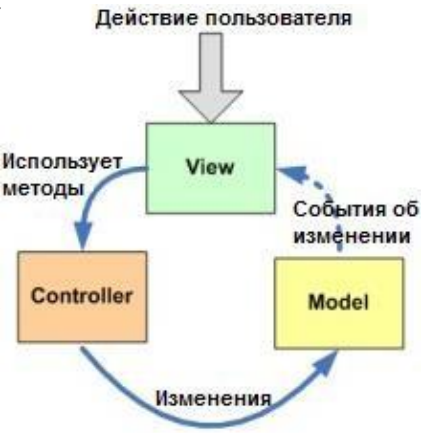
Лабораторная работа № 6
Паттерн MVC: разделение отображения и логики работы с данными

1. Теоретические сведения

Паттерн Model-View-Controller – это методология разделения структуры приложения на специализированные компоненты. Схема MVC предполагает разделение всей системы на 3 взаимосвязанных компонента (подсистемы): так называемую модель (Model), представление (View) и контроллер (Controller). У каждого компонента своя цель, а главная особенность в том, что любой из них можно модифицировать, практически не затронув другие подсистемы. Преимущества такого под хода: модульность, расширяемость, простота поддержки и тестирования.

На диаграмме показана структура паттерна MVC.

Контроллер перехватывает событие извне и в соответствии с заложенной в него логикой реагирует на это событие, изменяя *Модель* посредством вызова соответствующего метода. После изменения *Модель* использует событие о том, что она изменилась, и все подписанные на это события *Представления*, получив его, обращаются к *Модели* за обновленными данными, после чего их и отображают.



Model-View-Controller

2. Пример реализации

Модель

Создадим класс модели задача которого – инкапсулировать всю логику работы с данными (`StudentModel`).

```
class StudentModel {
private:
	string group;
	string name;
public:
	string getGroup() {
		return group;
	}void setGroup(string group) { this->group = group;
	}
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
};
```

Следующая задача – адаптировать этот класс под паттерн Observer.

Паттерн «Наблюдатель» (Observer) определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

Это нужно для того, чтобы остальные классы приложения «знали» о любых изменениях в модели. Для этого создадим еще два класса, которые будут базовыми для остальных – это Observable (определяет методы для добавления, удаления и оповещения «наблюдателей») и Observer (класс, с помощью которого наблюдаемый объект оповещает наблюдателей).

```
class Observer{
public:
	virtual void update() = 0;
};
```

Класс Observevable содержит список всех «наблюдателей». Нового «наблюдателя» можно будет добавить с помощью метода addObserver(). При вызове метода notifyUpdate() класс Observable пройдется по списку «наблюдателей» и вызовет их методы update(), а они, в свою очередь, смогут каким-то образом на это отреагировать.

```
class Observable{
protected:
	vector<Observer*> _observers;
public:
	 void addObserver(Observer *observer){
		_observers.push_back(observer);
	}
	void notifyUpdate(){
		int size = _observers.size();
		for (int i = 0; i < size; i++){
			_observers[i]->update();
		}
	}
};
```

Теперь нужно сделать класс StudentModel «оповещателем», чтобы у него в последствии могли быть «слушатели», следящие за его изменениями. Т.е. сделать класс StudentModel наследником класса Observable.

```
class StudentModel: public Observable {
private:
	string group;
	string name;
public:
	string getGroup() {
		return group;
	}
	void setGroup(string group) {
```

```
            this->group = group;
            notifyUpdate();
        }
        string getName() {
            return name;
        }
        void setName(string name) {
            this->name = name;
            notifyUpdate();
        }
};
```

Обратите внимание, что добавились вызовы `notifyUpdate()` в методах модели. Таким образом достигаем цели: «слушатели» будут оповещены в случае любых изменений в модели.

Представление

Далее нужно создать представление – класс, выводящий изменения модели на консоль. Назовем его `ConsoleView`.

Класс `StudentView` является наследником класса `Observer`, потому он сможет получать сообщения от модели. `StudentView` хранит в себе указатель на модель, который передается в конструкторе. Обратите внимание, что там же `StudentView` «подписывает» себя на изменения модели вызовом метода `addObserver()`. Метод `update()` выводит текущие данные, вызывая метод `print()`.

```
class StudentView: public Observer{
private:
    StudentModel* model;
public:
    StudentView(StudentModel *model) {
        this->model = model;

        this->model->addObserver(this);
    }
    void update() {
        cout << "-=-=(view1 update)=-=-" << endl;
        print();
    }
    void print() {
        cout << "Student" << endl;
        cout << "Name: " << model->getName();
        cout << " Group: " << model->getGroup() << endl << endl;
    }
};
```

Для большей наглядности создадим второе представление `StudentView2`, которое отличается от первого форматом выводимых данных:

```
class StudentView2 : public Observer {
private:
	StudentModel* model;
public:
	StudentView2(StudentModel *model)
	{
		this->model = model;
		this->model->addObserver(this);
	}
	void update() {
		cout << "-=-=(view2 update)=-=-" << endl;
		print();
	}
	void print() {
		cout << "Student" << endl;
		cout << "--- NAME:  " << model->getName() << endl;
		cout << "--- GROUP: " << model->getGroup() << endl << endl;
	}
};
```

Контроллер

Класс контроллера, также как и представление, получает ссылку на модель и представление в конструкторе. Также контроллер содержит методы для изменения модели.

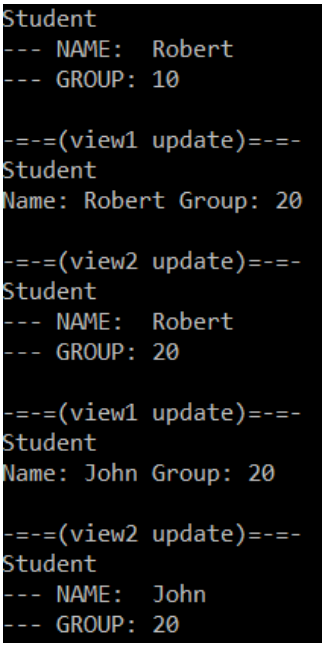
Итоговая реализация:

```
#include "pch.h"
#include <iostream>
#include <string>
#include<vector>
using namespace std;


class Observer{
public:
	virtual void update() = 0;
};

class Observable{
protected:
	vector<Observer*> _observers;
public:
	 void addObserver(Observer *observer){
		 _observers.push_back(observer);
	}
	void notifyUpdate(){
		int size = _observers.size();
		for (int i = 0; i < size; i++){
			_observers[i]->update();
```

```
		}
	}
};

class StudentModel: public Observable {
private:
	string group;
	string name;
public:
	string getGroup() {
		return group;
	}
	void setGroup(string group) {
		this->group = group;
		notifyUpdate();
	}
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
		notifyUpdate();
	}
};

class StudentView: public Observer{
private:
	StudentModel* model;
public:
	StudentView(StudentModel *model) {
		this->model = model;
		this->model->addObserver(this);
	}
	void update() {
		cout << "-=-=(view1 update)=-=-" << endl;

		print();
	}
	void print() {
		cout << "Student" << endl;
		cout << "Name: " << model->getName();
		cout << " Group: " << model->getGroup() << endl << endl;
	}
};

class StudentView2 : public Observer {
private:
	StudentModel* model;
public:
	StudentView2(StudentModel *model)
	{
		this->model = model;
		this->model->addObserver(this);
```

```
	}
	void update() {
		cout << "-=-=(view2 update)=-=-" << endl;
		print();
	}
	void print() {
		cout << "Student" << endl;
		cout << "--- NAME:  " << model->getName() << endl;
		cout << "--- GROUP: " << model->getGroup() << endl << endl;
	}
};

class StudentController {
private:
	StudentModel* model = new StudentModel();
	StudentView* view = new StudentView(model);

public:
	StudentController(StudentModel* model, StudentView* view) {
		this->model = model;
		this->view = view;
	}
	void setStudentName(string name) {
		model->setName(name);
	}
	void setStudentGroup(string group) {
		model->setGroup(group);
	}
};

int main()
{
	StudentModel* student = new StudentModel();
	student->setName("Robert");
	student->setGroup("10");

	StudentView* view = new StudentView(student);
	StudentView2* view2 = new StudentView2(student);

	view->print();
	view2->print();

	StudentController controller(student, view);
	controller.setStudentGroup("20");
// приведет к автоматическому обновлению модели
// и изменению представлений
	controller.setStudentName("John");
// приведет к автоматическому обновлению модели
// и изменению представлений
}
```

Результат:



3. Задание к лабораторной работе

На базе любой из ранее выполненных лабораторных работ разработать приложение (C++, C#) (применение Windows-форм позволит заработать дополнительные баллы) с применением паттерна MVC для разделения сложной модели (основной класс) и её представления.

Разработать UML-диаграмму классов и диаграмму последовательности.

Отчет по лабораторной работе – файл формата pdf. Формат имени файла отчета: <НомерГруппы>_<ФамилияСтудента>.pdf. В отчет включить построенные диаграммы и исходный код программы (при необходимости и заголовочные файлы). При использовании Windows-форм вместо исходного кода в отчет вставить ссылку на репозиторий GitHub с проектом. Формат отчета см. в Приложении. Отчет загрузить в LMS.

При защите лабораторной работы: уметь объяснить логику и детали работы программы; реализацию паттернов проектирования на примере разработанной программы.