МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное

учреждение высшего образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

**Институт компьютерных технологий и информационной безопасности**

**Кафедра Математического обеспечения и применения ЭВМ**

**ОТЧЁТ**

по лабораторной работе №5

по курсу «GoF-паттерны ООП»

на тему «Паттерн Наблюдатель»

Выполнила:

студентка группы КТмо2-3

Куприянова А.А.

Проверил:

доцент кафедры ТОР

Максимов М.Н.

Оценка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2017 г.

Таганрог 2017

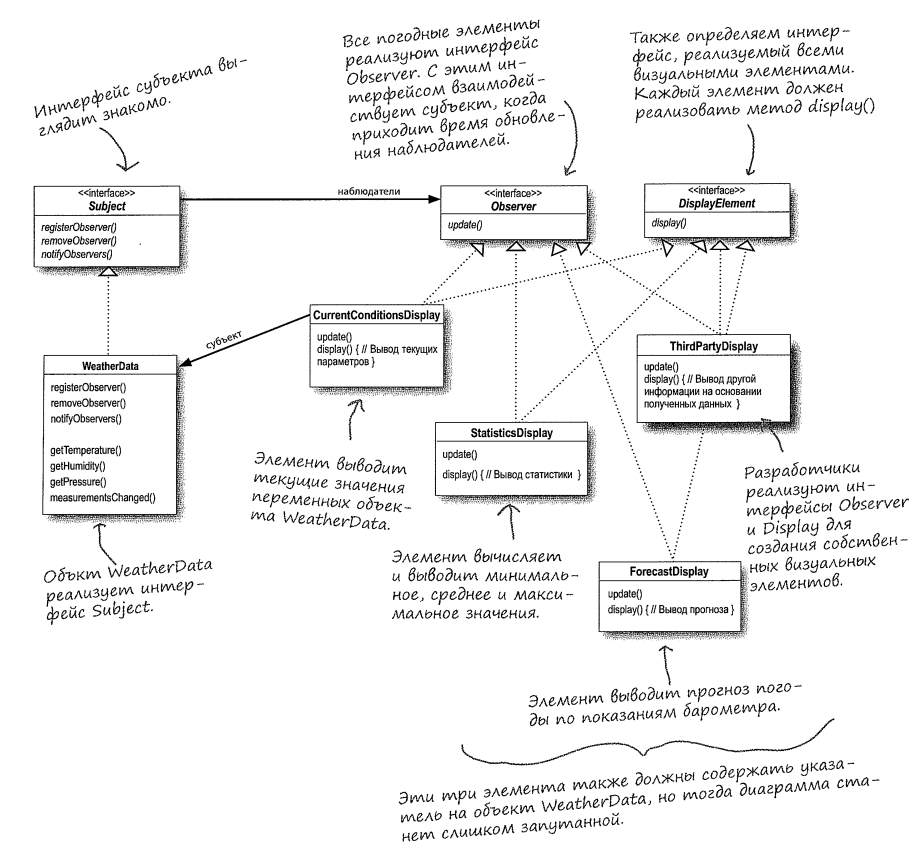
**Цель работы:** Получение практических навыков работы с паттерном программирования «Наблюдатель».

**Задача**.

Объект WeatherData умеет получать от физической метеостанции обновленные данные. Затем объект WeatherData обновляет изображение для трех основных элементов: текущего состояния (температура, влажность и давление), статистики и прогноза.

Задача — создать приложение, которое использует данные объекта WeatherData для обновления текущих условий, статистики и прогноза погоды.

Общая схема решения:



Выполнение работы.

Для того, чтобы метеостанция обрабатывала также солнечную активность, а наблюдатели её получали и выводили на экран, был введён тип новый перечисляемый тип данных (тип датчика), который кроме прочего содержит данные о солнечной активности:

enum sensorType {

temperatureSensor,

humiditySensor,

pressureSensor,

sunSensor

};

Данные метеостанции характеризуются двумя значениями: тип датчика и его значением. В связи с этим обновление и вывод на экран данных теперь происходит в зависимости от типа данных:

void display() { // вывод на экран данных

cout << "\n Current conditions: ";

switch (sensor) {

case temperatureSensor: {

cout << "temperature: ";

break;

}

case humiditySensor: {

cout << "humidity: ";

break;

}

case pressureSensor: {

cout << "pressure: ";

break;

}

case sunSensor: {

cout << "sun activity: ";

break;

}

default:

break;

}

cout << value << "\n";

Когда метеостанция получает новые данные, информация о полученных данных передаётся в зависимости от того, какие данные пришли: передаётся тип данных как параметр в функцию measurementsChange:

// возможно задавать не все значения датчиков

void setMeasurements(float t = NULL, float h = NULL, float p = NULL, float s = NULL) {

if (t) {

temperature = t;

measurementsChange(temperatureSensor);

}

if (h) {

humidity = h;

measurementsChange(humiditySensor);

}

if (p) {

pressure = p;

measurementsChange(pressureSensor);

}

if (s) {

sunActivity = s;

measurementsChange(sunSensor);

}

}

**Полный текст программы.**

// Observer.cpp: определяет точку входа для консольного приложения.

//

#include "stdafx.h"

#include <iostream>

#include <list>

#include <algorithm>

#include <vector>

using namespace std;

// тип информации с датчика, значение датчика задаётся отдельно

enum sensorType {

temperatureSensor,

humiditySensor,

pressureSensor,

sunSensor

};

/\* Класс, обеспечивающий семантику подсчёта ссылок.

\* Объект, на который ссылается указатель, автоматически

\* уничтожается при удалении последнего экземпляра CountedPtr

\* для данного объекта.

\*/

template <class T>

class CountedPtr {

private:

T\* ptr; // Указатель на значение

long\* count; // Количество владельцев (общие данные)

public:

// Инициализация объекта существующим указателем

// - указатель p должен быть получен в результате вызова new

explicit CountedPtr(T\* p = 0) : ptr(p), count(new long(1)) {}

// Копирующий указатель (увеличивает счётчик владельцев)

CountedPtr(const CountedPtr<T>& p) throw() : ptr(p.ptr), count(p.count) {

++\*count;

}

// Деструктор (уничтожает объект, если владелец был последним)

~CountedPtr() throw() {

dispose();

}

// Присваивание (перевод указателя на новый объект)

CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {

if (this != &p) {

dispose();

ptr = p.ptr;

count = p.count;

++\*count;

}

return \*this;

}

// Доступ к объекту, на который ссылается указатель

T& operator\*() const throw() {

return \*ptr;

}

T\* operator->() const throw() {

return ptr;

}

private:

void dispose() {

if (--\*count == 0) {

delete count;

delete ptr;

}

}

};

class WeatherData;

class Observer;

class Subject {

public:

virtual void registerObserver(Observer\* ob) = 0;

virtual void removeObserver(Observer\* ob) = 0;

virtual void notifyObserver(sensorType st) = 0;

};

class Observer {

public:

virtual void update(sensorType st, float v) = 0;

};

class DisplayElement {

public:

virtual void display() = 0;

};

class CurrentConditionsDisplay : public Observer, DisplayElement {

private:

// данные о погоде характеризуются двумя параметрами: тип данных и значение датчика

float value;

sensorType sensor;

std::vector<Subject\*> weatherData;

public:

CurrentConditionsDisplay(Subject\* wD, sensorType st) : sensor(st) {

weatherData.push\_back(wD);

for (auto s : weatherData)

s->registerObserver(this);

}

void update(sensorType st, float val) {

if (st != sensor) return;

value = val;

display();

}

void display() {

cout << "\n Current conditions: ";

switch (sensor) {

case temperatureSensor: {

cout << "temperature: ";

break;

}

case humiditySensor: {

cout << "humidity: ";

break;

}

case pressureSensor: {

cout << "pressure: ";

break;

}

case sunSensor: {

cout << "sun activity: ";

break;

}

default:

break;

}

cout << value << "\n";

}

void removeObserver() {

for (auto s : weatherData)

s->removeObserver(this);

}

~CurrentConditionsDisplay() {

for (auto s : weatherData)

s->removeObserver(this);

}

// добавление данных

void addNewSubject(Subject\* wD) {

weatherData.push\_back(wD);

weatherData.back()->registerObserver(this);

}

};

typedef CountedPtr<Observer> ObserverPtr;

class WeatherData : public Subject {

private:

list<ObserverPtr> list\_Obs;

float temperature;

float humidity;

float pressure;

float sunActivity;

public:

void registerObserver(Observer\* ob) { list\_Obs.push\_front(ObserverPtr(ob)); }

void removeObserver(Observer\* ob) {

list<ObserverPtr>::iterator pos;

for (pos = list\_Obs.begin(); pos != list\_Obs.end(); ++pos)

if (&(\*(\*pos)) == ob) {

list\_Obs.erase(pos);

break;

}

}

void notifyObserver(sensorType st) {

list<ObserverPtr>::const\_iterator pos;

for (pos = list\_Obs.begin(); pos != list\_Obs.end(); ++pos) {

switch (st) {

case temperatureSensor: {

(\*pos)->update(st, temperature);

break;

}

case humiditySensor: {

(\*pos)->update(st, humidity);

break;

}

case pressureSensor: {

(\*pos)->update(st, pressure);

break;

}

case sunSensor: {

(\*pos)->update(st, sunActivity);

break;

}

default:

break;

}

}

}

void measurementsChange(sensorType st) { notifyObserver(st); }

// возможно задавать не все значения датчиков

void setMeasurements(float t = NULL, float h = NULL, float p = NULL, float s = NULL) {

if (t) {

temperature = t;

measurementsChange(temperatureSensor);

}

if (h) {

humidity = h;

measurementsChange(humiditySensor);

}

if (p) {

pressure = p;

measurementsChange(pressureSensor);

}

if (s) {

sunActivity = s;

measurementsChange(sunSensor);

}

}

~WeatherData() { list\_Obs.clear(); }

};

int main()

{

WeatherData \* pWD = new WeatherData;

WeatherData \* pWD2 = new WeatherData;

CurrentConditionsDisplay \*SDT = new CurrentConditionsDisplay(pWD, temperatureSensor);

CurrentConditionsDisplay \*SDH = new CurrentConditionsDisplay(pWD, humiditySensor);

CurrentConditionsDisplay \*SDP = new CurrentConditionsDisplay(pWD, pressureSensor);

CurrentConditionsDisplay \*SDSA = new CurrentConditionsDisplay(pWD, sunSensor);

SDT->addNewSubject(pWD2);

cout << "pWD:\n";

pWD->setMeasurements(10, 20, 30, 40);

cout << "pWD2:\n";

pWD2->setMeasurements(-20);

cout << "pWD:\n";

pWD->setMeasurements(40, 10);

cout << "pWD2:\n";

pWD2->setMeasurements(-30);

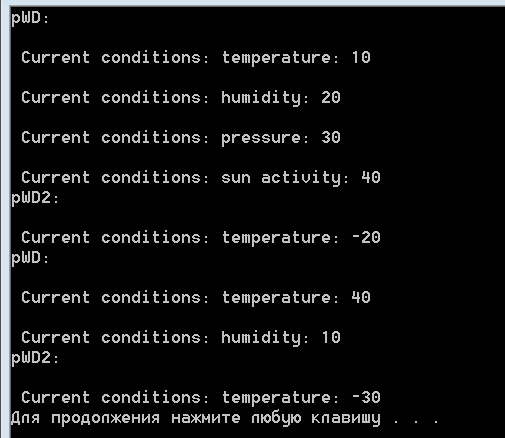
system("pause");

delete pWD, pWD2;

return 0;

}

**Результат работы программы.**



При обновлении данных они выводятся на экран.