Ministry of education and Higher science of the Russian Federation

Peter The Great St. Petersburg Polytechnic University

—

**Institute of cybersecurity and information security**

**Laboratory work №3**

« **Patterns**»

In the discipline «Object-oriented programming»

Perfomed

student gr. 4851001/80201 S.I. Vetrov

<*signatureь*>

Verified

teaching assistant A.Y. Chernov

<*signature*>

Saint-Petersburg

2020

1. **Formulation of the problem**

Explore different types of patterns and apply the knowledge gained when developing VIM-like application using patterns.

Tasks:

1. Learn MVC paradigm and patterns observer, adapter and etc.
2. Write a text editor that meets the following conditions:
   1. The program must be written in C ++ using STL;
   2. The program must be developed using the MVC;
   3. The program must have TUI - Text User Interface. TUI must support cursor and status bar. The status bar should display current operating mode, the name of the file, line number / total lines;
   4. The program must work with 1-byte text encoding;
   5. The program must support 4 modes: navigation mode, input mode, command mode and search mode;
   6. To work with text, you need to use your own string class in the form of a static or dynamic library;
3. **Progress**

Due to the fact that the tax editor is developed for the WIndows operating system, it is not possible to use the ncurses library. Instead, it will use its analogue - the PDCurses library, written in the C language. besides this STL is used.

* 1. **Patterns**

Before we start designing the above application, let's describe the main patterns that we plan to use in the course of work.

* + 1. **MVC**

MVC stands for model-view-controller. This is a way of organizing code, which involves the allocation of blocks responsible for solving different problems. One block is responsible for application data, another is responsible for the appearance, and the third controls the operation of the application.

* Model - this component is responsible for data and also defines the structure of the application. The model knows nothing about the View or the Controller, which makes it possible to develop and test it as an independent component. And this is the main point of MVC.
* View - This component is responsible for user interaction. That is, the code of the view component defines the appearance of the application and how to use it. There can be several views - they can display the same data in different ways, for example, in the form of a table or graph, or they can be responsible for displaying different parts of the data from the Model.
* Controller - this component is responsible for communication between model and view. The controller component code defines how the application responds to user actions. Basically, it is the brain of an MVC application.

Let's take a look at the graphical representation of an MVC in the form of a diagram, for this refer to Figure 1. In our case, the Observer pattern will be used for interaction between modules. The model will notify the view that changes have been made to the data. View will not be able to access the model directly to take data.

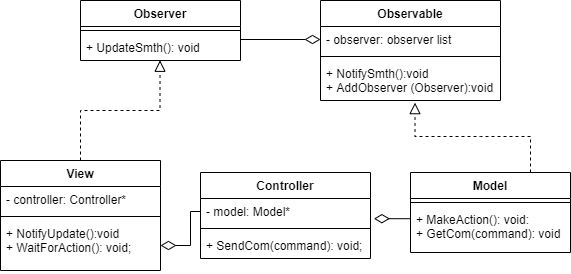


Figure 1- MVC diagram

* + 1. **Adapter**

The Adapter pattern is designed to convert the interface of one class to the interface of another. Thanks to the implementation of this pattern, we can use together classes with incompatible interfaces**.**

We need to use adapter when:

* When you need to use an existing class, but its interface doesn't match your needs;
* When you need to use an existing class in conjunction with other classes whose interfaces are not compatible;

This pattern was used to develop a wrapper class for using the functions of the PDCurses library written in C. Let's look at the structure of this pattern when using it in the implemented program, describing it in the form of a diagram, referring to Figure 2.

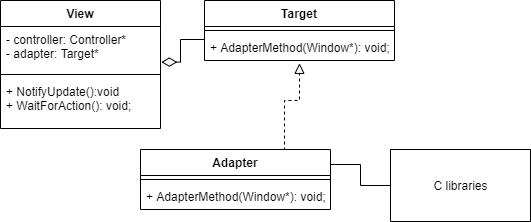


Figure 2 - Adapter diagram

* + 1. **Observer**

Observer is a behavioral design pattern that creates a subscription mechanism that allows one object to watch and respond to events occurring in other objects.

The publisher owns an internal state that is interesting for subscribers to track changes. The publisher contains a subscription mechanism: a list of subscribers and subscription / unsubscribe methods. When a publisher's internal state changes, it notifies its subscribers.

The subscriber defines the interface the publisher uses to send the alert. When the alert arrives, the subscriber needs to get the updated state of the publisher. The publisher can pass this state through the parameters of the notification method. An example of using the pattern when implementing a program is shown in Figure 1.

* 1. **Static library**

The lab directions tell you to use the MyString class implemented in the first lab to store data. To do this, we will compile a static library. The main provision for using a static library is that there is no need to transfer it along with the executable file. it is sewn into it.

Libraries are precompiled for several reasons. First, their code rarely changes. Hence it makes no sense to compile the same functions when compiling the entire program. Second, since all the code is precompiled into machine language, it prevents third parties from accessing and modifying it. To avoid problems with missing dynamic library, version inconsistencies, and so on, we will use a static library.

To create a static library, we need to create a separate project in Visual studio (static library C ++), compile it. Pfntv otzh, it is necessary to add the corresponding files through the project visor to the project with the main program.

* 1. **Creating a VIM-like text editor**

Let's describe step by step the logic of user interaction with the program, as well as the interaction between the modules during the program. First of all, the data is initialized: text fields are created for the console, screen with text, status bar and others. The view and controller objects are created, the corresponding links are created (they will be described in more detail in Figure 3).

View is responsible for interactive user interaction. The user can press any keys, and the view can intercept them. Then the view must pass commands to the model, but it cannot directly do this. To do this, it is rendered to the controller, the pointer to which is stored in one of the view fields. Thus, the view transfers the received command to the controller.

The controller, having received the command, sends the command key to the model. A pointer to the model is stored in one of the contoller fields. In addition, the controller can perform logical processing of the input command, but for convenience, you can bring this functionality into the model, and this was done.

After the model has received the command, it proceeds to its logical processing. Depending on the current operating mode of the program, the same commands may be perceived differently. For example, in text input mode, the resulting "/" will be simply displayed under the cursor. If the program is running in navigation mode, it will switch to the search mode and no data will be displayed in the text field. After the command has been processed, the model sends a view notification (using the Observer mechanism). The notification contains the necessary data for displaying or moving the cursor on the screen.

Upon receiving the notification, the view performs a graphical interpretation of the received data. For example, a view can move the cursor to the left, update the page counter, display a help screen, and much more. For different tasks there are correspondingly different types of notifications with different contents. After receiving a signal from the model and processing it, the view again takes action from the user. Let's summarize the above workflows by building a UML diagram of the application being implemented (see figure 4). The interface of the developed application is shown in Figure 3.



Figure 3 - VIM-like application interface

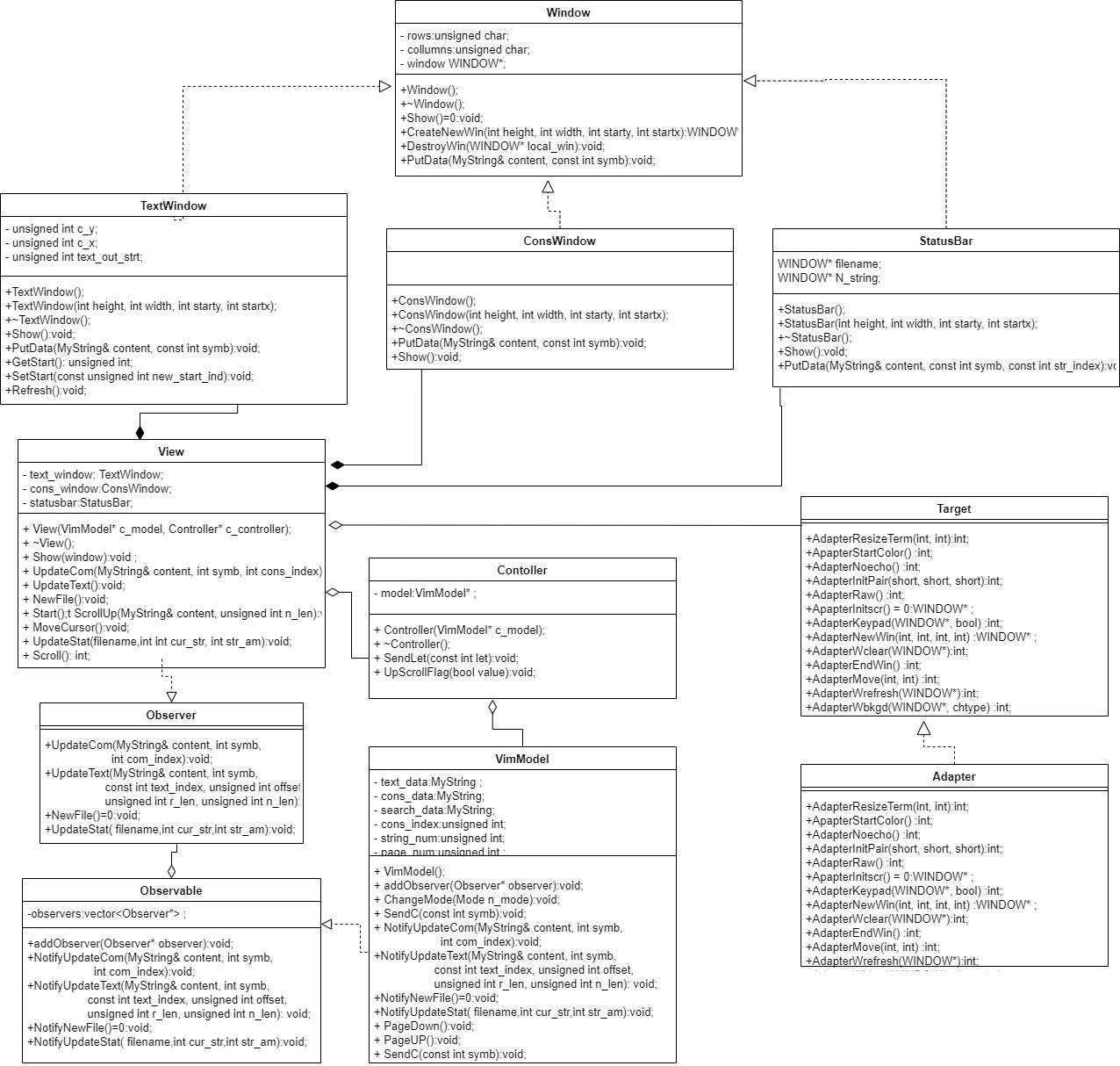


Figure 4- VIM-like application UML

1. **Conclusion**

In the process of doing the work in practice, we studied various patterns and how to apply them during the implementation of applications. For the patterns used, the principle of operation was studied, and the corresponding schematic images were also built. The result of the work is a VIM-like text editor, implemented using MVC.