The University of Western Ontario
CS 3307, Nazim H. Madhavji

Assignment 2: Design Patterns

Group 8

Ziqi Rui	zrui@uwo.ca
Zhao Lin	zlin45@uwo.ca
Robert Goldfarb	rgoldfar@uwo.ca
Peter Antoniou	pantoni@uwo.ca
Sam Taghavi-Zadeh	staghav@uwo.ca

November 10, 2014

1 Business Scenario: Enterprise Archiving Solution for IT Risk Mitigation and Legal Compliance

The trends of electronic data growth naturally bring out new amendments to federal laws on data governance and compliance. Laws in the U.S and Canada from The Federal Rules of Civil Procedure (FRCP) and The Sedona Conference, respectively, outline models necessary for organizations to comply with the preservation of data integrity, as well as requiring reasonable steps to prevent data alteration or destruction. Amid litigation, parties must "define business records that are created or kept in electronic format as discoverable giving the requesting party access to them" as outlined in FRCP. This e-discovery requirement enshrines the principal that all relevant non-privileged electronic data must be disclosed to party claims or defenses given a request by the court.

To comply with FRCP's e-discovery requirements, all relevant business data should be archived to satisfy proper backup and recovery. A party's discovery request is usually entitled to unaltered data, and this means the original files should be stored in its original formats with its metadata. Traditionally, this backup process utilized storage in magnetic tapes, disk drives across multiple machines, or a combination of both. However, with data growing at enormous rates and in numerous formats, the e-discovery process using outdated technology can become time consuming, expensive, and result in the exclusion of relevant data. In the case of data spoliation due to neglect, US courts have imposed sanctions in the form of monetary penalties and dismissals.

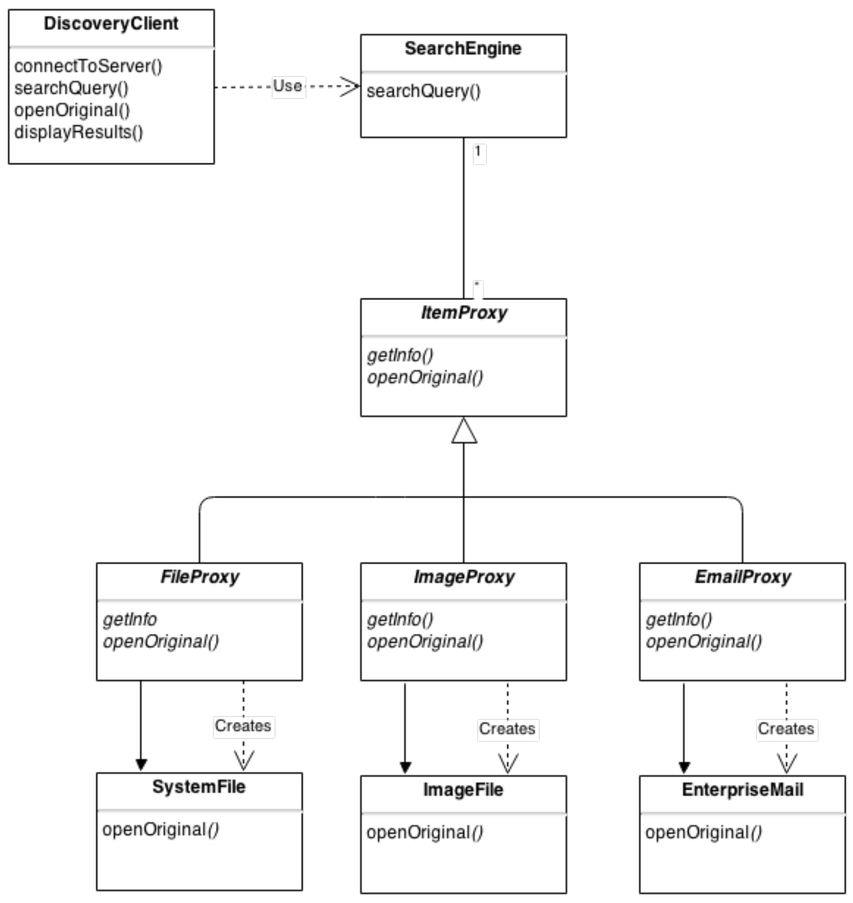
Our solution consists of a server component that processes enterprise data (emails, patents, files, etc.) and actively stores them to backup network devices. The server connects to data stores such as e-mail servers, file servers, or cloud services. Upon successful backup of a file, the server also indexes the data to a central repository and creates metadata entries to a database. In the event of a legal discovery request, supported desktop applications or web portals can make search queries to the data repository through the indexed data. This enterprise wide system ensures that all data is backed up securely and effectively while featuring flexible and secure retrievals. In the event of e-discovery obligations, firms deployed with this system would have more control over their data to mitigate legal risks.

2 Proxy Design Pattern

i) For the search and retrieval process of our system, we've chosen to use the proxy design pattern.

ii) Due to the potential nature of extremely large file sizes and the sheer number of them in large organizations (possibly hundreds of millions), it is not prudent to transmit the original files across the network during e-discovery searches; transfer of large result sets would simply not be feasible due to limiting network resources and long processing times. To address this issue, we deploy a file proxy in place of the original data file that contains only the essential metadata required for search and retrieval. In essence, discovery clients would receive results in the form of small, memory efficient, file proxies that represent the original large documents. The retrieval of the original file would therefore be delayed until the file proxy has been explicitly opened.

3 Graphical Representation in UML



4 How it works

A DiscoveryClient can be in the form of a desktop, web, or a mobile application. The client accesses the internal SearchEngine of our archive system by submitting search queries. Upon the initial search result, the SearchEngine returns -- over the network -- a set of ItemProxy items. Each ItemProxy merely represents the original file, like an alias, and contain only the essential metadata. If the DiscoveryClient were to open that ItemProxy explicitly from the list of results, SearchEngine delegates the request of opening the original file, i.e. calling the openOriginal() method, and ItemProxy will instantiate the appropriate file type for as needed.

This proxy design solves the e-discovery problem of the need to access files remotely and manage system memory and network traffic appropriately; we implement ItemProxy as both a remote and virtual proxy. ItemProxy acts as a remote proxy in the sense that locally, clients can see files on a different address space while having full access to them. It acts as a virtual proxy in the sense that expensive objects (e.g. large files, or huge result sets) are created, or transferred over the network on demand. Some implications include: the ability for multiple users to access the central archive repository; and another, the ability to access the files virtually from any computer device without high cost of network usage and congestion.

5 Skeleton C++ Code

- Code is under ProxyPattern/src
- To compile, type `make` in the same directory.
- Run ./eDiscovery.out for execution

6 Explanation of Code

The DiscoveryClient Class connects to a SearchEngine via DiscoveryClient::connectToServer(). Realistically, this would be implemented over the network. Once the SearchEngine has authenticated the client connection, the client can make queries using the call searchQuery(). The returned result set from the SearchEngine is in the form of ItemProxys. This list of proxies hold no real data and is cached in the DiscoveryClient, represented by ipResultsCache_. ipResultsCache_ is a set of proxy items that may represent generic system file types, images, or an enterprise mail. Here, the vector<ItemProxy *> container holds only the base class and by implementing a "is-a" relationship among the file types, we can use polymorphism to create derived classes using the same container.

The DiscoveryClient can then display the result cache, using displayResults(),to its user. Again, this would only display the essential meta-data of each result such as file names, email recipients, or file abstracts, etc. In our example, the proxy items only contain two properties: int id_ and string desc_ to identify the original files.

Once a user decides to explicitly open a particular file from the result set, the client can make a call to openOriginal(). This explicit call delegates the "open" call to the proxy item. Because the proxy item only represent a placeholder, it's responsible for retrieving the full contents of the original file. Due to differing location and retrieval methods of each file type, all the file types implement their own openOriginal() method.

7 Execution Trace

```
DiscoveryClient:    Starting Discovery System
DiscoveryClient:    Initiating case # 234
DiscoveryClient:    Active case: Proxy PA Ltd vs. Observer PD Corp.
SearchEngine:       Starting SearchEngine
DiscoveryClient:    Connected to SearchEngine
SearchEngine:       Processing query: All files regarding patent XY from 2000 to 2014
DiscoveryClient:    Successfully received results: 7
DiscoveryClient:    Displaying Proxy Results:
DiscoveryClient:    id: 0 System File
DiscoveryClient:    id: 1 Email File
DiscoveryClient:    id: 2 Image File
DiscoveryClient:    id: 3 System File
DiscoveryClient:    id: 4 Image File
DiscoveryClient:    id: 5 Email File
DiscoveryClient:    id: 6 System File
DiscoveryClient:    Requesting full restore of item 2
2:      Requesting original image file from proxy...
Opening original Image File: 2
DiscoveryClient:    Requesting full restore of item 3
3:      Requesting original system file from proxy...
```

```
Opening original System File: 3
```

8 Lessons Learnt

1. The proper use and implementation of the proxy pattern
2. Sometimes it's not sufficient for a system to just work, but we must also need to consider performance and resource management requirements
3. We can use design patterns, such as the proxy pattern, to better abstract away specific details where not needed to simplify the system. In our scenario, the proxy item acted as an abstraction away from the resource intensive item. The search client, upon receiving the search results, was only concerned with the general, abstract, details in its result set.

1 Business Scenario: Public Online eCommerce Bidding Solution

As online bidding sites and eCommerce sites are getting more and more popular, older large monolithic designs do not work efficiently and there are several improvements that can be made.

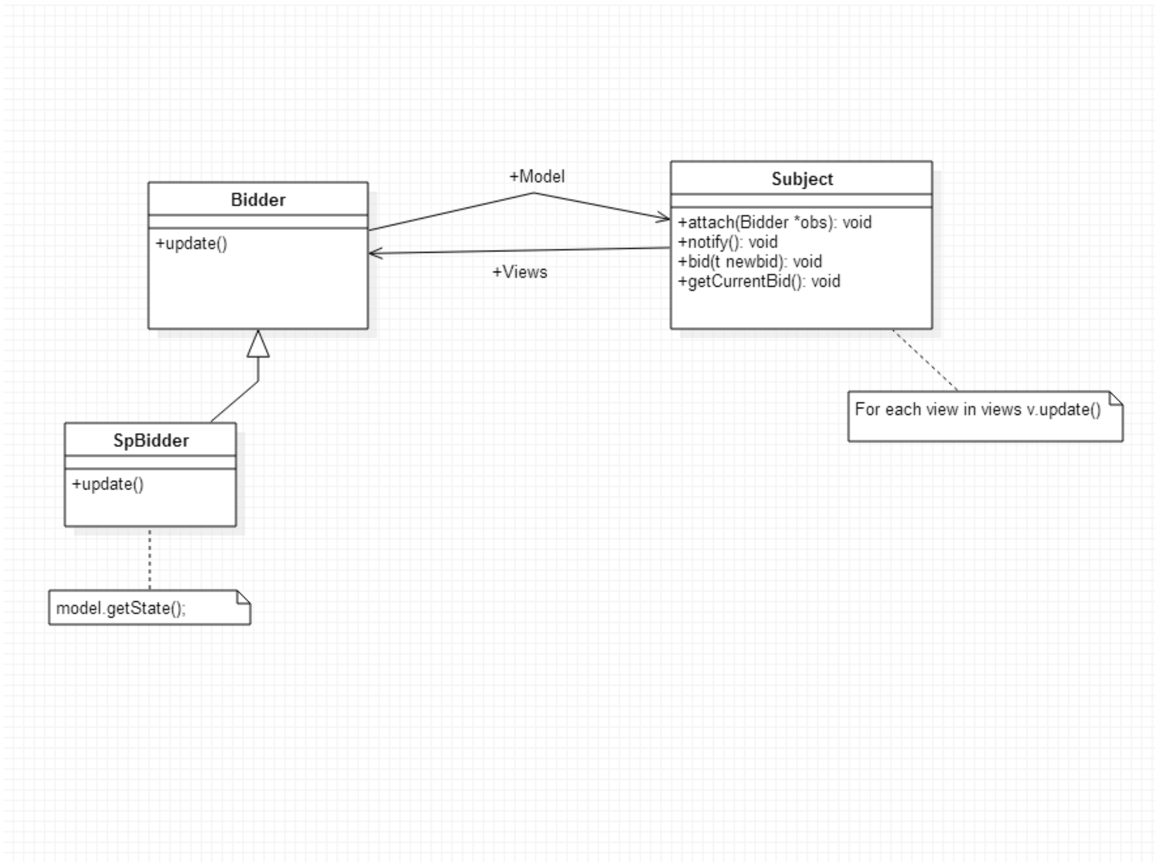
Our solution consists of several one to many dependencies that detect when an object changes its state so that the children can be warned and refreshed right away. The underlying engine of the system is this encapsulated and placed in an observer hierarchy.

2 Observer Design Pattern

i) The design pattern that we are going to use for our scenario is the observer design patter.

ii) Due to the sheer size of the Subject in this case (price of object for sale) and multiple candidates succeeding from the subject (bidders), requiring a form of inheritance is an extremely bad design because any type of bidders bid is a candidate for the object and because multiple inheritance isn't always used, the subjects already will have a superclass and overwhelm the system. Thus we use the Observer patterns amazing quality for the subject is the core that prompts the observer and the observer is the variable for abstraction, which calls back to the subject.

3 Graphical Representation in UML



Adapted from the general Observer pattern UML Source: http://huiyu.tistory.com/m/post/20

4 How it works

In an online auction, the amount of bidders is theoretically infinite. The system needs to be able to not only have a massive theoretic capacity but keep track of the highest bids in a way the least taxing to the system. In this observer pattern, all the bidders or subjects are kept in a list and are automatically notified in case of any changes by calling their methods. Also this allows for items to communicate each other without depending on one another.

5 Skeleton C++ Code

- Code included in ObserverPattern\src folder
- Run `make`, and `./observer`

6 Explanation of Code

2 main classes exist: Subject and Bidder. A Bidder object is an instance of the class Bidder that represents a bidder in an auction. The bidder can instantiate itself and gather information from the subject. The sub-class SpBidder was made to demonstrate how the program would work with subclasses assuming there were different types of bidders in the auction (ie. ones with priority, different currencies...).

A Subject object is an instance of the class Subject and represents the 'auction house'. The subject class contains a vector of all the bidders and uses this to update all of them. This is the only example of coupling between the two classes in this particular design. A bid is made through the bid() function which is part of the Subject class. Once a bid is made, all the bidders are notified through the notify() function.

The Bidder class is the Observer and the Subject class is the Subject. The above explanation demonstrates how the Subject is the core that prompts the observer and the observer is the variable for abstraction which calls back to the subject.

7 Execution Trace

```
Observer: ADD NEW BIDDER
Subject: NEW BIDDER ADDED
Observer: ADD NEW BIDDER
Subject: NEW BIDDER ADDED
Subject: UPDATE BID
Subject: NOTIFY BIDDERS: NEW BID: 14
Subject: NOTIFYING BIDDERS...
Observer: GATHERING NEW BID INFO
Observer: GATHERING NEW BID INFO
Subject: UPDATE BID
Subject: NOTIFY BIDDERS: NEW BID: 7
Subject: NOTIFYING BIDDERS...
```

```
Observer: GATHERING NEW BID INFO
Observer: GATHERING NEW BID INFO
```

8 Lessons Learnt

1. We have learned that having multiple inheritances can overwhelm a system and have drastic effects on performance. We also learned the importance of an observer pattern when we have multiple objects communicating with one-another but we do not want them to become dependent on one another.
2. We have learned about the difficulty of trying to decrease the amount of coupling. Often times a certain level of coupling will be required and the difficulty is about figuring out WHERE to include it that makes the most sense.
3. Choosing an appropriate design pattern is not always straight forward because there comes instances where multiple design patterns will be appear to be suited for the job but the user must weigh the pros and cons of each pattern. This requires a thorough abstract understanding of the project at hand to make an appropriate decision.