Project Grading Guide

Database systems course

Yael Amsterdamer

The following document describes the Database Systems course grading policy and enumerate over common problems detected in some previous years' works.

The grade is composed of 3 parts (source, database and application) – each gets a third of the points, and a bonus reaching the maximum of 10 points. In general the parts grading is based on details in several aspects. For clarity I generalize these aspects in the following categories:

1. **Excellent** - I have nothing to say. This is the way it should be done.
2. **Almost Excellent** - This is close to being perfect but something here is a bit annoying.
3. **Good** - Not too brilliant, but does the work.
4. **Well….OK -** you probably have to do a massive makeover to make it acceptable in the real-world.
5. **Not acceptable -** This is not one of your best works. Probably you didn't think about these matters at all – however we think you should.

Source part

This part examines how the code is implemented: Whether you understood and incorporated the stuff shown in class, and whether the code is acceptable in general. This part has the following aspects:

General structure

This section examines your overall program structure, how you submitted your work, and how it can be examined as a piece of software. The main perspective here is the software engineering aspects which include design and coding issues. The following categories describe the work:

1. **Excellent** – You have the correct overall structure, and demonstrated expertise in naming convention, comments, function separation, routine, parameter passing, etc.
2. **Almost excellent** – The overall structure is fine, but some classes and/or routines are implemented in a dirty way – so that they the general structure is not fixed. Try to make more abstractions or find mode code that can be logically reused.
3. **Good –** The overall structure is not good, however it demonstrates that you do know how to make it better – you simply chose not to invest on this issue.
4. **Well….OK –** code will work, but without any foreseen plan behind it. There are some "islands" that can be used but mainly restructuring is needed.
5. **Not acceptable –** your code was not planned at all, (or it was but with the "wrong" aspects in mind) this requires complete re-write for any real-purpose.

Modularization

Source code should be modular – the purpose for the modularization is to allow abstraction, encapsulation, reuse, and more. This includes package/module separation, routines and utilities definitions, global/local variables, static/non-static declarations, enumerations, etc.

1. **Excellent** – Your code is well separated and designed. You understand why you need to create business logic, and when to use utility classes. You know and use packages, and demonstrate reuse in your code.
2. **Almost excellent** – You defined the main packages and modules, however there are some 'dirty' classes floating around.

3. **Good –** You have some of the correct objects in the correct level, but there are many other places where it is less good. For example, as we discussed in class, the layering can give you hints about the modularization, or business logic should be implemented in business objects.
4. **Well….OK –** You implemented everything – but the resulting code does not reflect and show reuse/modules/structures. This usually happens on systems that had short design time (which is perfectly OK according to some methodologies such as extreme programming http://www.extremeprogramming.org), but were not restructured/refactored later on, according to the result.
5. **Not acceptable –** The code seems to be built without trying to reshape it. Such code is really very hard to maintain and recognize – so many times you probably will prefer rewriting instead of fixing broken code.

DB connectivity code

Using the various APIs presented in class, while demonstrating understanding key issues, such as connection strings, pooling, queries, result sets, etc.

1. **Excellent** – Your code interacts with a database correctly.
2. **Almost excellent** – Mainly the code implements the important constructs of the database connectivity, however either it is not complete or some of the features are not implemented correctly. Go over class notes and see what you are currently missing.
3. **Good –** You have the main pieces in place, but there are many issues that are being overlooked – such as where the query should be defined, what database features to use (stored queries, paging, pools) etc.
4. **Well….OK –** You use the APIs, but did not manage to set them in the correct order.
5. **Not acceptable –** Although it works, it will be difficult to maintain this.

DB code separation

As discussed in class, Database handling and programmers should be separated. Furthermore, queries and db structure is something that should be easily changed, and should not make strong implications on the software. These are the main reasons for separating the two.

1. **Excellent** – You implemented code which demonstrates how the DB code should be separated from the rest.
2. **Almost excellent** – You separated, but there are small cheats that are hiding such as static/global variables, assumptions on the connection, tables, etc.
3. **Good –** Some code portions demonstrate the requested issues, others do not. Look at parameters of functions, and variables to realize how to completely separate the two.
4. **Well….OK –** the code is mostly intermixed
5. **Not acceptable –** you didn't understand that point. Code is completely intermixed, causing a nightmare for maintenance for both the DBA and the programmers.

Fault tolerance and traceability

Code should handle errors in connection, database errors, schema mismatches, and errors related to use input, which are usually checked though input validation. Failing to check the system status after executing some of the APIs, or letting the user type problematic input usually cause problems in the long run, and are hard to detect.

This issue also includes reporting the correct state when problem had occurred, so that errors can be traced back to their origin.

Software should be fault-tolerant in the sense that minor problems in database (such as reconnection glitches, should be handled transparently to the user).

1. **Excellent** – this is how it should be done.
2. **Almost excellent** – some small number of validation functions is missing, or can be improved.

3. **Good –**Checking system and user inputs, but in some cases, not checking for sufficient error faults – for example exceptions that are be caught but not handled.
4. **Well….OK –** Sometimes you check, but mostly not.
5. **Not acceptable –** The software is built in an 'optimistic' manner – it assumes that everything will be OK.

Documentation

This includes both design description, but also user manual, external code documentation such as javadocs, and documentation in the code. I will not be the first to say that the code should document itself.

Database part

This part checks the structure of your database, while examining some of the points discussed in class by Tova and by Rubi. We examined the REAL database as implemented on ORASRV, so if you described the DB structure in your documentation but failed to implement it in the database, it is regarded as undone. In general, here we look at fulfillment of the following aspects:

Structure

Checks what are the tables and fields you defined including types, lengths, etc.

Queries and Indexing

Checks if you had defined indexes for the relevant fields, according to your queries.

Keys, FKs

Checks if your database has correct keys and foreign keys defined.

Additional issues

If you have used views, triggers, stored procedures and other database capabilities it brings you an extra edge in this section…

Application part

This part is the most controvertible part, but we feel that it is important, and will contribute to everyone. In this part we look at the overall outcome – examining the work using real-life parameters – how complete it is, how its target audience can use it, how it behaves, and so on.
We agree that this is a university project, but as stated throughout the course, and as some of the work had shown, it is possible to implement such applications without putting irrational effort. I strongly suggest having a look at the better projects, or even other software and to get an understanding what went wrong, and how you can improve your application.

Usability

This issue tells you how users can easily understand and use your software. It does not tells you how smart you are, and/or how much effort you've put into the project.

1. **Excellent** – This is the way to go.
2. **Almost excellent** – You've built a usable application, but didn't complete it – syntax errors, logic problems, significantly incorrect layout of the screen, etc.

3. **Good –** Some screens are more usable then others, but in general it requires some fixing and restructuring.
4. **Well….OK –** It is usable if your users are 'highly motivated' (and read the manual several times)
5. **Not acceptable –** it is very difficult to understand how to operate the application, and the overall result does not look like it can be used in real-life.

Features

Does your software implement sufficient features to be helpful in the domain? By features we mean sufficient query power and sufficient information insertion capabilities.

1. **Excellent** – You have a full application including all the actions that should be included. For example FULL implementation of user management, printouts, import/export, etc.
2. **Almost excellent** – your application is doing what it is supposed to do, and you also added some of the system features, but they are not built completely, and are more of an intension presentation.
3. **Good –** you can add records and query them, but mainly this is it
4. **Well….OK –** You can add records, and query them, but in a very restrictive manner.
5. **Not acceptable –** you don't have the critical mass to be an application. You need to rethink how users are going to use your application, and what is missing.

Complexity

How "smart" your application is, and how much you rely on the user intelligence to do the work. This issue includes did you implemented ideas that assist during the input process, and/or tools that allow the user to get an added value from the system. This also includes input validation and user assistance during the work in the application.

1. **Excellent** – This is the way to go.
2. **Almost excellent** – You've built a significant complex application, but didn't finalize it – syntax errors, logic problems, not the correct layout on the screen, etc.
3. **Good –** In some screens it is complex enough, but in some the result is too naïve.
4. **Well….OK –** It does implements all the necessary components, but none of them is really sufficient to assist real users to do their jobs.
5. **Not acceptable –** The resulting system is highly naïve and does not express sufficient complexity expected.

Additional features

Features that were not implemented by other applications, and demonstrate that you have a significant beneficial advantage over the applications shown by other users.

Bonus

Bonus can account up to 10 points, but it needs to be something original (really innovative), and well implemented. Examples for this are recommendation system, online video store (just the management...), decent GUI, etc.