**Министерство образования и науки Российской Федерации**

**Пермское федеральное государственное бюджетное образовательное учреждение высшего образования «Пермский Национальный Исследовательский Политехнический Университет»**

**Электротехнический факультет**

**Кафедра «Информационные технологии и автоматизированные системы»**

**ОТЧЁТ**

По лабораторной работе №18.6

Вариант №11

Выполнил студент группы РИС-20-2б

Савельев Виталий Антонович

Проверил доцент кафедры ИТАС

Полякова О.А.

Пермь 2021

**Цель работы** –научиться работать с одной из моделей работы программы – событийно-ориентированной моделью, в которой действия пользователя запускают определённые части программы, определённые интерфейсом.

**Постановка задачи**

Задача – реализовать на языке программирования C++ программу, демонстрирующую работу событийно-ориентированной модели программы, консольного интерфейса.

**Исходные данные для варианта №11:**

* Класс-родитель: «Человек»;
* Класс-ребёнок: «Студент»;
* Класс-обработчик событий;
* Класс-контейнер: «Дерево» - двумерный вектор классов «Человек»;

**Анализ задачи**

Для решения задачи были использованы следующие средства:

1. Язык программирования C++ (Microsoft Visual C++)
2. Текстовый редактор Microsoft Visual Studio Code
3. Система контроля версий Git

Класс-обработчик событий отвечает за обработку действий пользователя, в данном случае будет обрабатываться строка, которая по сути своей является некоторой командой, которую хочет выполнить пользователь в программе.

Объявление класса-обработчика событий в заголовочном файле:

#include "Tree.h"

#include "TEvent.h"

class ConsoleDialog : public Tree

{

protected:

int \_endState;

public:

ConsoleDialog(void);

virtual ~ConsoleDialog(void);

virtual void GetEvent(TEvent& event);

virtual int Execute();

virtual void HandleEvent (TEvent& event);

virtual void ClearEvent(TEvent& event);

int ValidValue();

void EndExecuting();

};

Реализация обработчика событий очень проста, на вход подаётся команда, которая перебирается по определённому шаблону, привязке команды и её имени через перечисление.

Запускается некоторая диалоговая область, пользователь в неё вводит данные, данные обрабатываются и в зависимости от результатов обработки, запускается необходимая команда или выводится сообщение об отсутствии некоторой команды.

#include "ConsoleDialog.h"

#include <iostream>

using namespace std;

ConsoleDialog::ConsoleDialog(void) : Tree()

{

\_endState = 0;

}

ConsoleDialog::~ConsoleDialog() {}

void ConsoleDialog::GetEvent(TEvent &event)

{

string charData = "+-?/qam";

cout << "Command>>";

string command;

cin >> command;

char commandName = command[0];

if (charData.find(commandName) >= 0)

{

event.Type = EventMessage;

switch (commandName)

{

case 'm':

event.Command = CommandMake;

break;

case '+':

event.Command = CommandAdd;

break;

case '-':

event.Command = CommandDelete;

break;

case '?':

event.Command = CommandShow;

break;

case 'q':

event.Command = CommandQuit;

break;

}

if (command.length() > 1)

{

string parameter = command.substr(1, command.length() - 1);

event.a = atoi(parameter.c\_str());

}

}

else

event.Type = EventNothing;

}

int ConsoleDialog::Execute()

{

TEvent event;

do {

\_endState = 0;

GetEvent(event);

HandleEvent(event);

}

while (!ValidValue());

return \_endState;

}

int ConsoleDialog::ValidValue()

{

if (\_endState == 0)

return 0;

else

return 1;

}

void ConsoleDialog::ClearEvent(TEvent &event)

{

event.Type = EventNothing;

}

void ConsoleDialog::EndExecuting()

{

\_endState = 1;

}

void ConsoleDialog::HandleEvent(TEvent &event)

{

if (event.Type == EventMessage)

{

switch (event.Command)

{

case CommandQuit:

EndExecuting();

ClearEvent(event);

break;

case CommandMake:

MakeTree(event.a);

ClearEvent(event);

break;

case CommandDelete:

DeleteLastPerson(0);

ClearEvent(event);

break;

case CommandShow:

PrintAllData();

ClearEvent(event);

break;

case CommandAdd:

string name;

int age;

cin >> name >> age;

Person newPerson(name, age);

AddNewPerson(newPerson, 0);

ClearEvent(event);

break;

}

}

}

Класс-контейнер «Дерево» представляет собой на деле не дерево, а двумерный массив с изменяющейся длиной. На эту роль отлично подходит вектор векторов, содержащих в свою очередь сами элементы.

#include <vector>

#include "Student.h"

class Tree

{

private:

vector<vector<Person>> \_data;

protected:

void MakeTree(int count);

public:

int Count() { return \_data.size(); }

Tree();

Tree(int count);

void AddNewGroup(vector<Person> personGroup) { \_data.push\_back(personGroup); }

void AddNewPerson(Person person, int groupIndex) { \_data[groupIndex].push\_back(person); }

void DeleteLastGroup();

void DeleteLastPerson(int groupIndex);

void PrintAllData();

};

Класс «Человек» объявлен следующим образом:

* Имя человека;
* Возраст человека;

Класс «Студент» является наследником от класса «Человек» и дополнительно содержит:

* Рейтинг студента;

Человек:

#include <string>

using namespace std;

class Person {

protected:

int \_age;

std::string \_name;

public:

int Age() { return \_age; }

std::string Name() { return \_name; }

void SetAge(int age);

void SetName(string name);

virtual void PrintData();

Person();

Person(string name, int age);

};

Студент:

#include "Person.h"

#include <string>

using namespace std;

class Student : public Person {

private:

int \_rating;

public:

int Rating() { return \_rating; }

void SetRating(int rating);

void PrintData();

Student();

Student(string name, int age, int rating);

};

Поскольку вся логика с меню и событиями была вынесена в отдельный класс, то в основной программе остаётся лишь запустить ту самую диалоговую событийно-ориентированную часть программы, возможности которой необходимо продемонстрировать.

#include "ConsoleDialog.h"

using namespace std;

int main()

{

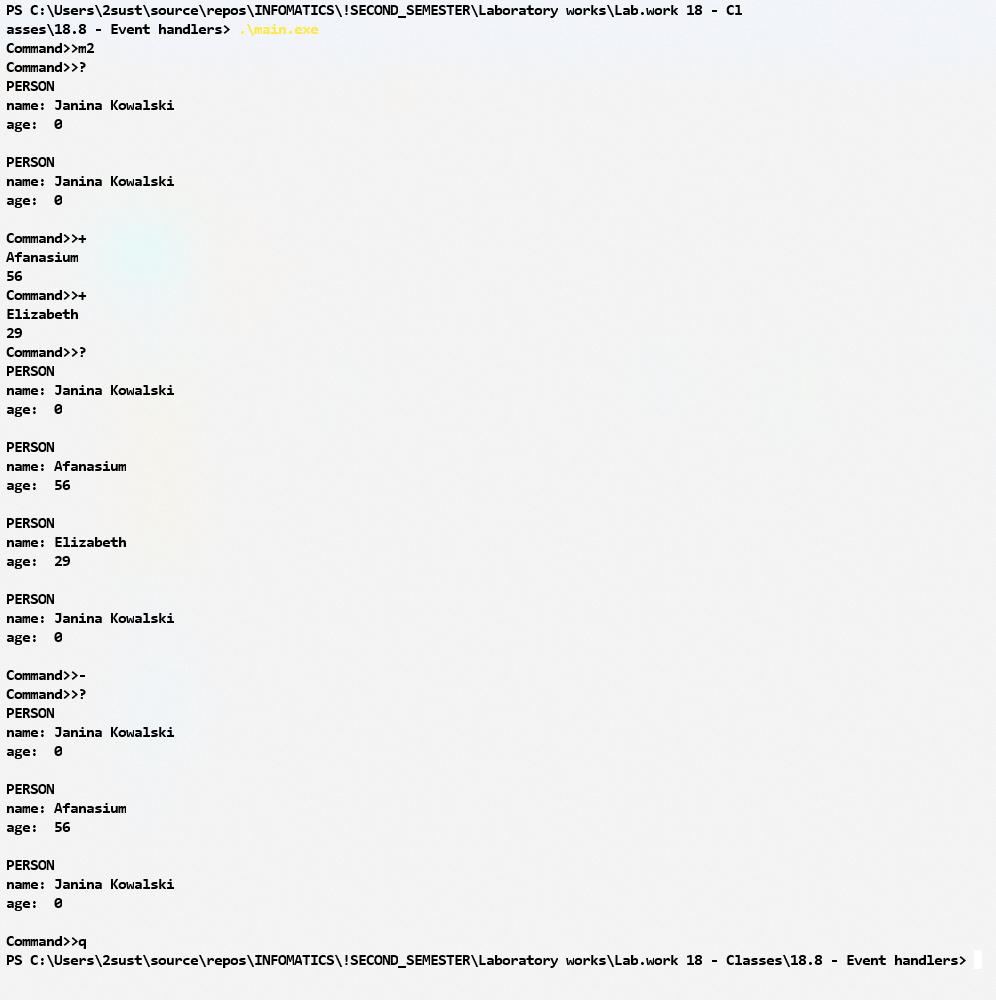
ConsoleDialog dialog;

dialog.Execute();

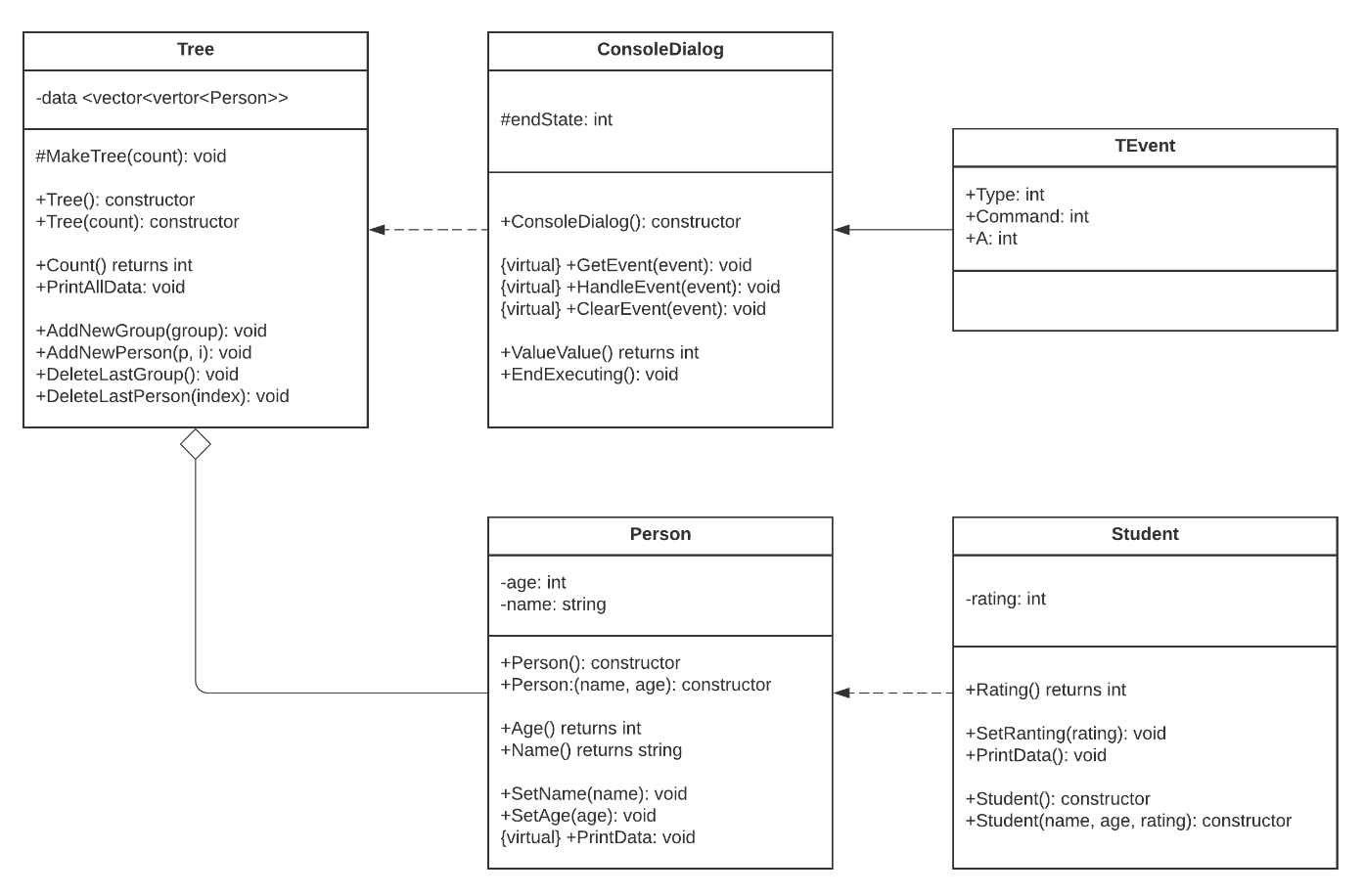
return 0;

}

**Скриншот выполненной программы:**



**UML-диаграмма программы:**



**Ответы на вопросы:**

**Что такое класс-группа? Привести примеры таких классов.**

это объект, в который включены другие объекты. Объекты, входящие в группу, называются элементами группы. Элементы группы, в свою очередь, могут быть группой. Привер: ФАКУЛЬТЕТ, КАФЕДРА, СТУДЕНЧЕСКАЯ ГРУППА.

**Какой вид иерархии дает группа?**

Агрегация.

**Почему во главе иерархии классов, содержащихся в группе объектов должен находиться абстрактный класс?**

Для того, чтобы была возможность обращаться к методам классов по указателю.

**Что такое событие? Для чего используются события?**

Пакеты информации, которыми обмениваются объекты и которые создаются объектно-ориентированной средой в ответ на те или иные действия пользователя.

**Какие характеристики должно иметь событие-сообщение?**

код класса сообщения, отличающий сообщения объектов одного класса от объектов другого класса;

адрес объекта, которому предназначено сообщение (м. б. не задан, тогда сообщение могут прочитать все объекты);

**Задана структура события**

**struct TEvent**

**{**

**int what;**

**union**

**{**

**MouseEventType mouse;**

**KeyDownEvent keyDown;**

**MessageEvent message;**

**}**

**};**

**Какие значения, и в каких случаях присваиваются полю what?**

mouse – Движение мышью

keyDown – Нажатие кнопки вниз

message – Введено сообщение

**Задана структура события**

**struct TEvent**

**{**

**int what;//тип события union**

**{**

**int command;//код комманды**

**struct//параметры команды**

**{**

**int message;**

**int a;**

**};**

**};**

**};**

**Какие значения, и в каких случаях присваиваются полю command?**

Command – введена команда

Message – введено сообщение

**Задана структура события**

**struct TEvent**

**{**

**int what;//тип события union**

**{**

**int command;//код комманды struct//параметры команды**

**{**

**int message; int a;**

**};**

**};**

**};**

**Для чего используются поля a и message?**

Для передачи дополнительной информации

**Какие методы необходимы для организации обработки сообщений?**

GetEvent – формирование события;

Execute реализует главный цикл обработки событий. Он постоянно получает событие путем вызова GetEvent и обрабатывает их с помощью HandleEvent. Этот цикл завершается, когда поступит событие «конец».

HandleEvent – обработчик событий. Обрабатывает каждое событие нужным для него образом. Если объект должен обрабатывать определенное событие (сообщение), то его метод HandleEvent должен распознавать это событие и реагировать на него должным образом. Событие может распознаваться, например, по коду команды (поле command).

ClearEvent очищает событие, когда оно обработано, чтобы оно не обрабатывалось далее.

Обработчик событий (метод HandleEvent).

**Какой вид имеет главный цикл обработки событий-сообщений?**

Главный цикл обработки событий реализуется в методе Execute главной группы- объекта “прикладная программа” по следующей схеме:

int TMyApp::Execute()

{

do

{

endState=0;

GetEvent(event);

HandleEvent(event);

if(event.what!=evNothing) EventError(event);

}

while(!Valid());

return endState;

}

**Какую функцию выполняет метод ClearEvent()? Каким образом?**

ClearEvent очищает событие, присваивая полю event.What значение evNothing.

**Какую функцию выполняет метод HandleEvent ()? Каким образом?**

Метод HandleEvent программы обрабатывает событие “конец работы”, вызывая метод EndExec. EndExec изменяет значение private – переменной EndState. Значение этой переменной проверяет метод–функция Valid, возвращающая значение true, если “конец работы”.

**Какую функцию выполняет метод GetEvent ()?**

Получает событие и передает его в Execute()

**Для чего используется поле EndState?**

Значение переменной EndState отвечает за окончание выполнения основного цикла.

**Какой класс (объект) содержит это поле?**

Класс Dialog (Основной класс обработки событий)

**Для чего используется функция Valid()?**

Проверяет значение EndState, возвращающает значение true, если “конец работы”.

**Полный исходный кода программы на языке программирования C++:**

ConsoleDialog.h

#include "Tree.h"

#include "TEvent.h"

class ConsoleDialog : public Tree

{

protected:

int \_endState;

public:

ConsoleDialog(void);

virtual ~ConsoleDialog(void);

virtual void GetEvent(TEvent& event);

virtual int Execute();

virtual void HandleEvent (TEvent& event);

virtual void ClearEvent(TEvent& event);

int ValidValue();

void EndExecuting();

};

ConsoleDialog.cpp

#include "ConsoleDialog.h"

#include <iostream>

using namespace std;

ConsoleDialog::ConsoleDialog(void) : Tree()

{

\_endState = 0;

}

ConsoleDialog::~ConsoleDialog() {}

void ConsoleDialog::GetEvent(TEvent &event)

{

string charData = "+-?/qam";

cout << "Command>>";

string command;

cin >> command;

char commandName = command[0];

if (charData.find(commandName) >= 0)

{

event.Type = EventMessage;

switch (commandName)

{

case 'm':

event.Command = CommandMake;

break;

case '+':

event.Command = CommandAdd;

break;

case '-':

event.Command = CommandDelete;

break;

case '?':

event.Command = CommandShow;

break;

case 'q':

event.Command = CommandQuit;

break;

}

if (command.length() > 1)

{

string parameter = command.substr(1, command.length() - 1);

event.a = atoi(parameter.c\_str());

}

}

else

event.Type = EventNothing;

}

int ConsoleDialog::Execute()

{

TEvent event;

do {

\_endState = 0;

GetEvent(event);

HandleEvent(event);

}

while (!ValidValue());

return \_endState;

}

int ConsoleDialog::ValidValue()

{

if (\_endState == 0)

return 0;

else

return 1;

}

void ConsoleDialog::ClearEvent(TEvent &event)

{

event.Type = EventNothing;

}

void ConsoleDialog::EndExecuting()

{

\_endState = 1;

}

void ConsoleDialog::HandleEvent(TEvent &event)

{

if (event.Type == EventMessage)

{

switch (event.Command)

{

case CommandQuit:

EndExecuting();

ClearEvent(event);

break;

case CommandMake:

MakeTree(event.a);

ClearEvent(event);

break;

case CommandDelete:

DeleteLastPerson(0);

ClearEvent(event);

break;

case CommandShow:

PrintAllData();

ClearEvent(event);

break;

case CommandAdd:

string name;

int age;

cin >> name >> age;

Person newPerson(name, age);

AddNewPerson(newPerson, 0);

ClearEvent(event);

break;

}

}

}

Person.h

#include <string>

using namespace std;

class Person {

protected:

int \_age;

std::string \_name;

public:

int Age() { return \_age; }

std::string Name() { return \_name; }

void SetAge(int age);

void SetName(string name);

virtual void PrintData();

Person();

Person(string name, int age);

};

Person.cpp

#include "Person.h"

#include <iostream>

#include <string>

void Person::SetAge(int age)

{

if (age < 0)

std::cout << "Error: invaid age;\n";

else

\_age = age;

}

void Person::SetName(std::string name)

{

if (name.length() < 2)

std::cout << "Error: invalid name;\n";

else

\_name = name;

}

Person::Person()

{

\_name = "Janina Kowalski";

\_age = 0;

}

Person::Person(string name, int age)

{

SetName(name);

SetAge(age);

}

void Person::PrintData()

{

cout << "PERSON\n";

cout << "name: " << \_name << endl;

cout << "age: " << \_age << "\n\n";

}

Student.h

#include "Person.h"

#include <string>

using namespace std;

class Student : public Person {

private:

int \_rating;

public:

int Rating() { return \_rating; }

void SetRating(int rating);

void PrintData();

Student();

Student(string name, int age, int rating);

};

Student.cpp

#include "Student.h"

#include <iostream>

void Student::SetRating(int rating)

{

if (rating <= 0)

std::cout << "Error: invalid rating;\n";

else

\_rating = rating;

}

Student::Student()

{

\_name = "Janina Kowalski";

\_age = 0;

\_rating = 0;

}

Student::Student(string name, int age, int rating)

{

SetName(name);

SetAge(age);

SetRating(rating);

}

void Student::PrintData()

{

cout << "STUDENT\n";

cout << "name: " << \_name << endl;

cout << "age: " << \_age << endl;

cout << "Rating: " << \_rating << "\n\n";

}

TEvent.h

#pragma once

const int EventNothing = 0;

const int EventMessage = 100;

const int CommandAdd = 1;

const int CommandDelete = 2;

const int CommandShow = 4;

const int CommandMake = 6;

const int CommandQuit = 101;

struct TEvent

{

int Type;

int Command;

int a;

};

Tree.h

#include <vector>

#include "Student.h"

class Tree

{

private:

vector<vector<Person>> \_data;

protected:

void MakeTree(int count);

public:

int Count() { return \_data.size(); }

Tree();

Tree(int count);

void AddNewGroup(vector<Person> personGroup) { \_data.push\_back(personGroup); }

void AddNewPerson(Person person, int groupIndex) { \_data[groupIndex].push\_back(person); }

void DeleteLastGroup();

void DeleteLastPerson(int groupIndex);

void PrintAllData();

};

Tree.cpp

#include "Tree.h"

Tree::Tree()

{

\_data.clear();

}

Tree::Tree(int count)

{

MakeTree(count);

}

void Tree::PrintAllData()

{

vector<Person> currentVector;

for (int i = 0; i < \_data.size(); i++)

{

currentVector = \_data[i];

for (int j = 0; j < currentVector.size(); j++)

\_data[i][j].PrintData();

}

}

void Tree::DeleteLastGroup()

{

\_data.pop\_back();

}

void Tree::DeleteLastPerson(int groupIndex)

{

\_data[groupIndex].pop\_back();

}

void Tree::MakeTree(int count)

{

Person filler;

vector<Person> fillVector;

fillVector.push\_back(filler);

for (int i = 0; i < count; i++)

\_data.push\_back(fillVector);

}

main.cpp

#include "ConsoleDialog.h"

using namespace std;

int main()

{

ConsoleDialog dialog;

dialog.Execute();

return 0;

}