INF312 mandatory exercise

Igor V. Rafienko (igorr@ifi.uio.no)

Motivation and general information

The midterm project in INF312 is aimed at improving your understanding of concepts covered in the lectures. Specifically, the exercise concentrates on heterogenous systems and it emphasizes the relative merits of the relational and object-oriented approaches to database design. This paper gives a short introduction into the structure of the relational and object-oriented databases used in the exercise.

The basis for this project is a movie database, presented in both relational and object-oriented fashion. The task presented to you consists of modelling a database, running some queries (using SQL and either C++ or Java) and writing a small report explaining the results obtained.

The data in question are a fragment of the Internet Movie Database (imdb, [1]). IMDB has a lot of information on approximately 300000 movies and 800000 people. Ifi does not have a sufficiently powerful database server to run such a large database. In INF312, we will use a far smaller fragment – about eight percent of the original size. The fragment contains all movies with a connection to France. Both relational and object-oriented databases contain the same data, but they are naturally structured in different ways.

The relational database runs on the Sybase ASE 11.9.2 server `gand`. The database is rather small by industrial standards. It contains 11 tables, occupying around 50 MB for both data and indices. The largest table has around 230000 rows.

The object-oriented database runs on the ObjectStore 6 sp8 server `mjollnir`. The database size is around 100 MB for both data and indices. There is no concept directly corresponding to a table in ObjectStore, but there are two data structures representing the movies and the people, containing around 15000 and 80000 entries respectively.

At the time of this writing it is yet uncertain whether we will migrate from the Sybase server to an Oracle server. However, at least for the Java interface, the database backend has no bearing on the complexity of the tasks involved.

This exercise is due on October the 30th.

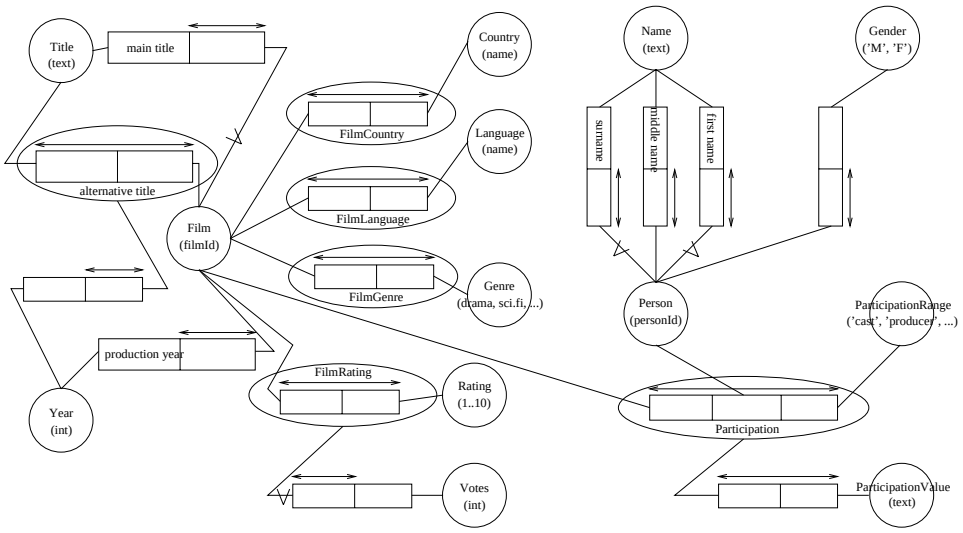

Figure 1: NIAM diagram for the relation database

Relational design

The database is centered around three entity types – films, people and participation, linking the first two together. You can see the NIAM diagram in figure 1.

Each concept in a NIAM model translates into its own table during the realization stage. Naturally, not all concepts always warrant a table, and in this particular case `Country`, `Language`, `Rating`, `Votes`, `Year`, `Title`, `Name` and `Gender` have been suppressed. You can see the resulting tables in figure 2[1].

The tables `Participation` and `ParticipationValue` merit a more in-depth description. These tables represent the connections between the films and the people. These connections are categorized (`ParticipationRange` contains the list of categories), and `Participation` registers the nature of a person's participation in a movie.

For instance, these rows describe Luc Besson's participation in "The Fifth Element":

```
[7] gand.inf212db: select * from Participation
[7] gand.inf212db: where personId = 26513 and filmId = 76914 ;
 pid           personId      partName         filmId
 ------------- ------------- ---------------- -------------
       2278671         26513 director                76914
       2749250         26513 writing credits         76914
```

[1]Notice that `grep` is not a part of an SQL dialect, but refers to the standard UNIX command `grep`. sqsh ([2]) provides a number of useful features such as the traditional unix shell pipe mechanism

```
[1] gand.inf212db: sp_help
[1] gand.inf212db: go | grep -E 'user table'
 AlternativeFilmTitle           dbo      user table
 Film                           dbo      user table
 FilmCountry                    dbo      user table
 FilmGenre                      dbo      user table
 FilmLanguage                   dbo      user table
 FilmRating                     dbo      user table
 Genre                          dbo      user table
 Participation                  dbo      user table
 ParticipationRange             dbo      user table
 ParticipationValue             dbo      user table
 Person                         dbo      user table
```

Figure 2: Tables of the relational database

`pid` is an artificially created attribute, to facilitate joins between `Participation` and `ParticipationValue` (This attribute is superfluous, since <`personId`, `partName`, `filmId`> constitutes a candidate key. But it is easier to join on one attribute rather than three, and we duplicate less information in this fashion).

`ParticipationValue` contains the values describing a participation. Most commonly these values are roles played by actors in a movie (i.e. a role name for a given participation of type "cast"):

```
[4] gand.inf212db: select * from ParticipationValue
[4] gand.inf212db: where pid = 1846849 ;
 pid          value
 -------------  ----------------------
       1846849  Leeloo
(1 row affected)
```

... shows the role played by Milla Jovovich in "The Fifth Element"[2].

You are advised to experiment a little with different tables to get a feeling of how different attributes and tables relate to each other. Run a set of simple queries to examine the entries for any movies that you are familiar with.

Examples

Let us study an example showing all the roles played by Milla Jovovich with the corresponding movies. We will need to perform join operations between `Person` (to find Milla Jovovich's `personId`), `Participation` (to find her roles) and `ParticipationValue` (to find the name of the roles):

[2]The `pid` 1846849 is obtained by joining `Person` and `Participation` tables.

```
[2] gand.inf212db: select pv.value, f.mainTitle
[2] gand.inf212db: from ParticipationValue pv, Participation pa,
[2] gand.inf212db:      Person p, Film f
[2] gand.inf212db: where p.surName = 'Jovovich' and
[2] gand.inf212db:       p.firstName = 'Milla' and
[2] gand.inf212db:       pa.personId = p.personId and
[2] gand.inf212db:       pa.filmId = f.filmId and
[2] gand.inf212db:       pa.pid = pv.pid and
[2] gand.inf212db: go -m pretty
+====================+==========================================+
| value              | mainTitle                                |
+====================+==========================================+
| Mildred Harris     | Chaplin                                  |
+--------------------+------------------------------------------+
| Lucia              | Claim, The                               |
+--------------------+------------------------------------------+
| Leeloo             | Fifth Element, The                       |
+--------------------+------------------------------------------+
| Joan of Arc        | Messenger: The Story of Joan of Arc, The |
+--------------------+------------------------------------------+
(4 rows affected)
```

Note that it is *not* the case that every entry in `Participation` has a corresponding entry in `ParticipationValue`. Some participations have no values linked to them, whereas some have more than one (for instance, if a person is both a director and an actor in the same movie). In that respect it might be better to use an outer join between `Participation` and `ParticipationValue`, although in this example the join type does not make any difference.

Object-oriented design

The main concepts in the object-oriented database remain the same as in the relational database. However, since it is possible (and advantageous) to represent relationships between objects via pointers, there is no direct representation of the `Participation` concept. You can see the UML diagram in figure 3.

The most interesting part of the object-oriented design is the represenation of the relational `Participation` table. Ideally, one would use ODMG's inverse relations with a suitable container to link Person instances to Film instances (or, indeed, other Person instances). Unfortunately, ObjectStore's Java interface (OSJI) does not offer this possibility, and it is the programmer's responsibility to uphold the referential integrity constraints. The `relations` attribute in `Person` and `Film` (see figure 4 on page 7) represents all the participation information available for each instance. This information is categorized (as in the relational

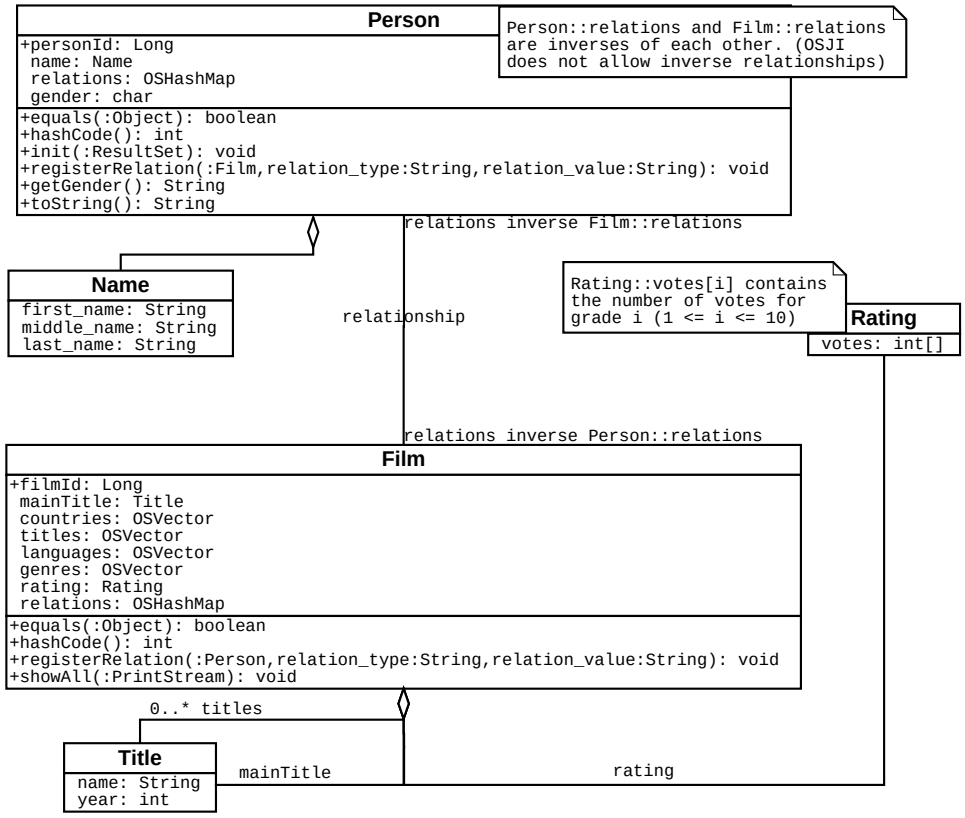

Figure 3: UML diagram for the object-oriented database

database), the hashmap keys being the categories. Since a participation can potentially be a pair (object,value), each category is represented as a collection of such pairs. ObjectStore offers several possibilities for such a collection. `OSSmallMap` is a reasonable choice, for it is designed for few entries and allows duplicate keys. Each (object,value) pair represents a connection from the holder of this pair to "object" with value "pair".

Also, notice that `productionYear` attribute does not appear in the UML diagram. This attribute has been embedded into `mainTitle`; i.e. `productionYear` for a `Film` instance is in fact `mainTitle.year` of that instance.

Figure 4 shows a snapshot of the data structure in memory. Notice how `Person` and `Film` instances are linked together.

Practical considerations

The relational database used in this exercise is Sybase and the object-oriented database is ObjectStore. Sybase comes with a C interface, and one can download a JDBC driver (jConnect) to access the database from Java. jConnect has already been installed at

```
/local/sybase11/jConnect-5_5/
```

and made available to you.

ObjectStore's main language is C++, but it comes with a Java interface – OSJI. The database can be accessed from either language, but you cannot rely on C++ specific features, for some of them do not exist in OSJI, which was used to implement the database.

In all cases only the Java interfaces have been tested. You are free to try the C and/or C++ interfaces at your own risk, but you will not get any technical assistance from the lecturers and the TAs.

This section provides practical hints on how to interface with these two systems.

Connecting to Sybase

The relational database runs on the Sybase server `gand` (default port 4100). The database's name is `inf212db`. To access this database (either programmatically or from an SQL shell) you need a Sybase user name and a Sybase password. Ifi's Sybase administrator distributes these at the beginning of the semester.

Sybase ships a JDBC driver, jConnect, that one can use to fetch data from the database. We have this driver installed and you need to adjust the `CLASSPATH` variable in order to use it. Basically, you need to do

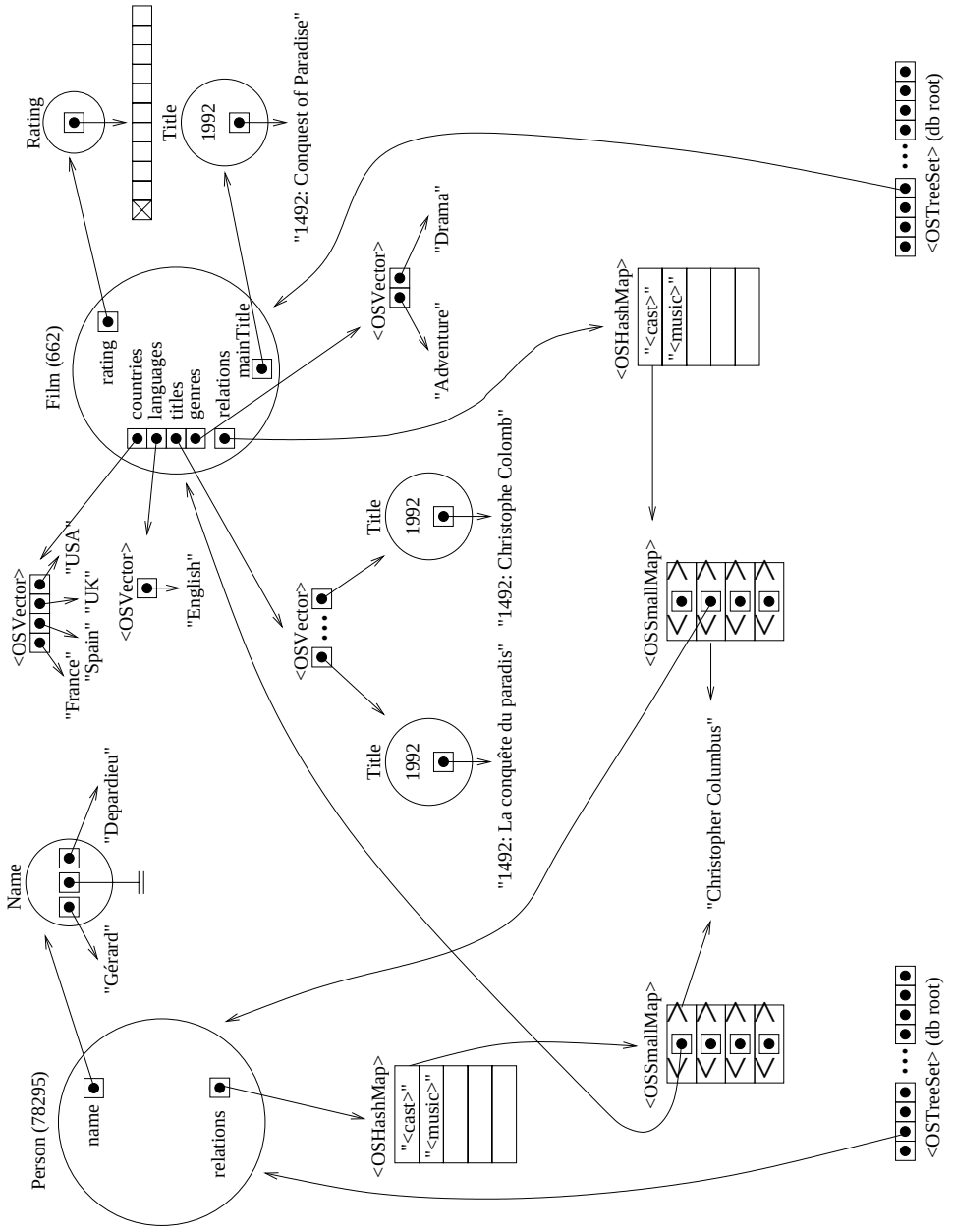

Figure 4: Runtime datastructures. This example shows Depardieu's role in "Conquest of Paradise"

three things to gain access to a database from your program via the Java database framework:

1. Load the JDBC driver
2. Register it with the JDBC DriverManager
3. Open a connection to a database

Loading and registering is more or less the same for all JDBC drivers, but opening a connection requires among other things a ”magic” string, describing the driver and some additional parameters. The jConnect manual describes exactly what this ”magic” should be. This manual also contains several examples that you should study. Alternatively you can consult references such as [4].

If you are not familiar with the JDBC API, you are advised to take a look at some books describing the Java interface to relational databases such as [4]. Java API documentation and jConnect manuals provide a lot of useful information as well (see the resources section on page 12).

Connecting to ObjectStore

The object-oriented database runs on the ObjectStore server `mjollnir` (default port). ObjectStore has a number of ways to address a database, and in this case, we would use the ”host:path” scheme. The database’s location is:

```
mjollnir:/ifi/mjollnir/kurs/inf312/sybase.odb
```

You will also need root names to access the data structures (Do not worry if ”root name” does not mean anything to you. The ObjectStore documentation covers this and many other topics in great detail). There are two roots in this database – one for `Person` instances and one for `Film` instances. They are `"people_root"` and `"movie_root"` respectively. The containers where the objects are stored are of `OSTreeSet` type. There is an index (`IndexMap` in OS terminology) for each `OSTreeSet`. Indices are created on `Person.personId` and `Film.filmId` respectively.

After running queries against the existing databases you will be asked to design and create your own database. You are free to chose any data structure you find suitable. You will be given a directory on the database server where you should (for performance reasons) place your database:

```
mjollnir:/ifi/mjollnir/kurs/$USER/whatever
```

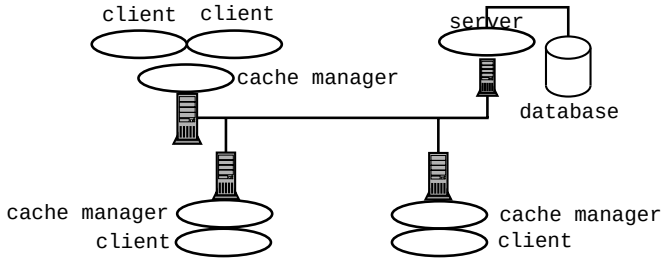

Figure 5: ObjectStore architecture

You can create indices for your own data structure (and you are advised to do so), subject to ordinary ObjectStore restrictions.

Connecting to ObjectStore is not difficult, and the exact procedure is described in the ObjectStore manuals. In order to use the OSJI, you have to set certain environment variables and adjust the `CLASSPATH` variable.

Each client connected to ObjectStore runs a cache manager (see figure 5). The cache manager needs some temporary scratch files, and there are two environment variables (see below) that control where these files are created. Notice that the directory where these files reside must exist prior to files' creation. The client manager should start automatically, once you attempt connecting to ObjectStore. Also, there is only one cache manager per host, regardless of how many clients run on that particular machine.

The basic procedure to access an ObjectStore database can be divided into the following steps

1. Initialization code
2. Open the database
3. Start a transaction
4. Fetch an ObjectStore root from the database
5. Access the relevant items
6. Commit the transaction

You should consult the ObjectStore manuals and study some examples for the details, but this list should provide a general procedure.

You will be given the Java source code for the object-oriented database. You are free to add auxiliary methods to the class definitions, but

you should not add or remove any of the attributes (otherwise the serialized objects in the database will not match the Java code available to the virtual machine).

To summarize, you might find the following setup useful for all INF312-related applications:

```
run312TNG()
{

    export OBLIGDIR="."
    # required by objectstore
    export OSJI_DIR=/local/ostore6/$HOSTTYPE/osji;
    export OS_ROOTDIR=/local/ostore6/$HOSTTYPE/ostore;
    # unfortunately required by objectstore
    export LD_LIBRARY_PATH="$OSJI_DIR/lib:$OS_ROOTDIR/lib:$LD_LIBRARY_PATH"
    # OSJI stuff
    export CLASSPATH="$CLASSPATH:$OBLIGDIR:$OSJI_DIR/osji.jar:\
                      $OSJI_DIR/tools.jar:$OSJI_DIR/browser.jar:\
                      $OSJI_DIR:/local/java/jdk1.2"
    # Certain OS applications need that
    export PATH="$OS_ROOTDIR/bin:$OSJI_DIR/bin:${PATH}"
    # Cache manager.
    export OS_CACHE_DIR="/tmp"
    export OS_COMMSEG_DIR="/tmp"
    # Disables IPv6 on Solaris 8 for ObjectStore
    export NETPATH="tcp:udp:rawip:ticlts:ticotsord:ticots"

    # OS6sp8 will NOT work with green threads
    export THREADS_FLAG=native

    # jConnect (Sybase JDBC driver)
    export JDBC_HOME="/local/sybase11/jConnect-5_5"
    export CLASSPATH="$JDBC_HOME/classes/jconn2.jar:$CLASSPATH"

    # we use jdk1.2 (or is it enough to set JDK to "1.2"?)
    alias java="/local/java/jdk1.2/bin/java"
    alias javac="/local/java/jdk1.2/bin/javac"
}
```

FAQ

This section presents a list of potential problems that you might face and possible solutions:

- You should not really need it, but the default heap size on a JDK 1.2 JVM is limited to 64MB. Should you require any more heap space, use '-Xmx**size**' option (see man java(1) for details). Such an option

was necessary when the entire relational database was converted to ObjectStore. If you need this option in part 2 of the exercise, you are probably storing too much information in your objects.

- Remember that you have to render the classes that you intend to store in the ObjectStore database persistent. Failure to do so will probably trigger a really strange exception when you try to access the database.

- Remember to set up the environment for working with ObjectStore/Sybase (CLASSPATH, LD_LIBRARY_PATH, etc.)

- Solaris 8 comes with IPv6 support; an interface which ObjectStore does not support. As documented in OS FAQ, setting the environment variable `NETPATH` to a certain value resolves this problem (c.f. the setup suggestion in the previous section).

- It is *extremely* important to implement your own versions of `equals` and `hashCode` in connection with ObjectStore containers.

  The default versions (inherited from `Object`) will *not work* properly and you risk not finding the objects that are already there.

  `Person` and `Film` classes supplied for the first part of the exercise have already properly defined methods; but for the second part of the exercise, you'll have to write your own versions. Remember the constraints dictated by the Java language specification that apply to these methods.

- You have probably noticed that Java's Long data type was used for various id fields. The reason for this seamingly silly decision is the inability of OSJI to provide indices for the built-in types. The documentation clearly points out that an index can be built over an `Object` subclass only. Since longs are not related to `Object` in any fashion, we had to use `Long` (which is a subclass of `Object`).

  Incidentally, this problem demonstrates yet again how *utterly broken* Java's type system really is.

- Although the data set in this exercise is very small by industrial standards, things still take time. Partly because of the old hardware, partly because of the deficient optimizers in Sybase and ObjectStore, partly because the available Java implementation has a high overhead. If you want to experiment with your datastructures or queries, try building a smaller test data structure or create a view spanning fewer rows (You can create a view in Sybase that queries a table residing in another database).

The OO queries in this exercise take time on the minute scale. Given a slow client machine (e.g. SPARC Ultra 1), it could take around six minutes to complete query 3 in task 1 (it does not *have* to be this slow, but this should give you an approximate figure).

The Sybase server is a bit slow too, but queries 1 and 2 take an insignificant amount of time; query 3 takes an order of a couple of seconds.

- Remember that recompiles of the persistent java classes require ”persistifying” (i.e., post-processing with osjcfp) of all relevant `class`-files.
- Despite the fact that OS' indices are at the very best clumsy, they are still useful, as soon as there are a couple of thousand elements. When you build your own data structure, remember that any kind of data loading that requires random access to elements would benefit from an appropriate index on these elements.
- Sybase ASE 11.9.2 supports outer joins. Look it up in the T-SQL reference volume 1 and/or T-SQL User's guide. You might need this to retrieve the information on participations.
- Apparently OSJI works best with Java JDK 1.2. It definitely does not support 1.4 or 1.3 specific features, but you might get away with using JDK 1.3/1.4 without such features. At Ifi you can set the environment variable JDK to ”1.2” to get the desired JDK version.
- OSJI requires native threads. Since JDK 1.2 on Solaris uses green threads by default, you have to set the `THREADS_FLAG` variable to the appropriate value (c.f. the setup suggestion).

Resources

As mentioned earlier, the relational database is running on a Sybase ASE 11.9.2 host `gand`. Since you have to interact with this database, it might be wise to look at Transact-SQL, Sybase's SQL dialect. There are several manuals pertinent to this installation available at `/local/doc/sybase/`. The most interesting volumes are the ones describing T-SQL – Reference Volume 1 and 2. For the love of God, do *not* print out the manuals at Ifi's printers, since each volume contains hundreds of pages.

Unfortunately Sybase is shipped with an incredibly awkward shell, `isql`. It is a fine tool for running scripts, but it is utterly useless for any kind of interactive work. If you want to fiddle a bit with any database, try sqsh (pronounced skwish, [2]).

Sybase's jConnect has a Programmer's Reference, that explains how to use this driver. To get started, look through chapter 2 of this reference. The information presented there should be sufficient for the exercise's purpose. The documentation is available at

```
/local/sybase11/jConnect-5_5/docs-45_55/
```

You might also want to take a look at the documentation shipped with ObjectStore. It has among other things a tutorial, an API user's guide, an API reference guide and a FAQ. Everything is located at `/local/ostore6/$HOSTTYPE`.

The chapters 1 and 2 of the Java API user's guide contain some basic information that you should familiriaze yourself with in order to be able to use ObjectStore. You should browse through the rest of the manual to get a feeling of the features available to you. Nevertheless, do not spend to much time studying the manual before starting to write an application, but look things up as needed.

The most interesting part of the API reference is the description of various containers available to you. Different containers implement different interfaces and you should think about what suits your needs best. Browse through chapter 7 of the user's guide to make the proper choice.

Once you have a grasp of the basic building blocks, take a look at the examples (available at /local/ostore6/$HOSTTYPE/osji/com/odi/). This should give you an idea as how to proceed with storing and accessing your data.

There is also an interactive "shell" (a point-and-drool interface, in fact) supplementing the OSJI. It can be started with `java com.odi.browser.Browser` and it can give you some ideas as to how the entire structure is linked together. You could also run simple queries through this interface. You might find it easier to navigate this structure during the development stage, rather than write a lot of print statements.

The author of this document is available for technical questions by mail (`<URL:mailto:igorr@ifi.uio.no>`) or in person. I will not grade this exercise, nor am I responsible for the exact list of requirements, but I can offer some assistance, should you have any inquiries.

Source code

You will be given the database "schemas" for the object-oriented and relational databases. Also, those of you who want to run their own server at home can download the SQL table definitions and indices. The data can be extracted from Sybase by using the `bcp` command.

You can add or remove methods from the classes you will be given, but you cannot add or remove data members (ObjectStore would probably die with an obscure message, since the data stored would not match the classes available to the JVM). As the bare minimum, you would have to copy the relevant java-files to a suitable directory and modify them to fit your solution for this mandatory exercise. The two most interesting files would be `Person.java` and `Film.java`.

All the relevant information is located at

`URL:http://www.ifi.uio.no/inf312/oblig/`

A word of warning – the person who wrote all the Java templates does not have a firm grasp of the Java language[3]. If you find some code that does not look like anything a Java programmer would do, you would probably be right in your criticism. Sorry about that.

Acknowledgements

I thank David Ranvig (davidra) for helping with the conversion of the IMDB datafiles into SQL insert statements. Unfortunately IMDB does not exactly strive to present their data in a format readily available for machine processing and David helped a lot with some serious data munging code.

I also thank Rune Aske (rune), who helped me test the IMDB ”mirror” on his personal computer (mogwai.ifi.uio.no).

[3] actually, I think that Java is somewhere between Tcl and VB on the silliness scale, but that's a different story.

References

[1] The Internet Movie Database, `http://www.imdb.com/`

[2] SQSH – SQL Shell for UNIX, `http://www.sqsh.org/`

[3] Kildekoden til inf212db, `http://www.stud.ifi.uio.no/%7eigorr/inf212db/`

[4] *"Java Enterprise in a nutshell"*, David Flanagan, Jim Farley, William Crawford and Kris Magnusson, ISBN 1-56592-483-5E, O'Reilly 1999