# ITC205 A2 Code Overview

Dear All – it has been suggested that some documentation or an overview of the code base for the assignment might be a help. So here goes:

The application implements a simplified ‘library system’. Many of you may be familiar with the Backwoods Regional Library example from ITC203 – this application essentially implements parts of that system.

The overall architecture is that of a standalone ‘3-layer’ system. By 3-layer, I mean that the UI classes are separated from the business logic control layer, which is separated from a lower level ‘entity’ layer. I put ‘3-layer’ in quotes because there is no separate data layer – that would add to the too much complication to the code for this exercise.

The app supports several use cases. These include administrative, and operational functions. You should be familiar with these from ITC203.

Administrative use cases include:

1. Add a member to the library
2. Add a book to the library

Operational use cases include:

1. Process taking out a loan for a book or books.
2. Process a returned book
3. Process fixing a damaged book
4. Process a payment for any overdue or damage fines

There are also a couple of ‘helper’ functions that allow you to test the application interactively. These include:

1. List all books in the library
2. List all members in the library
3. List all current loans
4. Increment the current date (this allows loans to become overdue)

You can check out all the functionality by running the Main class. At this point the persistence mechanism should be mentioned.

The app uses a very basic form of persistence. All entity classes are Serializable. This allows the Library class which contains references to all other objects to be written out to a file, and by the magic of java serialization protocols, all referenced objects are written out as well – and references between objects are preserved.

The catch to using this very simple approach is that it is sensitive to exactly what class was saved, and what class that save file is being loaded to. If you change the definition (i.e. code) of any of the saved entities, then the save file won’t load – instead it will generate an error. What this means is that you have to delete the save file (‘library.obj’) every time you update any of the entities, or the next time you execute Main (say to test that you haven’t broken anything) it will fail with an ‘InvalidClassException’. Its not a problem – simple delete the library.obj file and run main again. You may need to refresh your IDEs view of the development directory so you can see the library.obj file.

OK – lets talk about the structure of the app a bit more.

The implementation for the administrative functions and the helper functions is very basic – essentially the business logic is contained in Main, and the appropriate entity methods are called directly.

However, the app uses the ‘mediator’ or ‘control class’ pattern to separate the UI classes from the entity classes for the ‘operational’ use cases. The control classes used control the flow and logic of the use cases implemented. This is intended to demonstrate how to implement the ‘control class’ pattern referred to in ITC203. This pattern is portable to using a GUI interface – i.e. an ‘event driven’ interface.

What this means is that the UI classes are written using an ‘event loop’ where what happens when an input is received is interpreted according to what state the UI is in. Remember how I’ve said that the way an object controls how it responds to events is through object state? This is an example of an object doing that. The UI knows which method of the control class to call because of which state it is in.

How does the UI state get set? The control class does that, because it is the control classes job to control the flow and logic of the use case – so it’s the control classes job to decide what state the UI should be in. Keeping business logic out of the presentation layer.

If you look at the control class, you can see that every time some ‘system operation’ gets called on the control class, the control class sets the UI to a new state – unless the input wasn’t valid, in which case some error message is sent and the UI remains in the same state. You’ll see that the control classes update their own state as well, so that only valid sequences of method calls are processed.

Overall, the pattern implements pretty closely the message sequence that you might have drawn for the use cases if you did the weekly exercise for ‘use case realization’ in ITC1203.

The Library class itself is implemented as a ‘singleton’ pattern. What this means is that there is only ever one instance of the Library class in the app, and you get a reference to that by calling a static ‘getInstance()’ method. This ensures that any client of the Library gets a reference to the same instance – without having to pass a reference to the Library as a parameter. Singletons sort of act like globals in an OO program. It also allows the class to check if there is a save file and load it – once, on initiation. In this app, the save file is updated when the program is shut down – but the save function must be called from Main, it doesn’t happen automatically.

I’m not sure how much more people need or want to know – that’s an overview of the structure of the app, which should help you understand what is going on, I hope.

Final note regarding updating the code: stick to the suggested divisions and sequence of updates. Do refactor renames to change variable and method names – but ONLY for the variables and method names that belong to the class you are modifying – NOT method names in classes your current class uses (I.E. not for classes that other people are responsible for updating). Accept that your changes are going to cause conflicts and debug issues when merged with other peoples work. That is intended so that you gain experience in deconflicting files and debugging merges. Don’t try to take shortcuts, just stick to the protocols that will be explained and take it step by step. Shortcuts inevitably end up in horrendously complicated tangles. The merge protocols that will be explained exist for a reason.