Analysis of Lab_#1: Hold Reference and Do Dirty Work, A Bidirectional Observer Pattern

1 Analysis of Lab

In this lab, as we will be realizing **case 4**, we are required to implement a stock-investor program which is capable of providing the abstraction of behaviour like customer buys a stock, stock company raises up price and customer sells the stock to earn some money (in response to the changing price). There are already some patterns like *Observer Pattern* that sets the mechanism for single direction notifying and being updated. However, the former pattern is lack of effective strategy for situations resembling, say the back updating requirement of the stock price changing for the company when they come up with the infomation that their products are popular as they can force it up or vice versa, aiming at which the former pattern can never cope with, so we propose a bidirectional pattern, that is to say, to hold references on both sides, solving this problem. Note that we use Java language in this project and features of components will be in certain format [1]

2 Overall Design

In this section, we will illustrate the overall design of our lab work. From implementing abstrations on basic *Observer Pattern* to modifying class contents to x the back calling without reconstructing the concrete frame of the subject-observer structure.

2.1 Understanding Observer Pattern

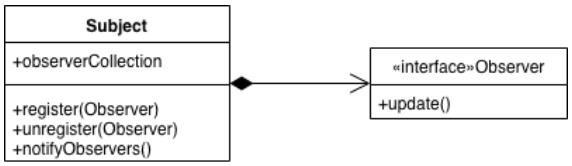

Figure 1: UML Diagram of *Observer Pattern*, there are totally two interfaces (abstract classes) with a relationship of composition, one should implement interfaces at left side of the connection if it focuses on functions of being observered and maintaining a collection of its observers, while the other should expose an interface for being notied.

So basically the *Observer Pattern* simply emphasize a loose coupling connection between the two. Notice there it is only one channel for single way message passing so long as the subject does not expose an *update()* interface and adding new observer leads to no alteration in original code, conforming to open-close principle.

2.2 Why Original Observer Does Not Work

It seems hard to implement a single directional pattern on concrete examples like meeting the requirements in this lab which is in need of a message response for situations like client selling the stock to earn money (may incur a chain of calling). Thus, we propose a *Bidirectional Observer Pattern* to solve the case 4.

[1] package, `class`, **le** , *method*, *pattern*

2.3 Bidirectional Observer Pattern

Conceptually a *Bidirectional Observer Pattern* is an idea consisting of holding a series of references for passing information across classes and exposing specific public interfaces for other subjects at sides so long as the individual class is hoping to sending messages and being called with notifications from others at the same time.

Say the `Subject` in *Observer Pattern* maintaining a collection of observers in side of its attribute list: $\{(S, O_i) | O_i \in \{O_1, O_2, \ldots, O_n\}\}$ [2], while the sum relationship of subjects is a collection: $C_{S \to O} = \{(S_j, O_i) | O_i \in \{O_1, O_2, \ldots, O_n\}, S_j \in \{S_1, S_2, \ldots, S_m\}\}$ for there are n observers and m subjects in the space, for at maximum of $n \times m$ tuples.

On meeting the requirements of the lab, `Observer` also maintaining a set of references in side of its attribute list (*HashMap* here, for an additional mapping from `SubjectStock` to `ClientLogic`), as $\{\Big(O_i, (S_j, L_k)\Big) \mid (S_j, O_i) \in C_{S \to O}\}$,where $k = j$ and $L_k \in \{L_1, L_2, \ldots, L_{mn}\}$. To be precise, there are $2 \times m \times n$ tuples in our design, without considering the mapping subset.

3 Method

In this section, we mainly focus on detailed implementation of our pattern. We are trying to illustrate some key components of projects and pieces of tricky code. Also, disadvantages of this design will be discussed.

3.1 Key Components and Detailed Design

3.1.1 Core Classes

Considering key components, we've got (default package):

- `SubjectStock` described in 6.1, the only task for this class is to maintain a list of `ObserverClient` in *ArrayList* inside of the class and expose a set of interfaces for the client to check price and do unregistering, *etc.*
- `ObserverClient` is described in 6.2. The only task for this class is to maintain a collection of mappings from `SubjectStock` to `ClientLogic` and expose a set of interfaces for the stock to send messages and fetch object status.
- `ClientLogic`, the value part of the *HashMap* (see section 6.3), is responsible for maintaining the selling / holding strategy for the stocks that the bargainers already bought in, which means to return a bool value to represent: whether the client want a notification or whether it would sell the stock in response to the fluctuations of prices at current state.

3.1.2 Strategies and Constraints

On meeting demands of this lab (to sell or hold), several strategies should be set:

- Mapping from sold stocks to current price (see **StockThresholdMapping.java** in helper), the method provides useful functions *amount* $\to$ *price* with a an average step function.
- Mapping from the threshold value $|priceOld - priceNew|$ representing certain thresh that the customer want to be notified with to a bool value, is described in **ClientThresholdMappingUpdating.java**.
- Being similar to above, there is also a function *int* $\to$ *bool* answering whether the customer should sell the stock or hold.

If we want to avoid situations resembling "customer purchases a stock with a selling-threshold that too low to prevent a reselling immediately once the bargain is done as he thinks the current price is high enough", we have to give a constraint to the conditions on not to sell stock if the customer harbours only one stock.

[2] We denote S, L and O to be subject, logic and the observer respectively.

3.1.3 Tricky Implementation

So we've got $S \rightarrow C$ in each instance of `SubjectStock` and $C \rightarrow (S, L)$ in `ObserverClient`, now that a stock hopes to notify a customer (client), how could he possibly make that? Obviously using a public interface (if in Java) offered by the client solves this case. Further more, the client makes the same way to send message back as `SubjectStock` just gives *register(Observer newObs)* (see 6.1). When being called *update()*, the client side just makes use of the *HashMap<S, L>*, reacting with various behaviour repectively.

The extraction of `ClientLogic` simplified the code structure with an independent instance for each of the $C \rightarrow S$. Maintaining the client's bargaining strategy and amounts of harbouring the, stock receives different answers from the client. Let's make it more straight forward, the chain of calling should be like this:

Algorithm 1 Chain of Recursive Calling

Require: $S \in C_S, C \in C_C, L \in C_L$

1: initialize $HashMap < S, L >$ and $Set < C >$ with $\emptyset$
2: customer C buys a stock S
3: add S and L to $HashMap$
4: add C to Set
5: **for all** S in $HashMap$ **do**
6: **if** L thinks C_i should sell **and** size of $HashMap > 1$ **then**
7: remove S_i and L_i from $HashMap$
8: remove C_i from Set, jump to 11
9: **end if**
10: **end for**
11: **if not** L amounts of stocks have changed **then**
12: jump to 19
13: **end if**
14: **for all** C in Set **do**
15: **if** L thinks C should be notified **then**
16: notify C_i, jump to 5
17: **end if**
18: **end for**
19: done

What's more is the postponed removal of items in collections as it avoid a concurrent exception frequently happened in situations when a internal removal is called inside a loop of a set of itself. The details of the trick is described in **ObserverClient.java**, method *update()*.

3.2 Tradeoff of Bidirection

As for bidirectional connection in between the two classes, there are lots of references pointing from one side to the other side. Actually those relationships lead to a tight coupling, whose methods seems hard to decompose or alternate with new methods inside of which.

Although it breaks the principle encapsulation in some degree, the concrete task wouldn't need too much of the robust features we learned from our courses, let alone extracting a higher level abstraction than the original structure if not necessary [3].

Apart from the tight coupling of the design, there somhow exists obstacle in the way users are trying to understand, say the chain of recursive calling and constraints in the *update()*.

4 Experiment

Testing on cases which show ablation of conditions such as selling at certain thresholds, being notified at various thresholds, individual price strategy on each stock company, *etc*, results are pasted from the console:

[3] Occam's razor: Entities should not be multiplied without necessity.

```
At this notification A's price is from 10 to 50:
D harbours 1 stock(s) (selling thresh: 0 for A) is holding A at 50
At this notification C's price is from 0 to 25:
D harbours 2 stock(s) (selling thresh: 0 for A) is selling A at 50
D harbours 1 stock(s) (selling thresh: 45 for C) is holding C at 25
At this notification B's price is from 5 to 18:
D harbours 2 stock(s) (selling thresh: 200 for B) is holding B at 18
D harbours 2 stock(s) (selling thresh: 45 for C) is holding C at 25
At this notification C's price is from 25 to 100:
D harbours 2 stock(s) (selling thresh: 200 for B) is holding B at 18
D harbours 2 stock(s) (selling thresh: 45 for C) is selling C at 100
At this notification C's price is from 100 to 50:
E harbours 1 stock(s) (selling thresh: 90 for C) is holding C at 50
At this notification C's price is from 50 to 100:
E harbours 1 stock(s) (selling thresh: 90 for C) is holding C at 100
F harbours 1 stock(s) (selling thresh: 200 for C) is holding C at 100
At this notification B's price is from 18 to 31:
D harbours 1 stock(s) (selling thresh: 200 for B) is holding B at 31
E harbours 2 stock(s) (selling thresh: 40 for B) is holding B at 31
E harbours 2 stock(s) (selling thresh: 90 for C) is selling C at 100
```

5 Conclusion

In this report of the lab, we illustrated basic paradigm of the *Observer Pattern* with UML diagram 1 in section 2. Then we analysed the capacity of this pattern in 2.2 and proposed our method concisely in 2.3. Meanwhile, we introduced our design with core components and illustrated some features of our design according to the requirements of the lab in 3.1. We also mentioned there are some drawback of our design in 3.2. At the end, we demonstrated the results, saying that we had met all demands.

6 Appendix

6.1 StockSubject

```
/* doing import */

public class SubjectStock implements Subject {
        /* attribute list declaration */
        private ArrayList<Observer> arr;

        public SubjectStock (String name, int priceLow,
            int priceHigh, int[] amountThresh) {
            /* constructor */
        }

        /* public getter for client */
        public int curPriceGetter() {}
        public String getName() {}

        @Override
        public void register(Observer newObs) {
            /* append observer into list */
            /* doing price changing */
            notifyAllObservers();
        }

        @Override
        public void unregister(Observer rmObs) {
            /* remove observer from list */
            /* doing price changing */
            if (this.arr.size() > 0) notifyAllObservers();
        }

        @Override
        public void notifyAllObservers() {
            /* print price changing */
            /* loop through list,
            * notify observer if they want to be */
        }
}
```

6.2 ClientObserver

```
/* doing import */

public class ObserverClient implements Observer {
        /* attribute list declaration */
        private HashMap<SubjectStock, ClientLogic> mapStock;

        /* constructor */
        public ObserverClient(String name) {}

        /* buy stock, core method */
        public void buyNewStock(SubjectStock stock, int amount,
            int threshSell, int threshUpdate) {
            mapStock.put(stock, ...);
            stock.register(this);
        }

        /* public getter for stock */
        public int amountGetter(SubjectStock stock) {}
        public boolean hopingToBeNotified(SubjectStock stock,
            int gap) {}

        @Override
        /* update, core method */
        public void update() {
            /* loop through HashMap<>
            * if want to sell and harbour more than 1:
            * sell stock, remove stock from HashMap
            * and itself in arr in SubjectStock
            * else:
            * holding stock */
        }
}
```

6.3 ClientLogic

```
class ClientLogic {
        /* attribute list */

        /* constructor */
        ClientLogic(int amount, int threshSell, int threshUpdate {}

        /* public getter for ClientObserver for conditions */
        public boolean considerSellingAt(int curPrice) {}
        public boolean considerBeingNotifiedAt(int gap) {}

        /* public getter for logic of client */
        public int getAmount() {}
        public int getThresh() {}
}
```