CURTIN UNIVERSITY (CRICOS number: 00301J)
Faculty of Science and Engineering
Department of Computing
Software Components 400
(Index No. 4547)

Assignment, Semester 1, 2014

1. Overview and Background

Your task is to implement a GUI front-end for displaying and maintaining the contents of a company's warehouse inventory. The company is in the business of collecting items ranging from furniture to cars to rather more exotic items, and has a very large inventory numbering in the millions. The company has numerous warehouses around Perth and has a database that records the items and their information. Your problem is to provide a front-end that allows the user to browse the database and allows editing / adding items in the database. The system must provide both a native Windows GUI as well as a Web interface, where both interfaces must provide a responsive experience despite the size of the database. In addition, as a proper software professional you should also be ensuring that you utilize the network bandwidth as efficiently as possible whilst maintaining the user's responsiveness requirements.

DATABASE

The customer has their own existing database that they require you to interface with (NOTE: the database is populated with a million auto-generated rows. This is so that you will have a pre-existing database to work with). However, since the customer's entire business depends on their data you will not be given direct access. Instead they have developed a C# DLL that will provide you with restricted access to the database, limiting access to that which is necessary for the application you are to write. This DLL is called WarehouseDLL.dll and will be provided to you.

Database Structure

The database is very simple with only one table called Warehouse containing the following fields:

WAREHOUSE:

- **ItemID** – Primary key field. Integer number from 1 to N, where N = number of items

- Category – String field. Type of item in the warehouse (desk, computer, etc). This is free-form in that any value can be entered (in reality there would be another table that lists the valid Categories, but that complicates your assignment for no real learning gain).
- Brand – String field. Manufacturer of the item (eg: Ford).
- Model – String field. Manufacturer's model name for this item (eg: Falcon).
- ReceivedDate – DateTime field. Date that the item was acquired.
- Weight – Integer field. The weight of the item in kg.
- Condition – String field. The condition that the item is in. Like category, this is also a free-form value.
- Location – String field. The warehouse that that the item is in (based on suburb). Like category, this is also a free-form value.
- PurchasePrice – Double field. The price that the item was purchased at.
- SellingPrice – Double field. The price that the item is to be sold by the company's salespeople.
- Notes – String field. Any other notes that are relevant (this may be multiple lines).
- LastModified – DateTime field. The timestamp when the item was last modified. Use for optimistic locking (see later). This field is automatically maintained by the database – you will only have read-only access to this.

Since it is a C# DLL you will be able to use it by simply adding a reference to it via Project >> Add Reference (ie: there is no need to use [DllImport] like for the C++ TrueMarbleDLL.dll in the practicals). The DLL contains one class – WarehousDB – with the following public methods and properties for accessing the database:

- WarehouseDB() – Constructor
- NumRows – Property that returns the total number of rows in the database
- GetItemIDList – Retrieve the list of all ItemIDs (primary key). Has an overloaded method that returns the item IDs of a subset of rows
- GetItemRecord – Retrieve the details of one item given its ItemID.
- AddItemRecord – Create a new item record given the item details, and returns the new record's auto-generated ItemID.
- UpdateItemRecord – Update the details of an existing item record.
- GetColumnNames – Retrieves a list of the names of each column in the database. You may find that you do not use this method.

Note that the DLL is not thread-safe, performs no table locking and has no synchronisation code to avoid lost updates, hence you will need to perform this yourself when controlling access to the DLL.

2. System Functionality

The following lists the functionality required by the system. Note that no login, authentication or security is required for this assignment, which is in keeping with most search engines.

2.1 Architecture

The user requires that this system be distributed, that is multiple GUI client machines must be able to view and update the data on a central database. In other words, at least two tiers are needed. It is up to you whether to add more tiers, but note that you will need to justify your choice in deliverable for Section 3.D1 (whichever option you take).

The system must implement both a Windows native GUI clients as well as a Web browser GUI frontend. The back-end tier(s) must also support this. Thus you must implement two types of GUI, where the Web GUI may be implemented as either using ASP.NET pages (recommended) or via Ajax and Web Services (discouraged since it makes things harder and you don't get taught Ajax until the very end!!). Both GUI types must provide the full functionality.

2.2 Warehouse Item List Viewing

The main user interface must provide the user some means to display a summary of the items in the warehouse. Summary info to be shown for each item includes:

- Item ID
- Category
- Model
- Brand
- Condition
- Selling Price

Other information is only to be shown if the user wishes to see the details of a given item. You are *not* required to sort the list – just display it in the order that the database provides it to you (in a real database, you could ask the database to sort it and provide it in sorted order).

The main issue facing you is the need to provide the list in such a way that it can be navigated relatively easily whilst not wasting bandwidth. In other words, you must not throw the entire database of a million records to the client GUI and have it display them, since that is not efficient use of bandwidth nor is it a usable interface given how long it will take to transfer and display.

- To do this, it is recommended that you display the items a 'page' at a time, where a page is simply (say) 1000 items (ensure that the last page displays correctly even if there are fewer than 1000 items in it!). Ensure that you allow the user to move forward and backwards between pages as well as jump to any page that they wish.

- Note that you will need to compromise between network efficiency and user interface responsiveness, and will need to discuss your choices in the critical self-analysis report.

- You may feel that additional or different approaches are warranted, such as including a Search mechanism that allows a user to narrow down the list of items, or something else entirely. Feel free to do differently to the above recommendation, but be aware that the GUI has no marks.

When the user is provided with the display, they should be able to select an item and View or Edit the details of that item. This will require opening a dialog window that allows the user to see the entire record of that item (see section 2.3). Note that there is a difference between View and Edit – View is read-only whereas Edit must allow the user to make changes and save them to the database (or cancel) In the real world, View would be for low-privilege users such as salespeople whereas Edit would be for admin and stocking users.

The user must also be allowed to Add a new item, which will also show the dialog but with it being initially with empty fields.

2.3 Warehouse Item Maintenance

As mentioned, the user must be able to open an item for viewing, editing or adding. All three should display the full details of the row (except the LastModified field) – they differ in what can be edited and what the user can do.

The Notes field should be a multi-line textbox (<textarea> in the Web)

In all cases, the user requires the ability to open multiple items simultaneously in the WPF interface (ie: Modelessly). Thus you cannot use ShowDialog. See the Appendix for hints on how to coordinate between the main form and dialog(s). **This is NOT required in the Web interface** (since it is quite a bit harder to do!)

View Item:

- All fields (except LastModified) must be shown but not editable (ie: ReadOnly=false in native GUI textboxes, “disabled” in Web textboxes).
- The user can only Close the dialog

Edit Item:

- ItemID must be read-only.

- All other fields should be shown with their current values and editable.
- The user can choose to Save the item (closing the dialog) or Cancel the edit.

Add Item:

- ItemID should be read-only and showing the value "<new item>".
- All other fields should be shown with a blank field (or current date) and editable.
- The user can choose to Add the item (closing the dialog) or Cancel the add.
- If the user Adds the item, a MessageBox must be displayed showing the ItemID of the new item.

It is recommended that you create a *single* dialog window class and simply change the behaviour/labels of the fields, buttons and titlebar based on what kind of dialog it is required to be (change the default constructor to add a parameter indicating the behaviour type, and store this type as a member field – then when the main form creates it, it must tell the dialog what type it needs to behave like).

When the user Saves or Adds an item, the main form must refresh the displayed list in order to reflect any possible changes. See the Appendix for hints on how to coordinate between the main form and dialog(s).

2.4 Database Integrity

Being a multi-user system, your system should try to ensure as best as possible that it does not lose updates to the database or otherwise corrupt the database:

- Synchronising access to the database methods is thus necessary. Since neither the database nor the DLL provide the ability to lock a record, you must use thread synchronisation in your singleton data tier controller object to ensure one-at-a-time update access to the database.

- However, this does not allow you to lock a record for the time it takes a user to *edit-and-then-save* a record, only for the time it takes to *save* the edited record. Thus it is possible that User B has edited and saved the same record already in between the time User A opens the record and saves their own changes, leading to a loss of User B's edits. Hence you must use *optimistic locking*, where on saving a record the system checks the LastModified field against the record to save. If the two DateTimes match then no-one else has edited that record and it may be safely saved. If they differ, the change must be discarded and the user (User A in the above example) told to retry editing the record.

- Note that LastModified is only used for this purpose, and is maintained by the WarehouseDLL itself : you'll only need to check it and the user won't want to see it.

There is no commit/rollback facility with this database. So you won't be able to make database updates completely bulletproof. But try as best you can.

A design question you have to resolve is how to pass the Item record data across the network, and how much of it do you need to pass, and at what times? You will need to evaluate your own choices in your critical self-analysis.

3. Deliverables

In addition to software copies of your Visual Studio projects and code, you will need to provide a written 'report' that addresses the deliverables below. **The report must be submitted in .pdf format via Blackboard by the assignment deadline with a dated assignment coversheet.** You should make the report look professional, creating a section for each deliverable area and providing discussion that demonstrates your critical thinking skills.

3.D1. Architectural/High Level Design

Provide an architectural (high level) design of your system. In particular:

a) Provide a diagram showing the potential deployment of your system in terms of server machines, client machines, networks/Internet involvement, the tiers that these fit into and the lines of communication between all of these components.

b) Provide a UML class diagram to show the main entities of your design and how they relate to each other. Do not forget to list the methods and field (properties) of each class, but don't worry about showing parameters or overloaded versions since that just causes clutter. **Identify / label the tiers you are utilising and delineate the boundaries of these tiers on the UML diagram** to show what classes exist in what tiers and by extension the RPCs.

Notes on the UML diagram:

- If a class inherits from .NET classes, show the inheritance but do not show the details of the .NET class (ie: just put the name of the .NET class in).
- You may use Visual Studio's UML diagramming tool to simplify the process of producing the UML, but you will need to update it to show all message passing and other relationships that it cannot extract automatically (ie: the difference between composition and aggregation).
- If the UML diagram is too large to fit on a single page, place each tier on its own page and for relationships with classes in other tiers just show the names of these cross-tier classes (since their details will be shown on the page for the other tier).
- You do not need to use the classical three-tier approach. Regardless of your choice, you will be required to justify it in D4.1.

You may also decide that UML use-case scenarios will be helpful in designing your system. However, this is not a software engineering unit so no marks are allocated for them.

See the Hints in Appendix A at the end of this assignment for tips on doing your design.

Marks Breakdown (Architectural Design):

- Coverage of user requirements 5 marks
- Tiers shown and labelled appropriately 2 marks
- Deployment architecture diagram 5 marks

Total: **12 marks**

3.D2. DETAILED DESIGN

UML Sequence Diagrams:

A full detailed design is too large to deliver for an assignment. Hence I only require a *UML sequence diagram* for the following use-case interactions:

a) Display/browse the list of warehouse items using the native GUI
   - From initial loading of items from the database through to the display of the first 'page'.

b) Add new warehouse item using the native GUI
   - From when the user clicks on 'Add' up to when the new item is saved to the database and the user is returned to the main list view
   - Note that this will be two separate sequences: (1) the loading of the form ready for the user to edit and (2) the click of the OK button of the dialog window
   - Include the OnClick button event handler in your sequence diagram
   - It must even include the display of the new ID and the notification to the main form of the update

c) Edit existing warehouse item using the Web interface
   - From when the user clicks on the item to edit, through loading the investor, up to when the item is saved successfully to the database and the user is returned to the main list view
   - Note that this will be two separate sequences: (1) the loading of the form ready for the user to edit and (2) the click of the OK button of the dialog window
   - Include the OnClick button event handler in your sequence diagram. However, it does *not* include the process of selecting an item to edit – start from the OnClick of the Edit button
   - It must also include the notification to the main form of the update
   - If you have significant code in your browser you must show this as well, plus delineate that it is occurring in the browser

Remember to delineate and show the tiers in your sequence diagrams

You must show the following information in the UML sequence diagram:

- The methods called, their order of calling and the objects that the methods exist in, delineating the boundary between objects (ie: standard UML sequence diagram)

- Group the objects by the tier that they reside in and delineate the boundaries of the tiers. This is to show that you understand where the objects are physically located.

There is no need to show what parameter information is being passed – just trace the execution of the order that functions are called in and what objects these functions exist in. Show calls to WarehouseDLL functions but do not show the details inside of WarehouseDLL (since that is not your design anyways). Additionally, show calls to *important* .NET functions but ignore calls to minor .NET functions. Important .NET calls are those that either involve user interaction or generate notable 'side-effects' such as thread creation (eg: MessageBox.Show, Form.Close(), asynchronous calls, completion callbacks, major GUI updates such as adding items to a ListView, etc). Minor calls are those that are not necessary to be shown in order to understand the flow of processing of the system (eg: calls to the Convert class, individual GUI updates such as setting the .Text fields, etc). If you find that you need to show these minor updates for the UML diagram to make sense, your functions are too large and do too much – make smaller functions that reflect the processing at a better level of detail.

See the Hints in Appendix A at the end of this assignment for tips on doing your design.

Critical Self-Analysis

In addition, you are required to discuss the pros and cons of your design choices. Aim for a discussion of at least two pages (single-spaced, 12pt, 2.5cm margins). Your discussion should not be only about measuring how well your system works right now in the labs, it is about considering how well it will cope in more difficult circumstances such as low-bandwidth networks, dozens of client PCs simultaneously connected, etc. Structure your discussion so that you examine the trade-offs you have made in ***at least*** each of the following aspects of your system:

- Network efficiency / bandwidth usage
- GUI responsiveness
- Design and implementation elegance / simplicity
- Memory usage on servers and clients
- Others…

PLEASE NOTE THAT THERE IS NO PERFECT IMPLEMENTATION: You *will* be forced to make compromises. Even if you don't realise that you made these compromises, they are happening. Sure, what you've done may be 'optimal' in the sense that it is better than any other possibility, but don't pretend that there are no downsides to your choices. The point of the self-critique is to spend time figuring out what compromises you made (consciously or unconsciously) when making your choices. So do not imply that you have no compromises – they may not be terrible, but they exist and I will see them even if you do not (remember, I will be taking apart your code).

Note that this is worth 15% of the total. So you must provide a thoughtful and intelligent discussion of what you did, why you did it and what was inelegant / inefficient about it. Do not simply put down one section per aspect (network, responsiveness, etc) – they are all interrelated since choices to improve one aspect affect the others.

Marks Breakdown (Detailed Design):

• UML Sequence diagram: Display	4 marks
• UML Sequence diagram: Add	6 marks
• UML Sequence diagram: Edit	6 marks
• Critical Self-Analysis	15 marks
Total:	**31 marks**

3.D3. IMPLEMENTATION

3.D3.1 Code:

This is the code part of the assignment. Zip the entire contents of your project directory(s) up (source code and compiled executables) and upload the zip file into the Assignment submission area of the Assessments section in Blackboard. You may upload the assignment multiple times if you press Save, but when you press Submit you have committed and cannot try again.

Ensure that Blackboard's copy is OK – download and unzip it yourself after submission to check, and let me know *immediately* if there is a problem. It is ***strongly*** recommended that you also keep a copy of the projects in your I: drive or Linux home area – that way if a problem occurs I can sit down with you and verify that the timestamp on the files in your home area pre-dates the deadline time, and will then copy them over manually. I cannot do the same for your personal PC / thumbdrive since you are able to change the date of the files.

Note that I will be using my own copy of the WarehouseDLL.dll, so there is no point making changes to WarehouseDLL.dll (unless it is a bug fix, and then you should tell me!)

3.D3.2 Report:

In the report just provide a brief description of how to run your application and a summary of major known problems with the application (eg: which functionality isn't implemented, places where it will crash or lock up, etc) so that I can run it and avoid problems.

Marks Breakdown (Implementation):

• Good commenting	4 marks
• Clarity	4 marks
• Consistency	3 marks
• Coverage of user requirements	

- System is distributed — 3 marks
- Shared/singleton data tier — 2 marks
- Bandwidth usage — 5 marks
- Display pagination – native GUI — 8 marks
- View/Add/Edit – native GUI — 10 marks
- Display pagination – Web interface — 8 marks
- View/Add/Edit – Web interface — 10 marks

- *Class differs from UML design* — *up to -3 marks per class*

Total: **57 marks**

GRAND TOTAL: 100 marks

I will be looking through your code and marking you on how easy it is to read and understand your code as well as how much of the system you actually managed to implement. The first three (commenting, clarity, consistency) and last one (Class differs from UML) are described in more detail below:

Commenting: Ideally I should be able to skim through the comments in your code to understand what the code is supposed to be doing. Good commenting has several aspects:

- Briefly describes the purpose of blocks of code, rather than restating every line of code.
- Explains not just *what*, but also briefly summarises *why* and *how* for the more difficult parts
- Accurately reflects the code

Clarity: Names should be meaningful. Moreover, avoid unnecessarily complicated code since it's unmaintainable and makes it hard for people to understand your code. I find that for complicated bits, if I write the comment first I am forced to do a mini-design and think my way through solving the problem, and the resulting code is far less messy. So think before you code! And if you later figure out a better/simpler way of doing something, refactor the code to do so (it'll make later coding a lot easier and faster).

Consistency: Variables, classes and methods should be named according to some kind of standard and you should stick with that standard. Also, similar structures in different parts of the code (eg: loops through an array) should follow similar styles – since you are writing all the code this shouldn't be too hard to ensure.

Class Differs from Design: I will also be checking that your implementation reflects the UML diagrams provided – each class that differs from the design will lose you up to 3 marks, depending on how significant the difference is. The easiest way to ensure you don't lose marks is to generate your final (submitted) design based on your actual code (particularly your UML sequence diagrams). *This is not cheating.* Nobody does perfect designs in one go – designs are *always* modified to take into account problems found at the coding level. Designs are not just to help you implement the system, they are also there to help *other people understand your implementation*. See the Hints of Appendix A for more on this.

Appendix A: Hints

- Section 2 (System Functionality) lists the requirements in the order that I suggest you tackle the assignment in. However, feel free to prioritize for yourself the requirements – use the marks to help you in your choices.

- Do NOT leave the Web interface until the last minute – you will find that it is in some ways more difficult than the native Windows Forms interface.

- Note that I will be using my own copy of the WarehouseDLL.dll, so there is no point making changes WarehouseDLL.dll code (unless it is a bug fix, and then you need to tell me!)

- Do not try to build the perfect UML design up-front – you simply do not have the experience to create a rock-solid design that can be translated into code without hitting unforeseen problems. Instead, come up with an initial rough design and begin programming from it – you will then start finding out the flaws in the design! Then alternate between design and programming to slowly refine your application, refactoring (rewriting) code when your design changes demand it and adjusting your design when your code shows up problems. Then produce the final UML class diagram from your code – this becomes part of the documentation of your application, not all the iterations leading up to it. Similarly, the final UML sequence diagrams should be a simple matter of translating the final code into call sequence diagrams.

- In the native Windows Form interface, the dialogs must be Modeless. However, this adds the problem that the main form isn't notified about when the dialog is finished (unlike Modal dialogs). Thus when the dialog is Saved/Added, it must notify the main form that a change has been made. One way to do this is to define an OnItemUpdate method in the main form and pass a delegate pointing at this to the constructor of every dialog that is created. Then the dialog can call this delegate (must store the delegate as a member field!) upon saving/adding.

- You'll probably want the dialog to do the actual save to the database rather than expect the main form to do it (because Adding is different to Saving).

- It is a good idea to create a 'manager' or 'controller' object for each tier that will control the creation and interplay between the other, data-oriented objects. They will also form the 'backbone' of the application, with managers connecting to each other across tiers and acting as the go-between for objects in one tier to gain access to the functionality available in another tier.

- Make the data tier manager a Server-activated Singleton (ie: one object services all clients) via RegisterWellKnownTypeService; see Practical 5. This is the only way to

ensure that all clients connect to the same database (since the database is in-memory and not in an external DBMS). It also allows the data tier to perform thread-safe locking / synchronisation and checking LastModified so as to avoid lost updates on the database. Otherwise each GUI client / business tier will be working with a different data tier and hence a different in-memory database, which is not what you want!

- For optimistic locking, keep the LastModified field in the object whenever a data-related object such as an Investor is loaded. Then when saving the object back after editing, check if the object's LastModified field is still the same as the database record's LastModified (which will require a second read just before the update). If they are then you can save the records without causing a lost update, otherwise throw an appropriate exception.

- I suggest that you only load the parts of the database that you need at any given moment and reload them whenever the user wishes to see details. This is more than just for network efficiency reasons: for example, if the user selects an item to edit, *reload* that item from the database even though you have at least some of the information in memory already (due to the listing). If you don't, the item might be out-of-date (eg: if the user has left the GUI listing sitting there overnight and another user changed it overnight), so your optimistic locking will become downright annoying for the user since they'll constantly be told to try again since another user edited the record (and you aren't ever reloading the latest version!)