**Project Guidelines**

This document provides you with some advice for completing your web programming project. If you find better ways of doing things, then don’t be afraid to do so. Talk to your supervisor regularly.

In general, follow the specifications faithfully. If you find ambiguities or choices, resolve them with your supervisor.

Iterate through the specification constantly. For example, when you design your tables, check that everything that the specification requires can be achieved with your table designs.

**ER diagrams and Database Tables.**

One of the very first tasks you will do in your project, once you have read the specification and understood it, is to begin work on your database. You will begin by establishing all of the entities involved. The best place to look for these is in the list of functionalities. Another good place to look is the table of contents.

A common problem you will have is too much choice, and the possibility of multiple correct answers. For example, in a banking project you may have the choice of creating a single entity called accounts. This is alright; however it will be difficult to accommodate for differences in account types. For example, a loan account will have a monthly repayment amount, a deposit or current account will not. You also have the option of creating an entity for each account type. This accommodates the differences in field types required, but requires you deal with each type separately.

You should choose which option is best by assessing the amount of work involved in each case. Minimise the effort required. This is not being lazy - this is being smart and efficient.

Once you establish your entities you will need to create your ER diagram. This, as its name suggests, establishes the relationship between entities. This can be tricky. If there is a relationship between two entities, then ultimately data will need to pass from one entity to another. If this doesn’t happen, then there is no relationship.

Sometimes establishing entities and relationships can be very confusing. Consider going into a hardware shop and buying a tin of paint. Are you, as a customer, an entity? If you are then information about you must be stored somewhere in the system. Is the tin of paint an entity? Is it stored somewhere on the system? Is the sale or transaction an entity? Is the person at the cash register an entity? In the latter case if you ask is information about the person stored on the system you may get different answers. Usually, the person’s details are not relevant to the transaction – they are just acting as an agent for the overall system. However, if the person’s name appears on your receipt then they are part of the transaction. This might happen if the system was proving an incentive scheme for the best salespersons, for example.

One item causes some confusion in these projects. Reports. A report is not an entity. Think about it – to generate a report you search the database looking for information based on some criteria. You then present this information in some ordering or format. The information is already in the database and is readily accessible. Reports are transient – they are not stored anywhere (as a report) and effectively cease to exist once we move on (unless we print it of course).

Finally, it is possible to have entities, or groups of related entities, that are completely detached from other parts of the ER diagram. Very often staff and users of the system may be detached. This is perfectly OK.

**Approach things from a user’s perspective.**

When implementing screens in your project, always try to see them from the perspective of the eventual users of the system, more so than that of the programmer.

Use language that is as helpful as possible and don’t use multiple terms for the same thing (e.g. the first macintosh computers used the terms *format*, *initialise* and *setup* in various instances when offering to format a disk – this is needlessly confusing and unhelpful).

Provide help where necessary, but discretely. For example, for each input field provide sample data in that field that helps the user. Show this sample data in a light grey colour (or similar) and make it disappear once the user starts inputting to the field. The user’s input should be black (or similarly emphasised) to distinguish it from sample data. The sample data should be carefully chosen so that the user can immediately grasp what format is expected. For example, sample data in a field to get the user’s telephone number might be +353 (0)59-9175000. This tells the user at a glance that numerics, ‘+’, ‘-‘, brackets and spaces are permissible.

Consider having a help button after each field to tell the user more formally about the expected formats of that particular field. This might be implemented as a small pop-up window (that doesn’t obscure the field in question).

Consider too, having a general help window for the entire screen.

In general, don’t impose help on the user, but have it easily accessible if they need it. Keep your help consistent over all of your screens.

Be careful of help when security is an issue. For example, when logging someone in to a system hold back on the help – don’t pinpoint which field an error is detected in as it might just help an opportunistic hacker. Don’t tell them the wrong number of digits was supplied in a password either.

**Validate all data.**

If data isn’t thoroughly validated, then it’s going to corrupt your database. At the same time, don’t over validate data so as to restrict the user unnecessarily.

When inputting names consider what can be present in a name. Obviously alphabetics should be expected. Do we allow hyphens, apostrophes and spaces? Do we permit characters such as numerics and other symbols? What about foreign language considerations, such as accented characters, Do we really need that much validation at all?

We’ve considered phone numbers already. We definitely need numerics. Allowing some other characters gives the user more flexibility. Ensure your validation is logical. A common error many students make is to set minimum and maximum numbers of digits with no justification whatsoever.

E-mail addresses. Look for the ‘@’ symbol. Should you look for dot separators ‘.’?

Web addresses. Should you look for ‘www’? Should you look for ‘http://’ or ‘https://’? Should you look for dot separators ‘.’? If you approach this from the user’s perspective and consider what they are likely to input, then you are more likely to get the correct balance.

**Required fields.**

In general, you should ensure that all fields are completed before writing to the database. However sometimes not all fields are required. The perfect example is address fields. How many lines should an address have? Sometimes three lines will suffice, possibly two (e.g. ‘Santa Clause’, ‘North Pole’). So, don’t insist on four lines – no more, no less - for an address. We have all suffered web sites that impose a very strict view of what should be in an address. They won’t let you buy stuff from them unless you live in a town, live on a street, have a postcode, and have a number affixed to your house. Whoever programed these systems just didn’t bother thinking about what is required, thus causing the user unnecessary, and sometimes insurmountable, difficulties.

Remember that not everyone has a land telephone line. Some people don’t have mobile phones either. Do you know that you can’t book your car in for the NCT online if you don’t have an e-mail address (or don’t want to disclose it)? Then they won’t send you reminders by post afterwards once they extract an e-mail address from you.

It is also a good idea to highlight required fields – this is often done by finishing the prompt for the field with an asterisk character. However you do it, do it consistently across all of your screens.

**Validation and required fields.**

Many students make a very good job of validating data and checking required fields when they create a new record (e.g. ‘Add a New Customer’, ‘Add a New Stock Item’, ‘Add a New User’, etc). Inevitably they will also have to program an equivalent View/Amend screen (e.g. ‘View/Amend a Customer’, ‘View/Amend a Stock Item’, ‘View/Amend a User’, etc). More often than not, validation and required fields are not checked for. The required code can be cut and pasted from the ‘Add’ code with minimal or no change. Don’t throw away easy marks.

**Confirmation screens.**

Every screen that has input fields (either to write a record to the database, or extract data for a report or some other check) should have some form of ‘submit’ button. First of all, use the term to carry out the screen’s function consistently. Use ‘submit’ all of the time, or use ‘save’ all of the time – don’t mix terms unnecessarily.

When you submit a screen, scan through the fields and check for valid data and required fields. Work systematically through the fields, and each time invalid data is found or a required field is found to be empty, return to the user highlighting which field the problem has been found in and a helpful diagnostic to tell them what the problem is.

Once all of the fields have been checked, ask the user to confirm that they still wish to perform the desired action, e.g. write to the database, using a pop up window. This is a good opportunity to present a summary of the data for the user to double check, if you wish.

If the user elects to go ahead with the action (e.g. click ‘OK’), then the action should be performed and the user notified on completion. If the user declines, then they should be returned to the screen with all if its input data still intact.

It is also a good idea to have a ‘cancel’ button adjacent to the ‘submit’ button that allows the user to abandon the current screen and go back to the previous one. A confirmation pop up might be a good idea in this instance too, to prevent unintentional data loss.

**Screen essentials.**

* Every screen should have the same design basics (e.g. layout, logo, etc).
* Input fields should be nicely aligned.
* Input field labels should also be nicely aligned, and as informative as possible.

**Add / Create screens.**

* Field validation.
* Required fields.
* Help.
* Submit and Cancel options.
* Confirm screens. Yes: database written, No: return to add screen with inputs intact.
* Generate a unique primary key for the record by finding the largest one in the table so far and adding one.

**Delete screens.**

* Search record methodologies (e.g. drop down lists, record ID, etc).
* Filter deleted and otherwise irrelevant records from search (important).
* Populate fields on screen.
* Fields locked (cannot be edited).
* Help.
* Submit and Cancel options.
* Confirm screens. Yes: database written, No: return to delete screen with inputs intact.
* Requirements (very important) – e.g. cannot delete a bank customer if they have an account with a non-zero balance.
* Delete fields – in general we don’t delete records – we have a ‘delete field’ in each record which we flag as deleted.

**Amend / View screens.**

* Search record methodologies (e.g. drop down lists, record ID, etc).
* Filter deleted and otherwise irrelevant records from search (important).
* View mode by default (all fields locked)
* Amend/View button to toggle into mode (editable fields opened – primary key always locked – other fields may also need to be locked)
* Field validation.
* Required fields.
* Help.
* Submit and Cancel options.
* Confirm screens. Yes: database written, No: return to amend/view screen with inputs intact.

**Report screens.**

* Search the database for information that meets some designated criteria (e.g. sales this week) and present it in a table on the screen.
* Filter deleted and otherwise irrelevant records from search (important).
* Table may need to be presented in different ways (by surname, age, etc.)
* Buttons to choose presentation mode.
* You may need to extract information from more than one table (important).
* You may need to be able to click on a certain row of the report to call up more information (important).