play.

With SQL injection you are really on your own. You have to make sure it works with the code Xplain generates.

5 XML output

When using xplain2sql to generate output for various dialects, the names of tables and attributes in those dialects can differ. For example, certain dialects require quoted identifiers when they have been declared with quotes. Other dialects don't support quoting at all, or have non-ANSI quoting conventions. This makes it hard to write portable code. With xplain2sql's XML output, it becomes possible to access the generated table and column names without introducing subtle inconsistencies in the source.

5.1 xplain2sql.xml file

xplain2sql will write an XML description of the generated SQL when supplied with the `-xml` option. The XML file is called `xplain2sql.xml`. This XML file contains the bases, types and procedures that are generated. This XML file can be used by XSLT scripts, or other tools, to generate middleware code, or just to extract the proper names to use for creating on-the-fly SQL.

Take for example the following Xplain:

> *base* customer name (A40).
> *base* address (A40).
>
> *type* customer (I4) = customer name, address.

When xplain2sql is called with the -xml flag:

```
xplain2sql -xml sample.ddl
```

The generated XML looks like this (it is slightly edited for readability):

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<sql>
 <table xplainName="customer" xplainDomain="(I4)"
     sqlName='"customer"'
     identifier="customer"
     sqlNameAsEiffelString='%"customer%"'
     sqlNameAsCString='\"customer\""'>
  <column xplainName="customer" xplainDomain="(I4)"
     sqlName='"id_customer"' sqlType="smallint"
     identifier="customer"
     sqlNameAsEiffelString='%"id_customer%"'
     sqlNameAsCString='\"id_customer\"'
     init="none"/>
  <column xplainName="customer name" xplainDomain="(V40)"
     sqlName='"customer name"' sqlType="character varying(40)"
```

```
        identifier="customer_name"
        sqlNameAsEiffelString='%"customer name%"'
        sqlNameAsCString='\"customer name\"'
        init="none"/>
    <column xplainName="address" xplainDomain="(V40)"
        sqlName='"address"' sqlType="character varying(40)"
        identifier="address"
        sqlNameAsEiffelString='%"address%"'
        sqlNameAsCString='\"address\"'
        init="none"/>
  </table>
</sql>
```

For every type a <table> tag is generated. In this tag there are one or more <column> tags.

Every XML tag contains three things:

1. The Xplain name and type.
2. The SQL name and type (if applicable).
   If the SQL name is quoted, the name includes those quotes as many SQL dialects regard quotes as part of the name.
3. Names useful in programming languages which try to access the database.
   The 'identifier' attribute can be used as the name of a variable for example. Any spaces occurring in 'xplainName' are replaced by underscores.
   'sqlNameAsEiffelString' and 'sqlNameAsCString' contain the name as known in the SQL dialect. They contain the proper quoting for Eiffel and C respectively. Usually XSLT scripts are used to transform this XML and it's a real pain to detect and do proper quoting there, so that's why xplain2sql already does these things.

Another example might clarify the quoting. Let's translate the Xplain example to Transact-SQL.

```
xplain2sql -xml -tsql sample.ddl
```

The file `xplain2sql.xml` now looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<sql>
  <table xplainName="customer" xplainDomain="(I4)"
        sqlName="[customer]"
        identifier="customer"
        sqlNameAsEiffelString="[customer]"
        sqlNameAsCString="[customer]"
        spDelete="[sp_Deletecustomer]"
        spDeleteAsEiffelString="[sp_Deletecustomer]"
        spDeleteAsCString="[sp_Deletecustomer]"
        spInsert="[sp_Insertcustomer]"
```

```
      spInsertAsEiffelString="[sp_Insertcustomer]"
      spInsertAsCString="[sp_Insertcustomer]"
      spUpdate="[sp_Updatecustomer]"
      spUpdateAsEiffelString="[sp_Updatecustomer]"
      spUpdateAsCString="[sp_Updatecustomer]">
    <column xplainName="customer" xplainDomain="(I4)"
      sqlName="[id_customer]" sqlType="smallint"
      identifier="customer"
      sqlNameAsEiffelString="[id_customer]"
      sqlNameAsCString="[id_customer]"
      init="none"/>
    <column xplainName="customer name" xplainDomain="(V40)"
      sqlName="[customer name]"
      sqlType="[Tcustomer_name]"
      identifier="customer_name"
      sqlNameAsEiffelString="[customer name]"
      sqlNameAsCString="[customer name]"
      init="none"/>
    <column xplainName="address" xplainDomain="(V40)"
      sqlName="[address]" sqlType="[Taddress]"
      identifier="address"
      sqlNameAsEiffelString="[address]"
      sqlNameAsCString="[address]"
      init="none"/>
  </table>
</sql>
```

5.2 Processing the XML file with XSLT

xplain2sql comes with some XSLT scripts that turn the generated XML file into Delphi or Eiffel classes. For example the `ecli_stored_procedure.xsl` generates an Eiffel class that allows one to call a stored procedure and get access to the parameters of that procedure and the resulting output.

Usage:

```
Xalan \
  -o mw_retrieve_currencies.e \
  -p procedureName "'retrieve currencies'" \
  xplain2sql.xml ecli_stored_procedure.xsl
```

The name of the procedure for which a class is generated is passed as parameter. Note that the stylesheet `ecli_stored_procedure.xsl` is an example. It is fully functional, but nothing should stop you from adapting and refining it to suit your needs. That's why xplain2sql generates XML in order to make such customisations possible.

xplain2sql also comes with an example Makefile script, `make_ecli_stored_procedures.xsl`, that generates a makefile that will call `ecli_stored_procedure.xsl` for every stored procedure.

6 Implementation

Xplain is written in Eiffel. This chapter explains how xplain2sql was designed and implemented.

In figure **6.1** the flow of data through xplain2sql is depicted. In goes a file with Xplain commands, out comes SQL code.

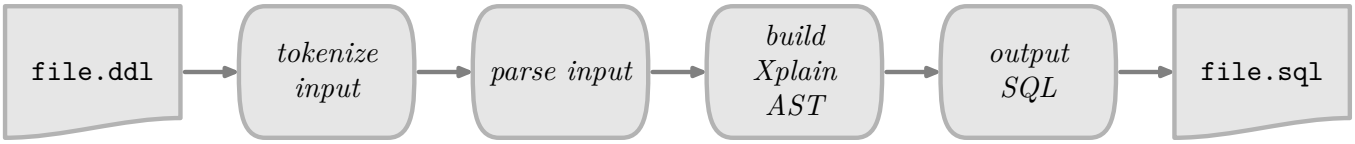

Figure 6.1 High-level overview of xplain2sql workings

The actual behavior is more cyclic. The input file is not first all tokenized, and next parsed. No, the parser tokenizes as much as is needed for a certain construct, and next generates SQL code for that construct. Than it starts tokenizing, parsing and outputting again.

xplain2sql is a prime example and implementation of the Builder pattern (see ??). The **Director** participant is implemented by the tokenizer and parser in XPLAIN_SCANNER and XPLAIN_PARSER. The **Builder** participant is the class SQL_GENERATOR.

In the following sections the tokenizing, parsing and sql generation are discussed in detail, by showing how the Builder pattern is used.

6.1 Tokenizing

The tokenizer is the first stage of the Director participant. The tokenizer class is written with Lex, see `xplain_scanner.l`. This file is converted by `gelex` to the Eiffel class XPLAIN_SCANNER. The responsibility of the tokenizer is to output tokens.

However, the tokenizer knows a bit about sql generation, because of inline sql and sql comments.

A '`{`' starts literal sql. The tokenizer doesn't bother the parser with this (literal sql can occur anywhere currently) and writes the sql code straight to the output file. Literal sql of course makes an Xplain file not portable. Maybe there should be support for sql dialect specific literal sql with something like '`interbase{ ... }`'.

A '`-`' starts a one line comment that is also written to the output file. As not all sql dialects support one line comments, the tokenizer calls the sqlgenerator (see

section **6.3**) directly so it can transform a one line comment to a multi line like comment. Multi line comments seem to be supported by all sql dialects, so the tokenizer writes these straight to the output file.

There's one problem that needs to be solved someday. The default case is matched, but it shouldn't.

6.2 Parsing

The parser is the second stage of the Director participant. The parser is implemented in typexplain_parser.y. This is Yacc (with the Bison extensions `%type`), which is converted by `geyacc` to an Eiffel class. The responsibility of the parser is to parse Xplain input and to give only correct Xplain to the sql generation routines.

The parser doesn't build a full Abstract Syntax Tree (AST), but only partial ones. As soon as a construct has been parsed, the sqlgenerator is called. In terms of the Builder pattern, the **Products** that the Builder creates are XPLAIN_... classes.

Within the Directory, the Interpreter pattern (see (Gamma et al., 1995)) is used a lot. Simple examples are the parsing of representations, or domain restriction.

An advanced application of the Interpreter pattern is the expression parsing part, see the partial expression inheritance hierarchy in figure **6.2**. The **Abstract-Expression** participant is implemented by XPLAIN_EXPRESSION. An example of **TerminalExpression** is XPLAIN_STRING_EXPRESSION. An example of **NonterminalExpression** is XPLAIN_INFIX_EXPRESSION.

The **Context** participant is XPLAIN_UNIVERSE. This class is implemented as a Singleton (see (Gamma et al., 1995)). This class holds all objects which have a global scope (see figure **6.3**. Only XPLAIN_EXTENSION is not stored in XPLAIN_UNIVERSE because the extension scope is only known within a type.

There are two **Client** participants: the Client XPLAIN_PARSER builds the AST, while the Client SQL_GENERATOR (see figure **6.4**) is given the AST.

The AbstractExpression XPLAIN_EXPRESSION has several Interpret methods. Method sqlvalue returns the sql code for that particular expression as converted to be used in a sql select statement. sqlinitvalue does the same, but is specifically for init statements. See section **6.3** for how the sql code is obtained. AS usual, the parser also embellishes the partial AST with type information. For this conversion, this could be kept quite simple. An example of this is found when parsing a name (or names separated by *its*) in an expression. Example:

> *get* employee *its* department *its* business_town.

The 'department *its* business_town' is an example of an XPLAIN_ATTRIBUTE-_EXPRESSION. But when parsing, we don't know this beforehand of course. So the parser initially accepts anything a user has typed. So it would initially accept 'its not_an its attribute'. The name or names are stored in a linked

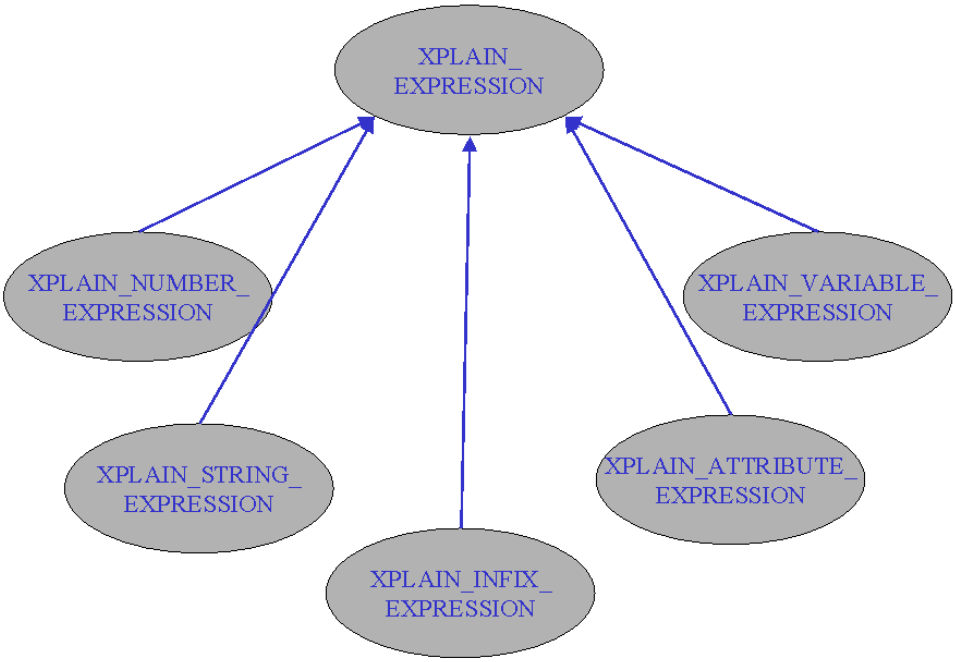

Figure 6.2 Partial Xplain expression inheritance hierarchy

list of XPLAIN_ATTRIBUTE_NAMEs. There after, every name in this linked list is supplied with with its type in get_object_if_valid_tree, a routine in XPLAIN_PARSER. This routine returns the object for the first name in this linked list of node. We ask this object to create the expression for us. If the object is a variable, it returns an XPLAIN_VARIABLE_EXPRESSION. If it's a value, it returns an XPLAIN_VALUE_EXPRESSION. If it's an attribute, it returns an XPLAIN_ATTRIBUTE_EXPRESSION.

The parser (and all other parts of xplain2sql) use only one type of data structure: a single linked list. This could be optimized a bit, for example in XPLAIN_UNIVERSE to make lookups a bit faster, but works well. The generic linked list is defined in XPLAIN_NODE. For each kind of linked list a separate class is defined. Examples are JOIN_NODE, XPLAIN_ATTRIBUTE_NODE or XPLAIN_ATTRIBUTE_NAME_NODE.

6.3 SQL generator

The SQL generator is the Builder participant of the Builder pattern and is implemented in the class SQL_GENERATOR. The actual sql dialect generation is done by the **ConcreteBuilder** participants, namely the descendants of

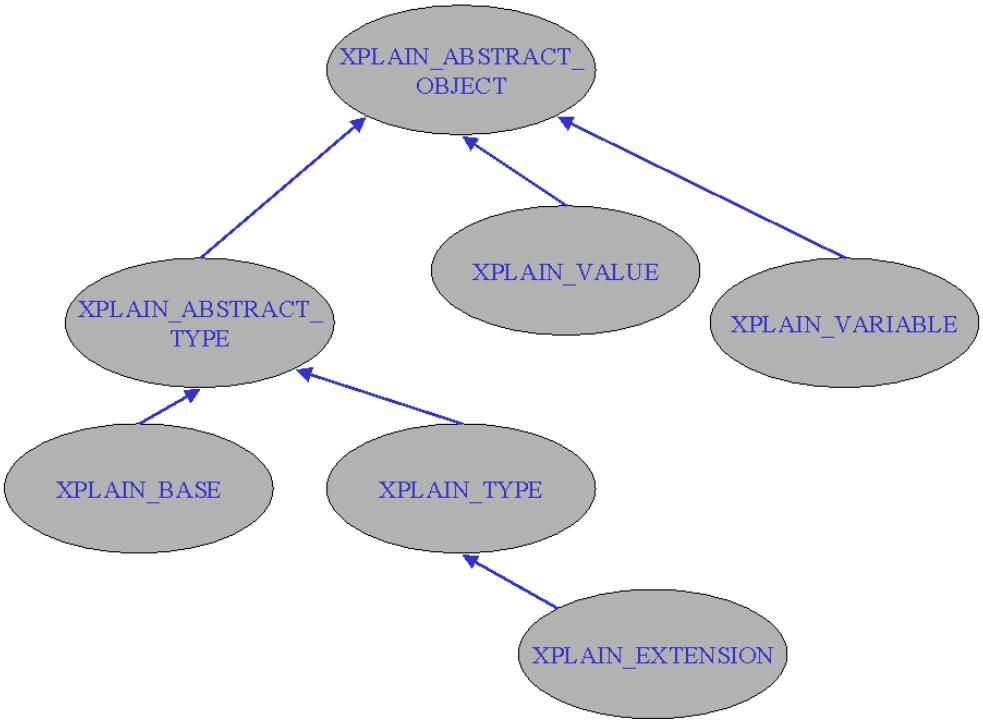

Figure 6.3 Xplain objects inheritance hierarchy

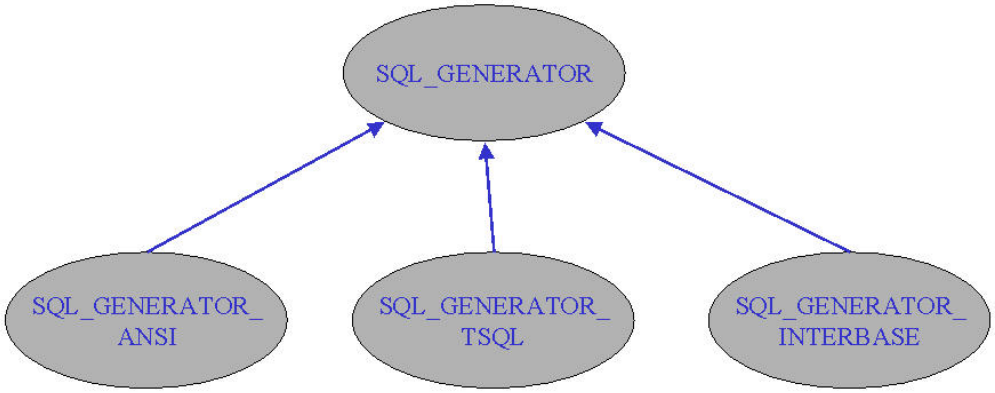

Figure 6.4 SQL Generator inheritance hierarchy

SQL_GENERATOR, see figure **6.4**. SQL_GENERATOR is a deferred class, doing the lowest common denominator stuff, descendants fill in the details.

As suggested by BM in (Meyer, 1997) (chapter 24.3) the handle technique is used to implement the SQL_GENERATOR class. A different approach could have been to add all sql generating code to XPLAIN_PARSER and inherit from

XPLAIN_PARSER to get the various dialects. But the handle approach is preferred. So XPLAIN_PARSER doesn't know a single bit about generation sql code, it just calls SQL_GENERATOR through its sqlgenerator feature at appropriate times. This approach looks like the Bridge pattern (see (Gamma et al., 1995)), but it would be a real Bridge if the interface between XPLAIN_PARSER and SQL_GENERATOR was abstracted, which isn't the case.

The code generation within SQL_GENERATOR itself is divided in two parts: the write_ methods are the only methods called by XPLAIN_PARSER. The write_XXXX methods check if certain options are enabled or supported. If so, they call the corresponding create_XXXX which outputs the sql code.

The relation between the Product (for example XPLAIN_TYPE) and the Builder is complex if you look at how and when they call each other, but there responsibilities are clear. the Product does know nothing about sql generator, it only knows about Xplain representations. The Builder itself knows a lot about Product, but to generate certain code, it calls Product. Product next calls the Builder back at some other method to get the correct sql.

A good example is the creation of domains or user defined data types. In Xplain you define a domain with:

> *base* name (A30).

The corresponding sql code is something like:

```
create domain name as character(30).
```

The Builder accepts the Product base (XPLAIN_BASE). To output the representation of a domain, it asks the representation property of XPLAIN_BASE to give it the correct sql code. Depending on the representation, they call SQL_GENERATOR (or its descendants) back in methods starting with the name sqldatatype_XXXX. In this particular case sqldatatype_char is called. This happens continually. Creating check constraints is done by asking the Product domain restriction (XPLAIN_DOMAIN_RESTRICTION) for its sqlcolumnconstraint. The actual domain restriction will call SQL_GENERATOR back in its sqlcheck_XXXX methods.

6.4 Other details

In the Builder pattern a separate client is discussed which creates the Directory (XPLAIN_PARSER) and configures it with the desired Builder object (SQL_GENERATOR). There is no separate client with xplain2sql. That part is handled by the execute method of the XPLAIN_PARSER itself.

7 What is Xplain

Xplain is a non-procedural data definition and manipulation language invented by Johan ter Bekke and completed in 1983. xplain2sql is based on the most recent definitions of the language, see for example (ter Bekke, 1992). Xplain also refers to the Xplain product, a full implementation of a semantic database, but in this document Xplain always refers to the language.

This chapter gives a short introduction to Xplain, with the goal to get people who don't know Xplain getting interested to learn more. A good start is `http://mmdb.kbs.twi.tudelft.nl/terBekke.html`. All material in this section has been taken without getting permission :-)

7.1 The Xplain book

For English readers, the best introduction to Xplain is given in (ter Bekke, 1992). I absolutely recommend this book, because it not only introduces Xplain, but it gives fundamental advise about database design and modeling. The semantic way of thinking has helped me a lot to tackle large design problems and create excellent database designs.

Abstract of this book (taken from Ter Bekke's website `http://mmdb.kbs.twi.tudelft.nl/terBekke.html`):

> The book explains the fundamental concepts and general principles of data modeling, with practical cases to illustrate the theory where appropriate. Recent developments in the database area have been included. The book is organized in four main parts:
>
> - *Overview of the discipline*, including an assessment of the relational theory. An overview of seven modern data modeling approaches is also presented in this part.
> - *Fundamentals of data modeling*, introducing semantic concepts leading to proper object modeling.
> - *Data modeling*, illustrated with numerous practical examples. Conversion into suitable traditional models (including relational), by applying just a few simple rules, makes the collection of data and query structures reliable and easier to understand.
> - *Case studies*. Semantic data modeling is illustrated with three large cases. They illustrate data modeling in complex situations and the problem of formulating queries in practical environments.
>
> Relational systems have become widely accepted the last few years. However, many pitfalls have also been discovered in the relational theory. This book presents an in-depth analysis of the problems and offers a deeper understanding. By putting emphasis on the semantic structure of a database, reliable solutions are created for both data modeling and data manipulation problems. The theory is based on both theoretical and practical research. It is illustrated with many examples and exercises. Semantic Data Modeling offers a sound basis for an education in modern data modeling techniques.

7.2 Short introduction to Xplain

In (ter Bekke, 1996) a short introduction to Xplain is given, before showing how Xplain can be used to model successive events, version management in that particular case.

7.2.1 Introduction

The semantic abstractions aggregation and generalization appeared for the first time in the database literature in a series of papers by Smith and Smith ((Smith and Smith, 1977a), (Smith and Smith, 1977b), (Smith, 1978)). These abstractions are considered to be suitable for modeling complex situations in which different types of relationships occur. Because of their nature, they are especially useful for modeling hierarchical relationships. Many examples can be found in literature. Considering these application areas, only minor differences can be discovered between semantic data models and other data models (e.g. relational and entity-relationship data models). This is not a surprise:earlier hierarchical and network data models already enabled us to find solutions for such situations.

The advantages of generalization appear especially in situations with many irregularities, alternatives or exceptions. These new aspects were not fully covered by other data models. It gave semantic models a clear advantage over other existing data models.

In this paper the advantages of semantic abstractions are further extended. Although semantic concepts enable us to create data models for ordering and sequencing, they are hardly mentioned in literature. This is also caused by the difficulties data models have with modeling time aspects. It is also illustrated with the phrases *is-part-of* and *is-a*, in other data models often general need for this extended functionality appeared also in papers discussing characteristics that must be satisfied by the next generation of database systems (literature, see original article).

Sequencing is also important in the area of object oriented databases. Examples can be found in literature (literature, see original article). However, often extended entity-relationship diagrams, flow diagrams or event-condition-action diagrams are used for the purpose. These specifications are far from complete; they must be accompanied with many informal procedural descriptions. These specifications/descriptions are inadequate for development of the required software.

Several commercial products support some form of version management (for example: Objectivity/DB,Itasca, ObjectStore, Ontos and Versant). However, a standard set of features for version management is lacking (literature, see original article).

This paper presents extensions in the usage of semantic abstractions. Both aggregation and generalization can be used for modeling ordering and sequencing. Because time is involved, also the phrases *was-part-of* and *was-a* should be used

instead of only the phrases is-part-of and is-a. The new opportunities will be illustrated with several practical examples.

The resulting high-level specifications can be used for standardization purposes. They can by implemented in any programming environment (e.g. C or C++) or database environment (e.g. relational or object oriented). For a better understanding of the opportunities, first a short introduction to the underlying semantic concepts is given.

7.2.2 Abstractions

This section contains a global overview of the concepts for semantic data modeling using well-known examples. Each object will be visualized explicitly by clearly distinguishing between identification and descriptive properties. The resulting data models gain in semantic contents as a consequence, while ambiguities and contradictions in the specification are avoided. Only three fundamental abstraction types with clear graphical equivalences in the structural diagrams are required to guarantee inherent semantic integrity. They make use of the fundamental *type-attribute relationship*.

The real world is described by considering types of relevant objects, a type being defined as a fundamental notion. The abstraction leading to a type is called **classification**. The examples (i.e. instances) occurring in a database and required for the recognition of a type are purely illustrations of the concept. The type is not being defined hereby. Types are represented by rectangles in diagrams, see figure **7.1**. The counterpart of classification is called **instantiation**.

Figure 7.1 Classification

Aggregation is defined as the collection of a certain number of types in a unit, which in itself can be regarded as a new type (note the analogy with the mathematical set concept). A type occurring in an aggregation is called an attribute of the new type.

Aggregation allows view independence: we can discuss the obtained type (possibly as a property) without referring to the underlying attributes. By applying this principle repeatedly, a hierarchy of types can be set up. An example is given in figure **7.2**. Normally the hierarchy contains only aggregated types. Aggregation is indicated by a line connecting the centers of two facing rectangle sides, while the aggregate type is (according to its definition) placed above its attributes. Of course, aggregation also has its counterpart: the description of a type as a set of certain attributes is called **decomposition**.

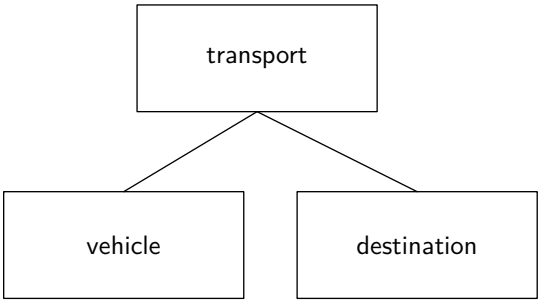

Figure 7.2 Aggregation hierarchy

A type is defined by listing its attributes, so we could have the following type definitions:

type transport = vehicle, destination, delivery_date, cargo.
type vehicle = manufacturer, model, price, fuel, construction_year.
type destination = client_name, address, city, telephone number.

An example to illustrate the database contents is in **table 7.1**:

transport		vehicle	destination	delivery_date	cargo
t1		v1	d2	19961206	paper
t2		v3	d4	19961207	milk

Table 7.1 Database contents

Type definitions carry semantics; they contain the essential properties (e.g. uniqueness of the identifications t1 and t2 in the table above) and essential relationships (the related vehicles v1 and v3 and the related destinations d2 and d4 must occur in related tables). Aggregation can be described using the verb to have. According to the above type definition, a vehicle has a manufacturer, model, price, fuel and construction_year. Identifications are properties denoted by type names (see table 1 above). This interpretation implies singular identifications. Attributes (not types!) may contain roles. An example is 'construction_year' related to type 'year'. Roles are separated from the type by an underscore. Spaces are irrelevant in type definitions.

The third kind of abstraction, important to conceptual models, is called **generalization**; it is defined here as the recognition of similar attributes from various types and combining these in a new type (note the analogy with the intersection operation from mathematical set theory). We can equally discuss the new type without mentioning the underlying attributes, and it can in itself again serve as a property (i.e. it allows view independence).

Example: consider manufacturer, model, price, fuel, construction_year, cabin, weight, wheels, power and coupling. The corresponding type is truck. Consider, in

addition, manufacturer, model ,price, fuel, construction_year, chassis, seats and color, where the type might be car.

The common attributes of the two types are: manufacturer, model, price, fuel, construction_year. If required, these attributes result in a new type "vehicle", which may be regarded as the generalization of truck and car. Generalization can be represented in abstraction hierarchies, as we have seen in the case of aggregation. This is shown in figure **7.3**.

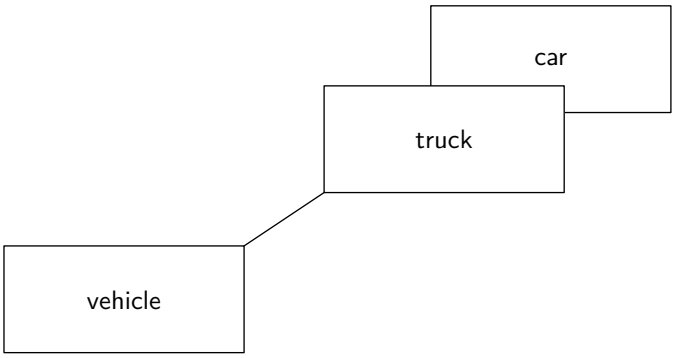

Figure 7.3 Generalization hierarchy

In abstraction hierarchies, generalization is schematically represented by a line connecting facing corners of rectangles, the generalized type being placed below the specialized ones. Generalization's counterpart (i.e. the union of attributes from different types) is called **specialization**.

In figure **7.3** we placed truck and car one behind the other, while only one line connects to vehicle. This is the usual representation of disjoint specializations — i.e. a vehicle can be either a truck or a car, but not both. The combination of a group of disjoint specializations is called a block, so truck and car constitute a block. Not all vehicles need to be specialized; an example would be a motorcycle occurring only as instance of vehicle.

The generalization, together with the attributes to be added to it, is (by definition of the concept) described in the type definition of the specialization. So the type definitions are:

type vehicle = manufacturer, model, price, fuel, construction_year.
type truck = [vehicle], cabin, weight, wheels, power, coupling.
type car = [vehicle], chassis, seats, doors.

An example of the database contents is given in **table 7.1**. This structure imposes uniqueness of attributes related to generalizations. Besides that, values for these attributes may occur only once in a block (i.e. v1 and v3 may not occur as values in the corresponding truck table).

Generalization is commonly associated with the verb *to be*. According to the above type definitions, a truck is a vehicle with cabin, weight, wheels, power and

coupling, while a car *is a* vehicle with chassis, seats and color. The introduction of new identifications for specializations (e.g. c1 and c2 above) makes generalization hierarchies *non-transitive dependent*.

The introductory definitions of aggregation and generalization above have already clearly demonstrated the hierarchical character of these abstractions. This will be elaborated in the following sections with an emphasis on ordering aspects.

7.3 Employee and department

In chapter 6 of (ter Bekke, 1992), a simple Xplain data model is given, see figure **7.4**. It defines the types department and employee. Every employee belongs to a department, and a department can have 0 or more employees.

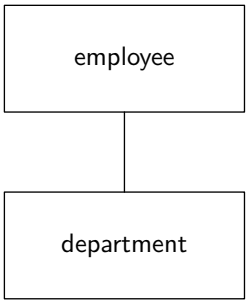

Figure 7.4 Employee hierarchy

The database definition is:

base department name (A30).
base town (A30).
base name (A30).
base salary (R9,2).

type department (A3) = department name, business_town.
type employee (A3) = name, home_town, department, salary.

We can now ask certain questions (all taken from (ter Bekke, 1992)) and demonstrate a feature of Xplain:

1. Select data of the employee with the identification E3.

   get employee "E3".

2. Select employees living in Guilding.

   get employee *its* name, department
   where home_town = "Guilding".

3. Select commuters.

```
get employee its name, home_town, department
    where home_town <> department its business_town.
```

4. How many employees work in Guilding?

```
get count employee
    where department its business_town = "Guilding".
```

5. What is the sum of all salaries?

```
get total employee its salary.
```

6. What is the highest salary?

```
get max employee its salary.
```

7. Are there any employees earning more than 50.000?

```
get any employee
    where salary > 50000.
```

8. Select the name of an arbitrary employee in the Purchase department.

```
get some employee its name
    where department its department name = "Purchase".
```

9. Provide an overview of the departments, including the number of employees.

```
extend department with numberofemployees=
    count employee
    per department.

get department its
    department name,
    business_town,
    numberofemployees.
```

10. Select departments with more than 100 employees.

```
get department its
    department name,
    business_town
  where
    numberofemployees > 100.
```

11. Find the number of departments with more than 100 employees.

```
get count department
  where numberofemployees > 100.
```

12. Which department has the most employees?

```
value maximum =
    max department its numberofemployees.

get department its
    department name,
    business_town
  where
    numberofemployees = maximum.
```

All these questions have two basic forms: if you go down in an Xplain model (from employee to department), you use the *get* statement. But if you go up in an Xplain data model (from department to employee), you first use the *extend* statement and next a *get*.

And as you see, all queries are clear and easy to understand. The corresponding SQL code is far harder to read.

7.4 Case study: Bank

In (ter Bekke, 1992) three case studies are given. Parts of the first case study are given here. Also there is an example file supplied with the xplain2sql distribution, `bank.ddl` which contains all data definition and data manipulation presented in this case study.

7.4.1 Case description

Consider a banking organization consisting of a few regional head offices and numerous branches, each branch reporting to one of the head offices. Customers can have various accounts with one of the offices: saving accounts (with conditions), current accounts (with cheque cards, Eurocheques, credit limits), business accounts (with credit commissions, credit limits) and mortgage accounts (with collection indications, redemption methods, redemption accounts). The following account information is recorded: type, currency, sum, transaction date and interest. Customers may deposit shares and bonds with a branch for a fee. Deposits are linked to customers' accounts. The recorded share information is name, number and nominal value, while for bonds this name, nominal value, starting number and end number. Customers can have multiple series of one bond in one deposit.

7.4.2 Assignment

Provide a semantic model for above environment and indicate the relevant attributes. Also provide the abstraction hierarchy. Indicate some applications of the model by using a query language.

7.4.3 Conceptual design

Although the relationships between head offices and branches have been described precisely, many designers find them difficult to interpret. The model should obviously contain a type defining the information about offices, because customers have accounts there. The difference between head offices and branches is apparent from the fact that branches belong to head offices; they are disjunct types in a block. There are, however, additional relationships between the offices:

type office =..., *case*
type head office = [office], ...
type branch = [office], ..., head office.

Accounts play a paramount role in the remainder of the case study. The specializations of the type account are all obvious. They have common and separate attributes and the individual types are disjunct. The attribute redemption account is the only potential difficulty in the definition of the mortgage account. The situation is comparable to the relationship between head offices and branches. If an individual mortgage account is also related to another instance of the type account, the second account could again be of the type mortgage account, which is undesirable.

The relationship between the mortgage and redemption accounts is therefore at the specialization level. Considering the four alternatives (mortgage, business, current and saving accounts), the only relevant relationship appears to be the one with the current account. We thus have the following structure:

type account = ..., *case*
type mortgage account = [account], ..., current account
type saving account = [account], ...
type business account = [account], ...
type current account = [account], ...

There can be little doubt about the relationship between account and office, because each account is related to a customer and held by one office and various accounts can be related to one customer.

One of the questions which can be considered during the design of the account concept is whether or not name, address, town and other specific customer data should be included in a separate data structure. In this case study we prefer to register customers. One of the best arguments is that banks must be able to

relate debit and credit amounts in the various accounts of any one customer. This implies a relationship between the various accounts a customer may have. We will introduce an external personal registration (e.g. fiscal identification, passport number and Chamber of Commerce registration number), and suggest regular checks for non-account holders in addition. The following type definitions therefore apply:

type account = holder, ...
type holder = ..., office, identification.

The final section of the example describes the relationships between shares, bonds, deposits and accounts. Of these, deposits are not physical items in banks - witness statements like:

- deposits are linked to accounts, which suggests an aggregation link between deposits and accounts;
- multiple series of one stock in a deposit, suggesting an aggregation link between bonds and deposits.

The noun deposits stems from the verb to deposit, which really means "to link stocks to accounts". Deposits themselves can therefore not be associated with properties. Thus we arrive at aggregation of stocks in relation to accounts, where stocks consist of shares or bonds. Obvious specializations lead us to:

type stock = account, ..., *case*
type share = [stock], ...
type bond = [stock], ...

We now have defined all relationships, and the addition of the properties specified in the description results in the following total model:

type currency (I4) = currency code, exchange_rate.
type office (I6) = address, postalcode, town,
telephone number, postal_giro, bank,
mycase.
type head office (I6) = [office], region.
type branch (I6) = [office], head office.
type holder (I9) = name, address, postalcode, town, office,
identification.
type account (I9) = holder, balance, transaction_date,
currency, mycase.
type current account (I9) = [account], dr_interest, cr_interest,
cr_limit, cheque card, Eurocheque.
type mortgage account (I9) = [account], current account, interest,
redemption method, collection_indication.
type saving account (I9) = [account], condition, interest.
type business account (I9) = [account], dr_interest, cr_interest,

	cr_commission, cr_limit.
type stock (I9)	= account, nominal_stockvalue, purchase_stockvalue, purchase_date, mycase.
type share (I9)	= [stock], number.
type bond (I9)	= [stock], start_number, last_number.

The above is summarized in the abstraction hierarchy in figure **7.5**. This figure is not entirely correct as there is a relation from branch to head office (every branch has a head office), and from mortgage account to current account (every mortgage account has a current account). In the next revision of this document, the correct figure will be presented (when I know how to do this with MetaPost).

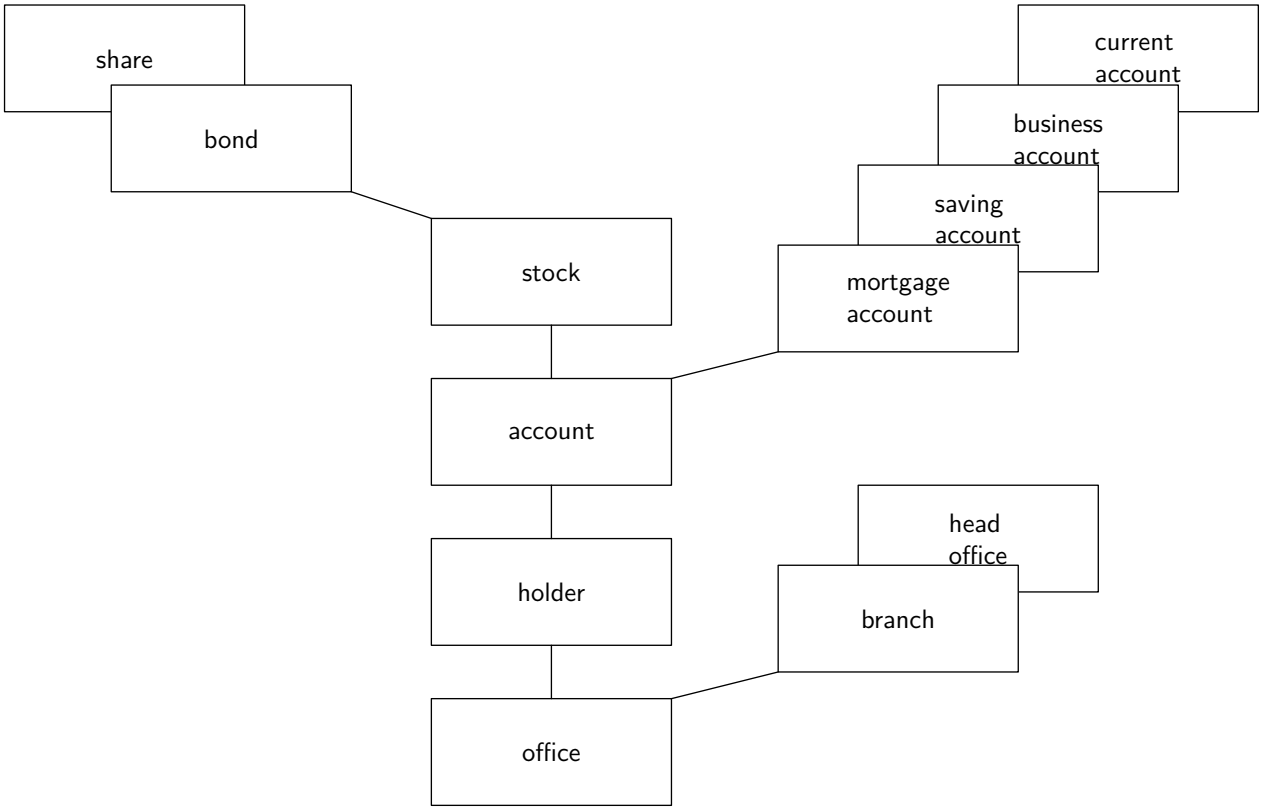

Figure 7.5 Bank abstraction hierarchy

Index

References

ter Bekke, J. (1992). *Semantic data modeling*. Prentice Hall International.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns*. Boston, MA: Addison-Wesley.

Meyer, B. (1997). *Object-Oriented Software Construction*. 2nd edition Addison Wesley.

ter Bekke, J. (1996). Semantic modeling of successive events applied to version management. In *International Symposium on Cooperative Database Systems for Advanced Applications, volume 1* .

Smith, J. and Smith, D. (1977a). Database abstractions: aggregation. In *Communications of the ACM* , pages 405–413.

Smith, J. and Smith, D. (1977b). Database abstractions: aggregation and generalization. In *ACM Transactions on Database Systems* , pages 105–133.

Smith, J. (1978). Principals of database conceptual design. In *Proceedings NYU Symposium on database design* , pages 35–49.