UNIVERSITY of GREENWICH
Alliance with FPT Education

BTEC

Contents

Introduction:

The article focuses on the segment display code and discusses the Observer Development Pattern code. In addition, the similarities and examples of this design's optimization and suitability were also clearly clarified based on the study of other models lacking in the design.

I. Scenario and UML Diagram

Scenario:

as we all know after each exam is given by the school. To get the results of each student's grades, the staff of that school will sum up the student's grades in 1 semester including a bar chart showing student scores, 1 student's transcript, 1 person observing the student, the principal and the student score data. The operating procedure will be explained as follows: when a student's score changes in the student's grade data, there will be an announcement to the student observer and then a notice to the bar chart showing Student scores, student transcripts and Principals and works their own way Since then the process of grading and entering scores for each student is completed in the best way. In the situation above. The observer pattern is a behavior pattern that an object, called a subject, maintains a list of its dependencies, named an observer, and automatically notifies them of any changes.

Class Diagram:

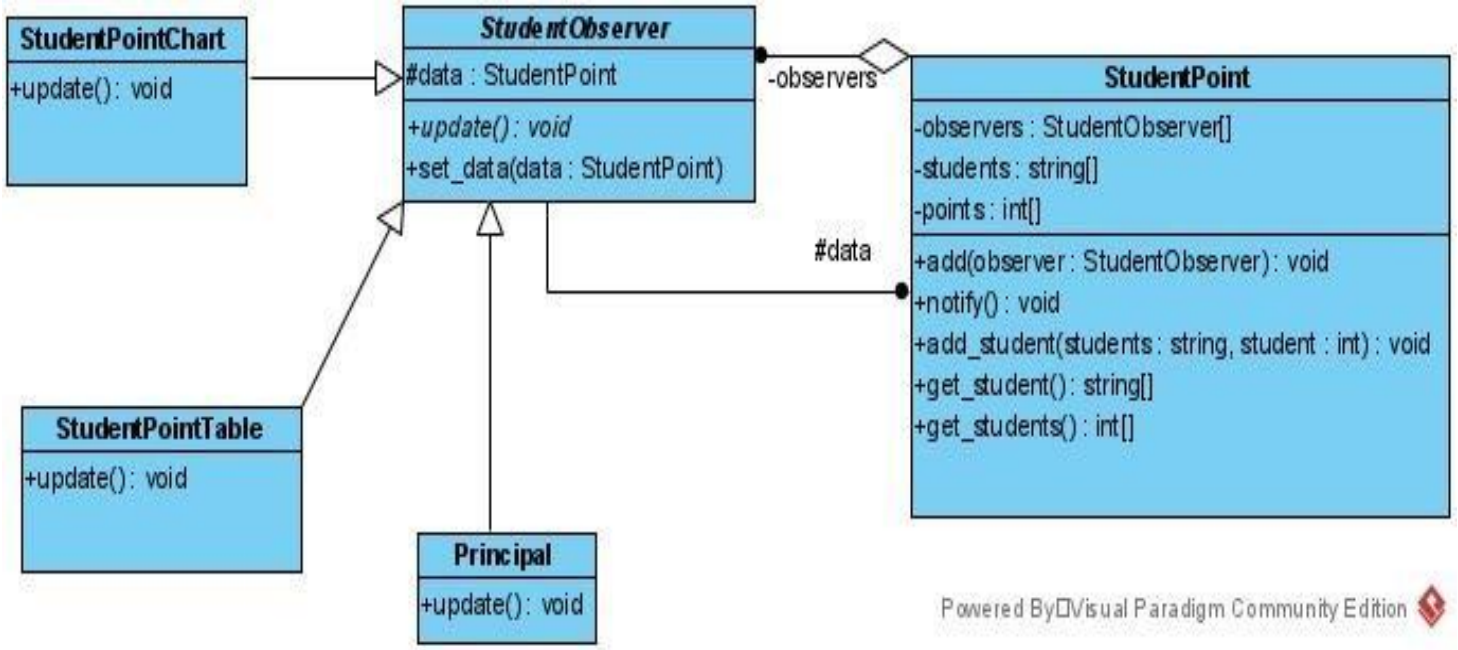


Figure 1: Class Diagram

StudentPoint here is acting as the subject, while the student is the observer. Class relationship is upgrade to aggregation relationship is container-content, one-to-many dependences between a subject (StudentPoint) and many observers (student). Inside theStudentPoint must show the methods to add student (attach), remove devices (detach), and change the Point (notify the observer is the student). The observer interface has only two method, update(): void. Whenever the StudentPoint updates the point through the add(observer : StudentObserver) : void method, it will notify the StudentObserver() to be able to notify Principal,StudentPointChart and StudentPointTable about the Point situation via update() : void method to call get_student () from StudentPoint. Besides, there is Principal, StudentPointChart and StudentPointTable class have a generalization relationship to StudentObserver class due to it inherit the method of update() to update point for student. When the System wants to add students, the program will call each particular method include add_students() through using addstudents() method of add_students () at StudentPoint class to perform add action.

II. Implementation

Code:

Because of Observer pattern establishes a one-to-many dependence between objects to automatically alert and update all of its dependents when one object changes state. So I'm just showing several important sections of code for my project that related to Observe solution in this implementation include Student(Observer) and StudentPoint(Subject). Next, this is the code for StudentPoint.h to initialize

```
class StudentObserver;

class StudentPoint
{
    private:
        vector<string> students;
        vector<int> points;
        vector<StudentObserver*> observers;
    public:
        void add(StudentObserver* observer);
        void notify();
        void add_student(const string &student, const int &point);
        vector<string> get_students();
        vector<int> get_points();
};
```

Figure 2: Code for initialize StudentPoint

All of the attributes in this class are set to private access. First, there must be an integer "students" variable to store the student calculated by StudentPoint, and a string variable "observers" is the StudentPoint final pointer must have an int "point". Additionally, StudentPoint also needs to create a "StudentObserver" pointer vector in StudentPoint class and add "user StudentObserver" to allow StudentPoint user to use this "StudentObserver" pointer to enforce it. I have set out a selection of methods I would use to incorporate StudentPoint in public access: - vector<string> students: Declare the variable that contains the student object

- vector<int> points: Declare the variable that contains the point object
- vector<StudentObserver*> observers: declare the variable that contains the Observer object whose pointer is pointing to the StudentObserver class
- void add (StudentObserver * observer): used to add StudentObserver to the list, it as an attachment method
- void notify (); Used to pass the message and pass the data to the StudentObserver class
- void add_student (const string & student, const int & point): used to add students, update data and call the update function of StudentObserver class;
- vector <string> get_students (): used to set values for students.
- vector <int> get_points (): used to set values for points and then, I come to StudentPoint.cpp setting enforce file:

```
void StudentPoint::add(StudentObserver* observer)
{
    observer->set_data(this);
    observers.push_back(observer);
}
void StudentPoint::notify()
{
    for (int i = 0; i < observers.size(); i++)
        observers[i]->update();
}
void StudentPoint::add_student(const string &student, const int &point)
{
    students.push_back(student);
    points.push_back(point);
}
vector<string> StudentPoint::get_students()
{
    return students;
}
vector<int> StudentPoint::get_points()
{
    return points;
}
```

Figure 3 Implement For StudentPoint

The add_student() function (attach method) is used for adding object student into observe list to observe; though the pointer “Observer” to pass new object observer, each student will be pushed as sequence into vector pointer called “StudentObserver”. The second method is notify() (notify method), in this, I set a loop for each of Observer in observer vector will call to update_observer() method of StudentObserver class in order to each of StudentObserver self-update new student. In other words, the notify method will notify to StudentObserver in sub-class to update new student. Besides, the add_student() function(detach method) is used for adding object student and point out of observer list to observe, their inside this function set to push back when there is a change, it will reset the data on the previous class Next, that is the code for initializing Student_Observer.h

```
class StudentObserver
{
    protected:
        StudentPoint* data;
    public:
        virtual void update() = 0; // abstract method
        void set_data(StudentPoint * data);
};
```

Figure 4 Code for initialize StudentObserver

They need to initialize a pointer to the StudentPoint class for studentObserver. In the above, because I want to build "data" pointer to use in StudentObserver class to point to StudentPoint class, so I need to initialize StudentPoint class so that StudentObserver can use StudentPoint's "data" pointer to enforce it on its own. The reason I set all attributes to be protected is to make StudentObserver the subclass like inherit can access these attributes from the based class. I set a set of methods in public access, which I will use to implement StudentObserver.

- virtual void update_weather(): used for updating new student,point from StudentPoin. I set it is a pure virtual method, so the subclass can override them to use for each own actions.
- void set_data(StudentPoint * data);: used for StudentObserver to perform select StudentPoint to observe via the pointer “data”.

```
class StudentChart : public StudentObserver
{
    public:
        void update();
    private:
        void draw_chart(vector<string> student,vector<int> students);
        void draw_bar(const int &value);
};

class StudentTable : public StudentObserver
{
    public:
        void update();
    private:
    void draw_table(vector<string> student,vector<int> students);
};
class Principal : public StudentObserver
{
    public:
        void update(); // override from StudentObserver
    private:
        void draw_table(vector<string> students, vector<int> point);
};

#endif
```

Figure 5 Code for initialize for studentTable,chart and Principal

- When the StudentObserver class is completed, below will have subclasses such as StudentTable, StudentChart, Principal inheriting methods, properties in the parent class StudentObserver through the function void update ();

After that, I come to implementation file of StudentObserver:

```
/// Implement StudentObserver
void StudentObserver::set_data(StudentPoint *data)
{
    this->data = data;
}

/// Implement StudentTable
void StudentTable::update()
{
    cout << "Student Table" << endl;
    vector<string> students = data->get_students();
    vector<int> point = data->get_points();
    draw_table(students, point);
}

void StudentTable::draw_table(vector<string> students, vector<int> point)
{
    printf("+----------+----------+\n");
    printf("|%10s|%10s|\n", "Student", "Point");
    printf("+----------+----------+\n");
    for (int i = 0; i < students.size(); i++)
    {
        printf("|%10s|%10d|\n", students[i].c_str(), point[i]);
    }
    printf("+----------+----------+\n");
}
```

Figure 6 Code for implement StudentObserver

I first implemented the parameterized constructor of studentobserver class to point to studentpoint through the pointer "data" to set data through this-> data = data ;. Next, I implement the StudentTable class, the constructor is parameterized via the update function, with the cout command to print to the screen the data received from StudentObserver. Besides the get_students () and get_points () functions are used by vector "data" to enter student grades and names in the table. To contain all of the above data, a draw_table function used by the vector "StudentObserver" is required. Next is to use

the printf function to create a table and store data such as grades and student names. In the end I implemented the same StudentChart and Principal classes.

```
void StudentChart::update()
{
    cout << "Student Chart" << endl;
    vector<string> students = data->get_students();
    vector<int> point = data->get_points();
    draw_chart(students, point);
}
```

Figure 7 Code for imlement

Also, the pure virtual approach is that update() does not do anything in this class, but this approach will only create an interface to override and enforce sub-class as follows

1. Running program

After running the program of the student system, it will print Student Table, Student Chart and Principal

```
Student Table
+----------+----------+
|   Student|     Point|
+----------+----------+
|      Vinh|         8|
|     Quyet|         5|
|      Tuan|         7|
|     Cuong|         9|
|less than 5 is fail|         4|
+----------+----------+
Student Chart
      Vinh:[]
     Quyet:[]
      Tuan:[]
     Cuong:[]
less than 5 is fail:[]
```

```
Principal
+---------+---------+
|If over 5 points will have a compliment|if less than 5 points will be disapproved|
+---------+---------+
|     Vinh|        8|
|    Quyet|        5|
|     Tuan|        7|
|    Cuong|        9|
|less than 5 is fail|        4|
+---------+---------+
```

Figure 8 system Student interface

After the student system is updated, more data will be added to the table

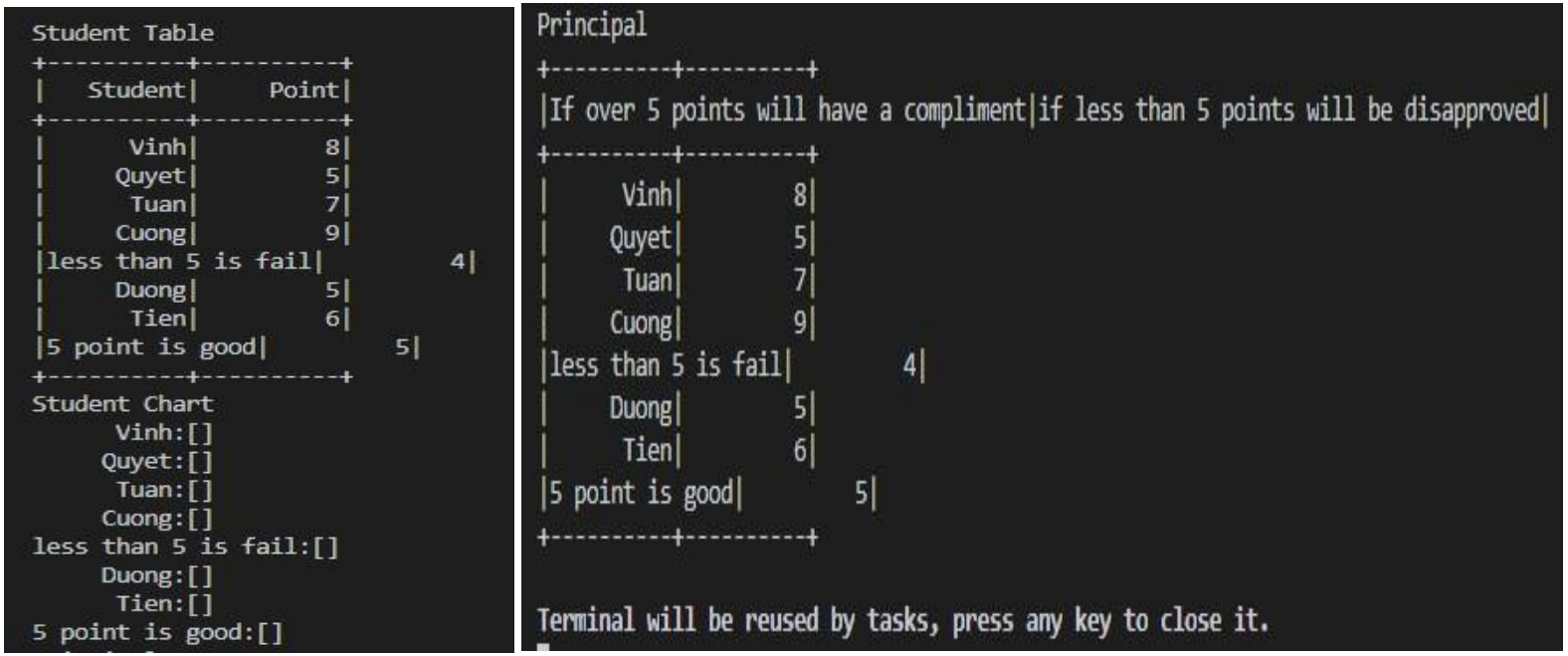


Figure 9 System Student interface after update

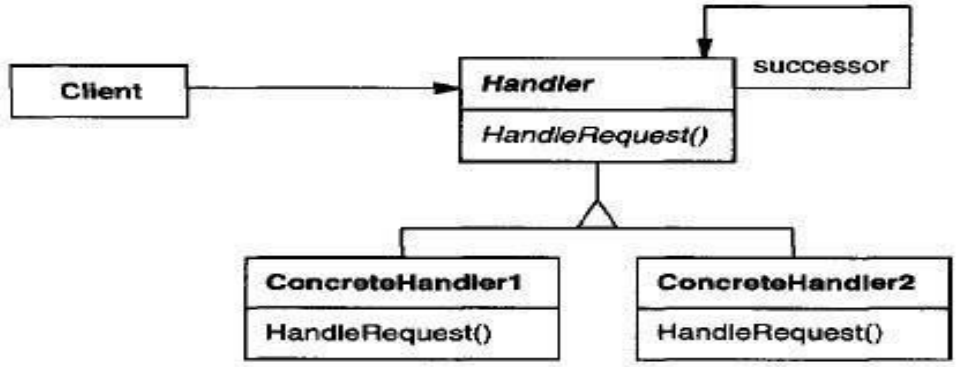
II. Range of Similar Patterns

Within this section, I will explain why the pattern should be chosen for the Observer and become the best choice to use. Next, I learned that using Behavioral Development Model would better manage my situation. Behavioral patterns not only characterize objects but also how they respond to each other. And based on my example, we can see that the focus of this requirement is on interacting between objects, and how they interact. (Gamma, E., Helm, R., Johnson, R. & Vlissides, J., , 1977)

1. Chain of responsibility

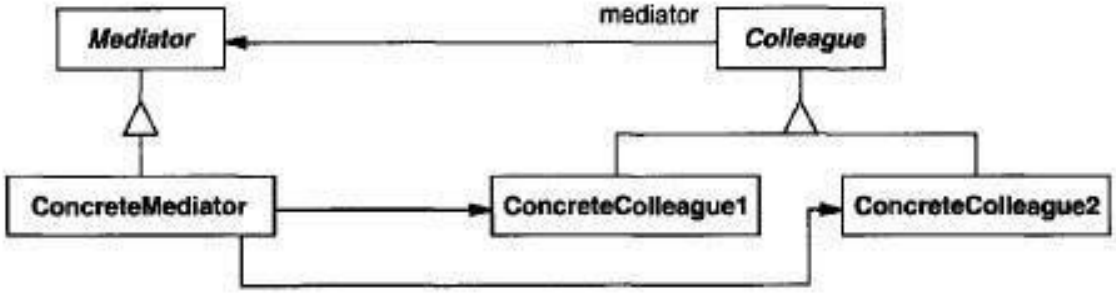
The Responsibility Chain is focused on transforming individual actions into stand-alone objects, called handlers. The pattern shows these handlers are linked into a chain. Inside the chain each linked handler has a field to store a reference to the next handler. (Gamma, et al., 1977). The chain of accountability is a function that has to be done by one of the objects and it is

unknown what object it should manage. An object will be available to test if it is the one that will handle this or not, otherwise it will turn to another object. Importantly, it is likely that the handler cannot transfer requests down. It may present a significant challenge when applied to the scenario since the relationships of the objects in the scenario have to be transparent, and the processor of the system will respond simultaneously with each new state of changing weather instead of reacting in the processing order to find the correct processing.



2. Mediator:

There is a pattern of design which is identical to the pattern of Observer which is the pattern of Mediator. The Mediator's main objective is to make a collection of system components depend only on a single component rather than on shared dependencies. The mediator will have the responsibility to coordinate the interaction between the target groups. The mediator must prevent the parties from making direct mention of one another. Mediator pattern may be used when communicating with a set of objects in well-defined but complex ways. When there is a group of objects that communicate and interact in a complex way, the mediator will resolve the problem. (Gamma, et al., 1977). And looking at our scenario, it is easy to see that it is not too complicated to need a mediator to interact between the objects.



3. Suitable Design Pattern

I chose to use an observer concept framework for the following reasons, after thinking about some behavioral patterns that can be applied to the scenario.

- Observer Design Pattern identifies and maintains the interaction between objects, with other objects receiving notification and automatically changing each time an object is changed.
- After analyzing the scenario, one can see that the monitoring devices will receive a notice to be able to change accordingly for each updated weather change. In addition, the equipment and centers for weather forecasts are one-to-many but not too tight. Additionally, the notification method of changing the new weather to make observe work is certainly the Observer pattern's distributed mechanism. So it can be concluded that the most optimal method is an observer
- In addition, there is also an important point to decide that the Observer pattern will be the best option for this scenario is the scalable and better reuse capabilities, which will have a huge impact on the program if the future system has expanded new function or device class

References

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., . (1977). *Design Patterns: Elements of Reusable Object-Oriented Software*. . anon: anon.