TECHNICAL REPORT OF APPLICATION AND PATTERNS USED

BY UDOISANG BLESSING SUNDAY

7588349

ADVANCED COMPUTING AND I.T. MANAGEMENT

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”. A knowledge base is a centrally managed repository of articles containing diverse kinds of information such as resolutions to common problems, customer information, etc. A ticket in an ITS is a record of a single issue containing various information about the issue.

DESCRIPTION AND SCOPE OF THE APPLICATION

Description coming soon based on space remaining out of 5 pages.

I focused on managing the “knowledge base” of the issue tracking system. This includes displaying knowledge base articles, adding and editing articles and converting resolved tickets to knowledge base articles. Each article is classified under a category. Categories can have sub categories containing articles also. The application used a local XML data store for persistence. Other areas of an issue tracking system were handled by other students in the sub group.

DOMAIN MODELLING

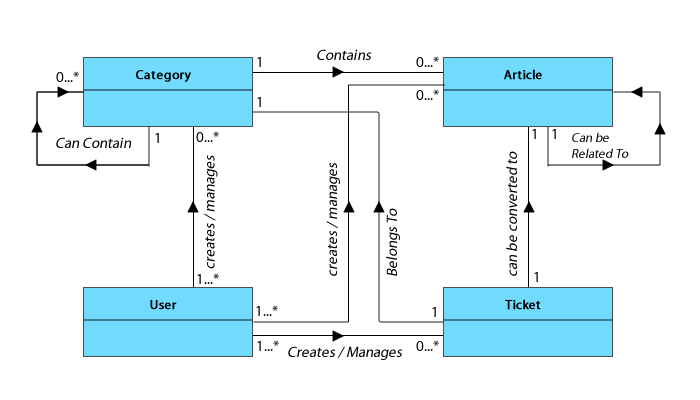
Light-weight domain modelling was done to gain a better understanding of the prototype application to be developed. A few use case scenarios and the partial domain model are shown below.

Use cases

View Knowledge Base Articles: User accesses the system. User elects to view articles. User selects an article from a category and obtains details on the article

Add Article to Knowledge Base: User obtains access to the system. User elects to add a new article to the system. The system receives input from the user, adds the article to the knowledge base and gives confirmation feedback to the user.

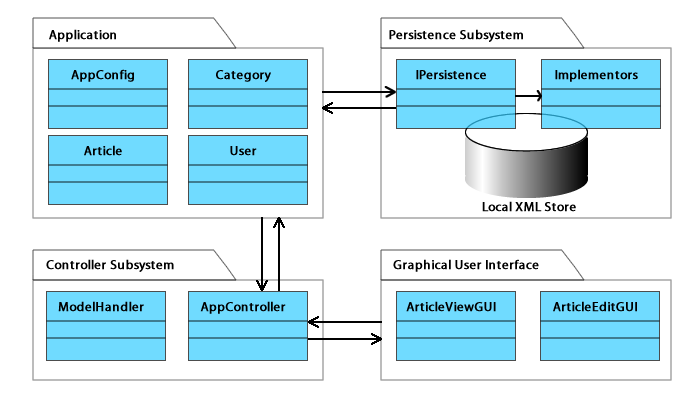
**Domain Class Diagram**



@The koko

**Design Class Diagram ([RC])**

**Overall Architecture of the Application**

The prototype application developed is divided into four subsystems as shown in the architectural diagram 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 contain GUI classes for interacting with the application via the controller subsystem.

TECHNICAL DETAILS OF PATTERNS USED

The prototype application made use of the following listed design patterns, which are clearly described with diagrams.

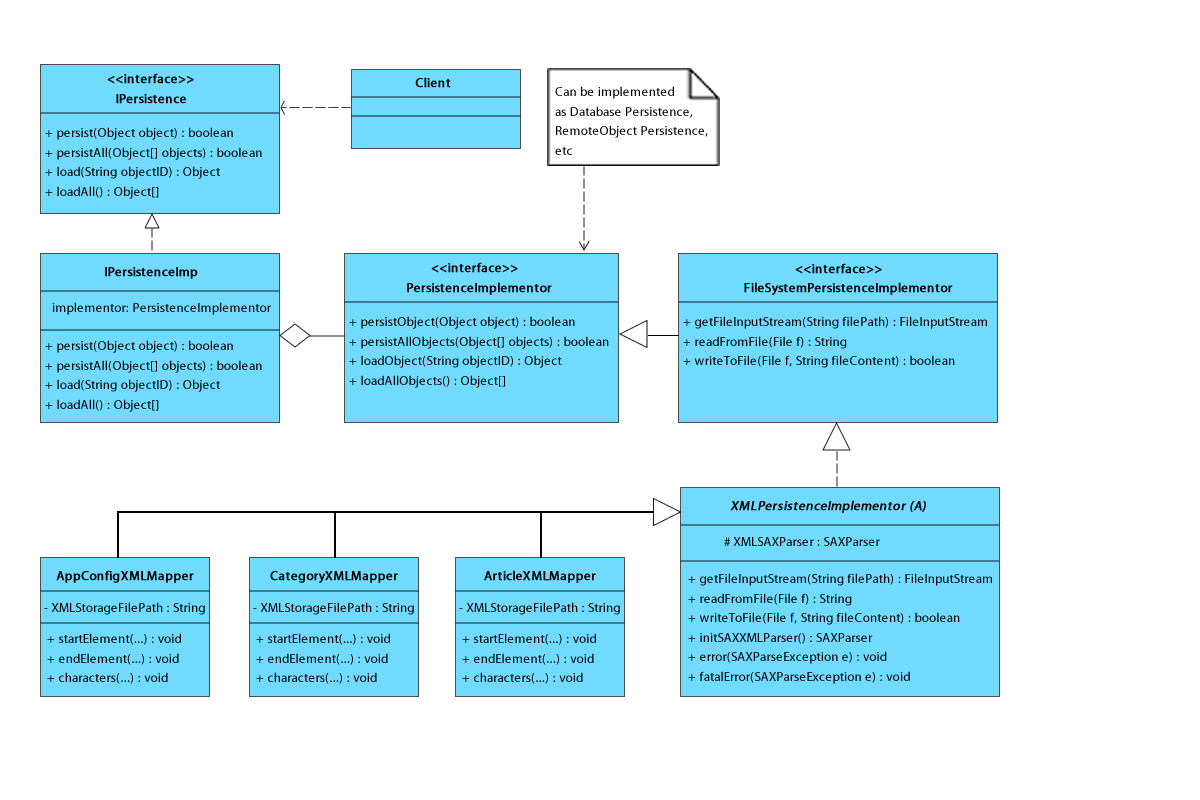
* The Bridge pattern
* The Composite pattern
* The Observer pattern
* The Singleton pattern
* The Controller pattern

**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.

Participants: @listed

Structure and Implementation: The **IPersistence** interface provides the abstraction for object persistence in the application. This abstraction is implemented by **IPersistenceImp**. **IPersistenceImp** does not directly implement the abstraction; rather, it does so via an ***implementor*** object which is defined by an implementation interface (**PersistenceImplementor**). PersistenceImplementor can be implemented for different storage formats e.g. RDBMS, File system, etc. In this case, 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 while defining other methods for subclasses. The concrete XMLMapper classes (ArticleXMLMapper, CategoryXMLMapper and AppConfigXMLMapper) extend XMLPersistenceImplementor. They implement the necessary SAX API methods for materialising objects from the local XML files. They also override necessary methods. They are also partly responsible for writing the objects back to xml files.



Sample Usage: [Might remove later, not too needful]

IPersistence persistenceAPI = new IPersistenceImp(new ArticleXMLMapper());

persistenceAPI.loadAll();

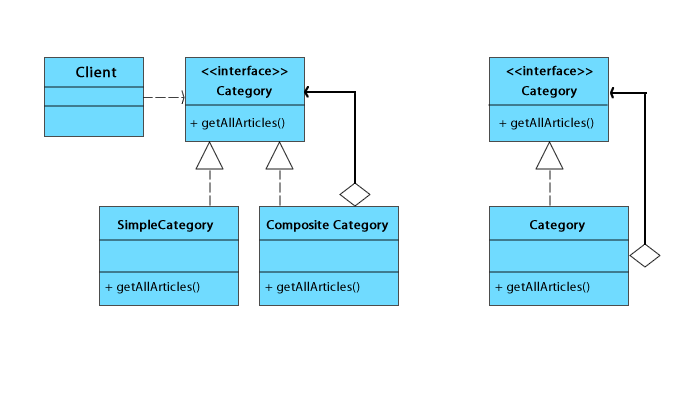
Justification: Decoupling the abstraction of the persistence layer from its implementation is necessary because by so doing, different kinds of persistent storage (e.g. RDBMS, Remote Objects, etc) can be easily used for the application. Normally, it would be easier to simply create subclasses of an abstract 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. In other words, 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 formats such as RDBMS can be easily implemented without changing the existing structure.

**THE COMPOSITE PATTERN**

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

Participants: @listed

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. This allows us 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: [Might remove later, not too needful]

Article[] allArticles = category.getAllArticles();

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.

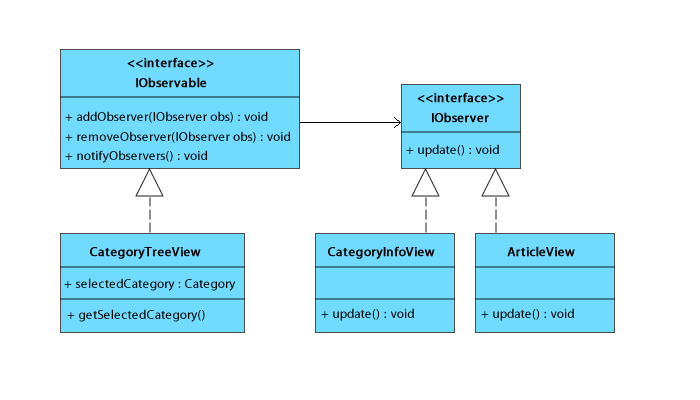
[**talk about of justification in terms of low coupling, high cohesion, protected variations]**

**THE OBSERVER PATTERN**

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

Participants: @listed

Structure and Implementation:



The CategoryTreeView GUI class displayed categories in a tree structure. This class implemented the IObservable interface which enabled it to attach and detach listeners. The CategoryInfoView and ArticleView respectively implemented 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.

Sample Usage: [Might remove later, not too needful]

IPersistence persistenceAPI = new IPersistenceImp(new ArticleXMLMapper());

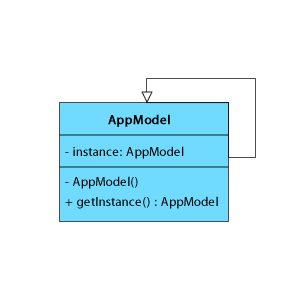
Justification & Consequences: The observer pattern enabled the other GUI objects to monitor the state of the selected category without being coupled to the CategoryTreeView object. This ensured low coupling is maintained and also guarantees protected variation. In the future I can change the GUI classes or even its implementation while nothing changes in the CategoryTreeView class.

**THE SINGLETON PATTERN**

Usage: The singleton pattern was used to ensure that only one model class holding a collection of objects existed per persistent object type.

Participants: @listed

Structure and Implementation:



The singleton pattern was used for all the model classes. After materialising the objects from the local XML files (by the XML Mappers), they are added to model classes which hold them in a collection. Each persistent class had a model such as the article, category and config models.

Sample Usage: [Might remove later, not too needful]

IPersistence persistenceAPI = new IPersistenceImp(new ArticleXMLMapper());

persistenceAPI.loadAll();

Justification & Consequences: Using a singleton ensured that only one model existed for each object type. It also ensured a global point of access to the models.

**THE CONTROLLER PATTERN**

Usage: The controller pattern was used to enforce a central point of access to the application from the GUI subsystem. The controller implemented was a light-weight container because it specifically delegated the tasks to more specific handlers for processing. That way I prevented my main controller from becoming bloated. This also ensured high cohesion.

Participants: @listed

Structure and Implementation: The

Sample Usage: [Might remove later, not too needful]

IPersistence persistenceAPI = new IPersistenceImp(new ArticleXMLMapper());

persistenceAPI.loadAll();

Justification & Consequences:

**ARTEFACTS DELIVERED**

Technical Report: this report is one of the artefacts produced.

Working Prototype Application: A prototype application has also been produced. See screen shots below. The statistics of the working prototype is as follows: SLOC, etc

**SCREEN SHOTS OF THE WORKING PROTOTYPE**

**REFLECTION & AREAS FOR FURTHER IMPROVEMENT**

<http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html>

The hammer trap? Use it for reflection

Create an index for the articles model so that when reading / listing them by category, one doesn’t have to iterate the whole collection, aggregating them, I can just iterate the index and get from the map the specified articles. Iterator Pattern could also be used here to create Iterator-By-CategoryID

Use caching and Lazy Materialization via Virtual Proxy to reduce (1) the number of times we have to return to the model to ask for articles listing by category and (2) reduce the memory foot print of the listed articles, only load full article when details needs to be viewed, otherwise, load a lightweight proxy in the JList component.

**CONCLUSION**

The aim of this mini project was to design a prototype application using at least 3 design patterns. I design a part of an issue tracking system; the knowledge base management. To understand the application, I used domain modelling to model the part of the application I wished to develop. By carefully applying design patterns, I was able to implement a prototype application using 5 different design patterns. The application has been demonstrated as expected. The source code has also been submitted. The submission of this technical report marks the successful completion of the mini-project.

OBSERVERs Vs OBSERVABLEs:

<http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html?page=2>

\*Borrow inspiration of images from here: <http://developerlife.com/tutorials/?p=28>

\*I noticed that the XML storage structure in itself represents a composite tree with each node containing a node. Can I use this info?

\*See SAX Tutorial for how to describe your classes and inspiration too. : <http://developerlife.com/tutorials/?p=29>

<http://developerlife.com/tutorials/?p=26> [Very Helpful too]

\*A simple Java.util.List could be enough to handle these things, let’s see sha

Hello

SAX: Simple API for XML

DOM: Document Object Model

<http://www.devarticles.com/c/a/XML/Java-and-XML-Basics-3/8/>

Some crazy data to support sax as better to use here than dom!

The three steps to using SAX in your programs are:

* Creating a custom object model (like Person and AddressBook classes)
* Creating a SAX parser
* Creating a DocumentHandler (to turn your XML document into instances of your custom object model).

SINGLE CLASS COMPOSITE

<http://wirfs-brock.com/blog/tag/pattern-repositories/>

“A single concrete class that could support either leaf or composite behaviors. Now that’s a thought…but is it still recognizable as a composite pattern? Sure...” [@REF Wirfs-Brock Associates (2006)]

<http://www.osebboy.com/blog/composite-pattern-with-parent-reference/>

Corroborates the idea of a Single Composite - Leaf[@REF, Omercan Sebboy, 2010]

FURTHER AREAS OF IMPROVEMENT

Create an index for the articles model so that when reading / listing them by category, one doesn’t have to iterate the whole collection, aggregating them, I can just iterate the index and get from the map the specified articles. Iterator Pattern could also be used here to create Iterator-By-CategoryID

Use caching and Lazy Materialization via Virtual Proxy to reduce (1) the number of times we have to return to the model to ask for articles listing by category and (2) reduce the memory foot print of the listed articles, only load full article when details needs to be viewed, otherwise, load a lightweight proxy in the JList component.