**TECHNICAL REPORT**

**for the**

**INDIVIDUAL MINI PROJECT**

**COMP61532**

**Pattern-Based Software Development**

**9th May, 2011**

**UDOISANG BLESSING SUNDAY**

**7588349**

**ADVANCED COMPUTER SCIENCE & I.T. MGT**

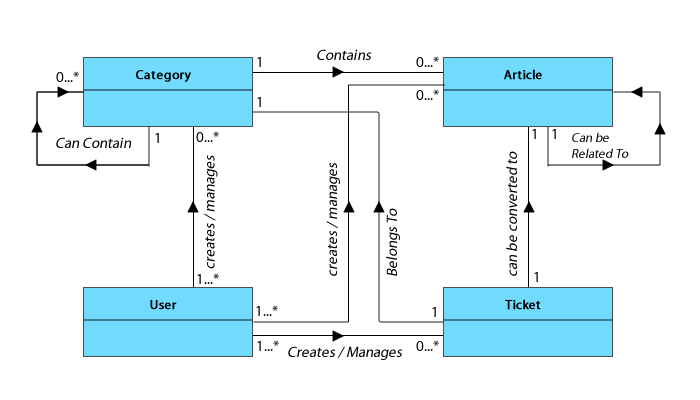
**1.0 INTRODUCTION**

An Issue Tracking System (ITS) is a software application used in an organization to record and track the progress of every problem from when it is identified until it is resolved; for example, logging customer complaints/requests in an organization’s customer support call centre. Additionally, an ITS usually contains a “knowledge base” which is a centrally managed repository of articles containing diverse kinds of information such as resolutions to common problems, customer information, etc. **1.1 DESCRIPTION AND SCOPE OF THE APPLICATION**

I focused on managing the “knowledge base” of the issue tracking system. This includes displaying knowledge base articles; adding and editing articles. Each article is classified under a category. Categories can have sub categories containing articles also. The prototype application uses a local XML data store for persistence.

**1.2 THE DOMAIN MODEL**

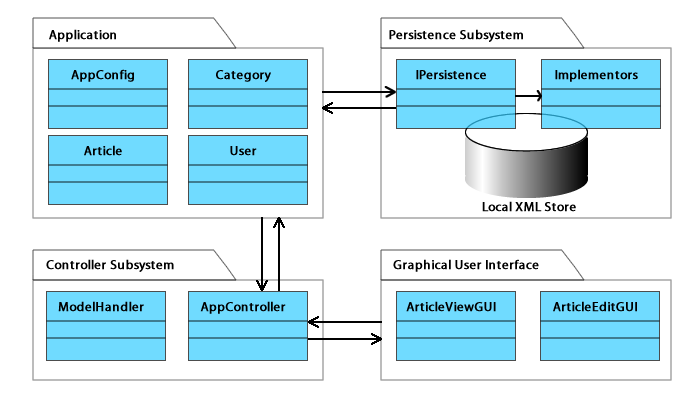
Light-weight domain modelling was done to gain a better understanding of the prototype application to be developed. The partial domain model is shown below.



**Figure 1 – Partial domain model**

**1.3 OVERALL ARCHITECTURE OF THE APPLICATION**

The prototype application developed is divided into four subsystems as shown below. The application subsystem contain the main classes required to run the application. The persistence subsystem contain classes that manage object persistence for the application. The objects are persisted in a local XML file. At runtime, the objects are materialised via the Simple API for XML (SAX) using XML Mapper classes for each type of object to be materialised. The controller subsystem implements the “Controller Pattern” thereby ensuring the graphical user interface classes do not directly interact with application classes. This ensures protected variation, low coupling and high cohesion. The graphical user interface subsystem contains GUI classes for interacting with the application via the controller subsystem.



**Figure 2 – Overall system design**

**2.0 TECHNICAL DETAILS OF PATTERNS USED**

The following design patterns were implemented in the prototype application

* The Bridge pattern
* The Composite pattern
* The Observer pattern
* The Singleton pattern
* The Controller pattern

**2.1 THE BRIDGE PATTERN**

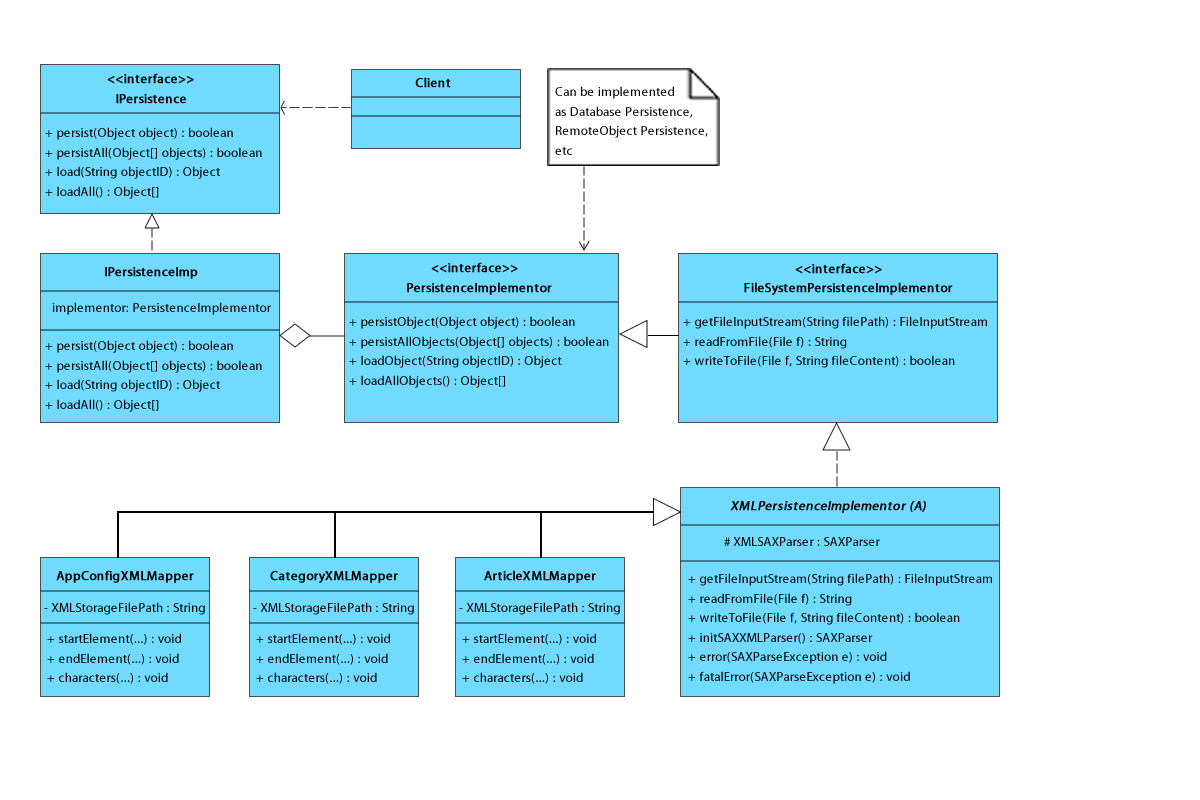
**Usage**: The bridge pattern was used to decouple the abstraction of the persistence layer from its implementation. This enables both of them to vary independently.

**Structure and Implementation**:

The “IPersistence” interface provides the abstraction for object persistence in the application. This abstraction is implemented by “IPersistenceImp” using an “*implementor*” object which is defined by an implementation interface, PersistenceImplementor. I extend the PersistenceImplementor to create a “FileSystemPersistenceImplementor” Interface. This interface is then implemented by the “XMLPersistenceImplementor” class which is also defined as a subclass of DefaultHandler (The class responsible for SAX Operations in Java). XMLPersistenceImplementor is abstract and provides default implementations for some SAX API methods. The concrete XMLMapper classes (ArticleXMLMapper, CategoryXMLMapper and AppConfigXMLMapper) extend XMLPersistenceImplementor, implementing the necessary SAX API methods for materialising objects from the local XML files.

**Sample Usage**:   
IPersistence persistence = new IPersistenceImp(new ArticleXMLMapper());  
persistence.loadAll();

**Justification & Consequences**: Decoupling the abstraction of the persistence layer from its implementation is necessary because by so doing, different kinds of persistent storage (e.g. RDBMS, Remote Objects, etc) can be easily used for the application. Normally, it would be easier to simply create subclasses of an abstract PersistenceMapper class for the different storage types. However, inheritance strongly couples the implementations to the abstraction thus making it difficult to modify, extend and reuse either the abstraction or the implementation independently. Modifying one would always lead to modifying the other. The bridge pattern solves this problem elegantly. Although only local XML storage is implemented in this prototype, in the future, other kinds of file system storages (e.g. YAML, CSV, etc) and other kinds of storage types such as RDBMS can be easily implemented without changing the existing structure.



**Figure 3 – Implementation of the Bridge Pattern**

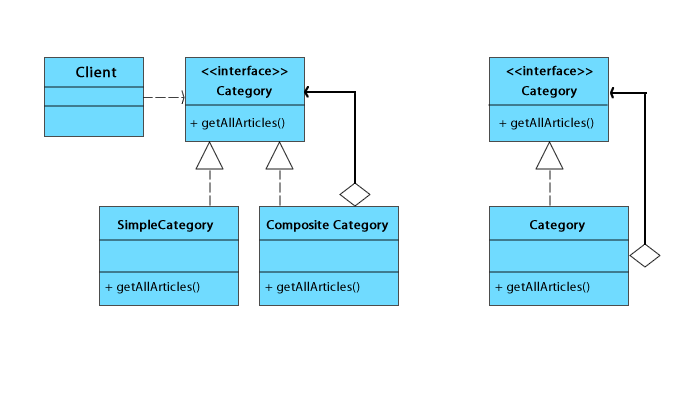
**2.2 THE COMPOSITE PATTERN**

**Usage**: I used the composite pattern to compose the nested tree structure of the article categories in order to represent them as a part-whole hierarchy.

**Structure and Implementation**:

Each article in the knowledge base belongs to a particular category. A category however, can contain other categories which themselves can contain other categories. This nested tree structure has been implemented by defining a composite pattern for the categories enabling me to treat both "branch" and "leaf" categories uniformly. In this case however, I have adapted the design pattern such that a single concrete class that can support both leaf and branch operations is used [Wirfs-Brock, 2006 and Omercan, 2010]. This helped me manage the recursive structure of the categories better.

**Sample Usage**: Article[] allArticles = category.getAllArticles();



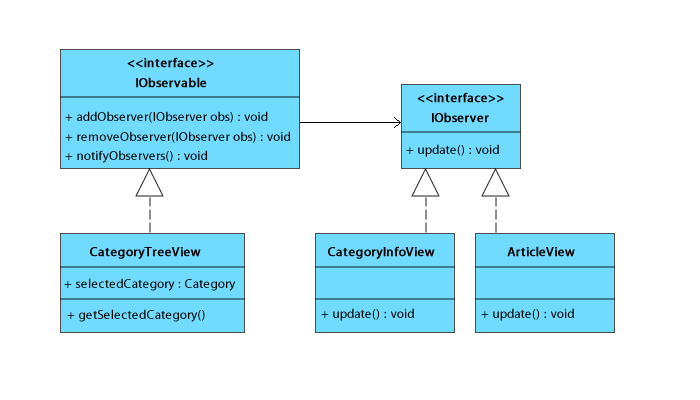
**Figure 4 – Implementation of the Composite Pattern**

**Justification & Consequences**: Composing the categories into a part whole hierarchy lets clients (e.g. the GUI) treat individual categories and composites uniformly. It also make it very easy for me to add new categories or remove existing ones.

**2.3 THE OBSERVER PATTERN**

**Usage**: The observer pattern is used to monitor the state of the category tree view (the observable) in the graphical user interface. The state being monitored is the selected category from the tree. When selected, observers were notified and they responded accordingly.

**Structure and Implementation**:



**Figure 5 – Implementation of the Observer Pattern**

The CategoryTreeView GUI implements the IObservable interface which enabled it to attach and detach listeners. The CategoryInfoView and ArticleView implement the IObserver interface which allowed them to be listeners to the CategoryTreeView object. When a category is selected, CategoryTreeView notifies all listeners attached to it. CategoryInfoView when notified, displays some useful statistics about the selected category and also gives a full description of the category. ArticleView on the other hand lists all the articles in the selected category.

**Justification & Consequences**: The observer pattern enables the GUI objects to monitor the state of the selected category without being coupled to the CategoryTreeView class. This ensures low coupling and high cohesion which also guarantees protected variation. In the future I can change the GUI classes or even their implementation without changing CategoryTreeView class.

**2.4 THE SINGLETON AND CONTROLLER PATTERNS**

The singleton pattern ensures that only one model class holding a collection of objects exist per persistent object type. It also ensures a global point of access to the models (and objects).

The controller pattern is used to enforce a central point of access to the application from the GUI subsystem. (See figure 2). This reduces coupling between the view and the application classes. By carefully applying the controller, several views (e.g. mobile view, web view, console view, etc) can be interfaced to the application. One consequence of using a controller is bloating, which occurs as a result of the controller doing too much work [Larman, 2005] thus leading to low cohesion. To prevent this, I have ensured the controller is a delegator, calling up more specific handlers to handle requested functionalities.

**2.6 REFLECTION & AREAS FOR FURTHER IMPROVEMENT**

Applying design patterns in developing the prototype application gave me a better understanding of object oriented design. However, the following are areas for further improvement.

An index model containing only object Ids should be created for the objects for faster access and iteration. Also, too many objects being materialised from the local file store at once can lead to high memory foot print and low performance. To prevent this, I can use caching and lazy materialization via virtual proxies [Larman, 2005] to reduce (1) the number of times I have to return to the model to ask for articles listing by categories and (2) the memory foot print of the listed articles, only loading the full article object when details need to be viewed.

**3.0 CONCLUSION**

The knowledge of design patterns is a powerfully dangerous thing. It is powerful because it encourages reuse of clean, time-tested designs. This can greatly improve design, stability, performance, etc of an application. On the other hand it is dangerous because it can lead to what is called “Pattern Trap” [Sintes, 2010]. This is a situation whereby instead of solving a solution in the best way possible, the developer focuses more on implementing as many design patterns as possible hoping to get a good design by so doing. Unfortunately, a design pattern does not guarantee good design. The guiding principle should be, if a design pattern does not lead to high cohesion, low coupling and protected variations, then it might not be worth applying at all.

The aim of this mini project is to design a prototype application using at least 3 different design patterns. I designed a part of an issue tracking system; the knowledge base management system. By carefully applying design patterns, I was able to implement a prototype application using 5 different design patterns.

**REFERENCES**

Gamma, E. Helm, R. Johnson, R. Vlissides, J., (1995). *Design patterns: elements of reusable object-oriented software*. Reading, Mass, Addison-Wesley.

Larman, C. (2005). *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. Upper Saddle River, N.J., Prentice Hall PTR.

Omercan S., 2010. *Composite Pattern With Parent Reference*. [online]. Available at: <<http://www.osebboy.com/blog/composite-pattern-with-parent-reference/>> [Accessed 21 April, 2011].

Sintes T. (2010). *Speaking on the Observer pattern, how can you use the observer pattern in your Java design?* [online]. Available at: < <http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html>> [Accessed 04 May, 2011].

Wirfs-Brock Associates (2006). *Pattern drift*. [online]. Available at: <<http://wirfs-brock.com/blog/tag/pattern-repositories/>> [Accessed 16 April, 2011].

**APPENDIX – SCREEN SHOT OF PROTOTYPE APPLICATION**

**here**