Assignment: DB

This assignment counts for 40% of the unit, and there are three weeks to do it in.

The main aim of the assignment is to gain experience of incremental development, including refactoring as well as unit testing. As well as your program, you should submit a report which describes the stages of development, the design decisions you made, any refactoring you had to do, and any particularly interesting issues that arose during implementation. In order to explain your progress, you might want to include zipped up copies of your code at each stage of development. In fact, if you want, you can submit each stage with its own mini report as you complete it.

This description deliberately describes more stages of development than you are likely to be able to carry out. And later stages are deliberately described with less and less detail, to allow for you to go in your own chosen direction. There is no one "correct" or "best" design here - there is a huge number of possibilities (though, of course, some designs are better than others).

It is important that you finish some well-polished and robust stages that you feel proud of, rather than racing ahead to try to get as many done as possible in a very rough way. And try to increase the breadth of your experience a bit, maybe practicing things that have been mentioned in feedback from previous assignments, or trying out new tools and techniques.

The idea

The idea is to design and implement a home-made database system. Some suggestions will be given for the first few stages of development, and then after that, a number of extensions will be outlined for the adventurous. I suggest not even reading about a stage until you have finished the previous ones, so that each stage comes as a change of requirements. As you finish each stage, you might want to keep a snapshot of it, to make it easier to produce a report at the end.

Please don't try to do everything - just work steadily, make sure each stage is clean and robust, and stop when you run out of time.

Here is some advice which might be helpful. When you are looking at the first stage, the Record class, you should not over-do it. It is probably much, much simpler than you think. I recommend not doing any input or output at all (trust your automated tests instead). You are not trying to write a program, only

implement one class. And that class is ultra-simple, probably only a handful of lines long.

The purpose of the class is to look after a bunch of strings (which are later going to form a row in a table). Just as the Board class in Oxo is effectively just a 3x3 grid, wrapped to make a meaningful game state, so the Record class is effectively just a row of strings, wrapped up to form a table entry.

Maybe you don't yet know what operations to provide on this row of strings, beyond getting strings in and out. That's normal, don't invent anything. Just wait until it becomes clear what's needed when you get to later stages.

You should write a main method in the Record class, but that's not to form a program. You might use it to experiment to help you understand what's going on, but it should become your automated unit testing for the class, based on a few hand-written strings.

Records

The first suggested stage is to sort out how to store individual *records*, also called *rows* or *tuples* or *objects*. A record contains named *fields*, also called *entries* or *values* or *cells* or *items*. For example, suppose you have a table of animals and a table of people like this:

Id	Name	Kind	Owner
1	Fido	dog	ab123
2	Wanda	fish	ef789
3	Garfield	cat	ab123

Username	Name
ab123	Jo
cd456	Sam
ef789	Amy
gh012	Pete

The first contains three records and the second contains four. The animals table has fields called "id", "Name", "Kind", "Owner". The people table has fields "Username", "Name".

The idea behind this stage of development is that you develop a single general-purpose class called `Record` (or whatever you want) which will hold any type of record, no matter what table it is from, as a collection of fields which are strings. The operations on a record object are, perhaps, to find out how many strings it has, and to get or set individual strings.

I would suggest that you ignore the field names for now, and just store the string values in a specific order. In other words, a record is hardly any more than an array of strings. The `Record` class is just a place to gather code later as development continues. Alternatively, you could store the field names in a record, but with a view to sharing them between all the records of one table later, so you don;t store the same data multiple times.

Tables

The second stage is to deal with *tables*, also called *relations* or *types*. A table is a collection of records, all having the same number of fields, in the same order, with the same names.

Again, the aim is to design a single general-purpose class, say `Table`, to hold any table. At this point, you may want a table object to hold the field names, rather than repeating them in every record object. In that case, the fields are often called *columns* or *attributes* or *properties*.

Common operations on a table which you might want to make convenient or efficient are to *select* a record (perhaps by row number), *insert* a record, *delete* a record, or *update* a record. You might also give a little thought to doing these operations on multiple records at the same time.

Less frequent operations, which don't need to be so convenient or efficient, might be to *create* a table with given column names, or *alter* a table, e.g. by adding an extra column.

Depending on your approach and attitude, you may find a potential problem at this point, because it seems that a table needs to know about the records in it, and a record seems to need to know what table it is in (so that it knows what field names it has). There are several ways to resolve this issue.

Files

In this stage, the data for each table can be stored in its own file. The file holds (say) a first line which acts as a header of column names, and one line for each records. You might want to use readable text files. In that case, you might use an ordinary character such as a comma between fields, and a newline as a record terminator. But then you need some mechanism to allow commas and newlines in field values. Alternatively, you could use a format where suitable control characters are used to separate fields and terminate records, while insisting that only readable non-control characters are allowed in fields.

The record and table classes now have to be adapted to allow for loading up their data from files, and writing their data back to files. You may want to assume, at this stage, that writing back to files is done just once when the overall driving program shuts down, rather than bit by bit as changes are made.

There are alternative file storage possibilites that you might want to investigate either now, or as a later stage of development.

Printing

At this stage, there should be a way to print out a table neatly, preferably with the columns lining up.

Keys

A key is a unique identifier of some sort for each record of a table. Either the key can be something separate, or it can simply be assumed to be the first field and handled more-or-less the same as the other fields (either approach can work well). Also, the key can be automatically generated, or the user might be responsible for it in the same way as any other field (either approach works).

Changes to the classes already developed might include making sure that keys are unique, i.e. that there can't be two records with the same key in the same table. Some thought, and possibly refactoring, may be required to find the best way to guarantee this kind of consistency property.

Keys can now be used to provide a more stable way of referring to records, rather than using row numbers - the row number of a record changes as other records are inserted or deleted, but the key doesn't change. So, in the animals and people example above, the "Owner" field in an animal record holds the key (username) of its owner.

Databases

A database is a collection of related tables, each with its own name. What is needed is some way of wrapping up this idea, perhaps with a database class and some conventions for a database folder.

Extensions

The stages described from this point on are extensions. Please feel free to stop at this point, or to choose which extensions you want to concentrate on, or to change their order, or to go in a completely different direction of your own.

Types

Each column in a table can be given a type. Values can still be stored as strings, but the type controls what strings are allowed and how they are handled. Alternatively, values of different types can be stored in different ways, but wrapped so as to make fields of different types compatible.

As an example, a column of type `integer` might restrict strings in fields to digits, and cause printing to be done with right-alignment. Another possibility is a tag type, where each field can only hold one of a short enumerated list of values. Most importantly, a reference type is needed, representing a "foreign key" field, i.e. a field which refers to a record in another table by its key.

Constraints

Once you have foreign key fields, there is the question of how you make sure that such fields always refer to other records which actually exist. Automatically checked constriants are one of the strengths of database systems.

Catalogs

Some mechanism might be needed to access meta-information about a database, such as a list of tables, the types of their columns, and so on, and some way of querying this information perhaps in the same way as querying ordinary records.

Query language

A textual language could be added for querying a database. The SQL language is standard, but dreadful, for a variety of reasons. You might prefer to try implementing some cleaner, after reading this:

SQL notes

Transactions

A transaction is a series of queries treated as an atomic unit, i.e. if anything goes wrong, the database is kept in or restored to the same state as before the queries were started.

Journals

A journal is one way in which changes to the data can be stored as you go along, in a way which is very robust. If the program or computer crashes, the database can be restored to a safe, consistent and recent state.

User Interface

A full textual user interface, or even a graphical one, could be added.