**Лабораторная работа №6**

**Исследование способов применения поведенческих паттернов**

**проектирования при рефакторинге ПО**

**Цель работы**

Исследовать возможность использования поведенческих паттернов проектирования. Получить практические навыки применения паттернов поведения при объектно-ориентированном проектировании и рефакторинге ПО.

**Постановка задачи**

1. Изучить назначение и структуру паттерна Цепочка обязанностей

(выполнить в ходе самостоятельной подготовки).

2. Применительно к программному продукту, выбранному для

рефакторинга, проанализировать возможность использования паттерна Цепочка обязанностей. Для этого построить диаграмму классов, на диаграмме классов найти класс-клиент, запрос от которого необходимо передавать по цепочке объектов, и классы-получатели запросов, объекты которых целесообразно объединять в цепочку.

3. Выполнить перепроектирование системы, использовав паттерн Цепочка обязанностей, изменения отобразить на диаграмме классов.

4. Сравнить полученные диаграммы классов, сделать выводы и целесообразности использования паттернов проектирования для данной системы.

5. На основе полученной UML-диаграммы модифицировать программный код, скомпилировать программу.

**Ход работы**

В качестве поведенческого паттерна для использования был выбран паттерн наблюдатель.

Имелась программа, в которой работникам могла выдаваться зарплата. Было решено, что информация о выдаче зарплаты может потребоваться в дальнейшем, например объекту, отвечающему за ведение бухгалтерского учета. Для этого в классе сотрудника были добавлены методы, которые позволяют другим объектам подписаться на обновление состояния их зарплаты или отписаться. В качестве наблюдателя был разработан класс SalaryObserver, который выводит информацию об начислении зарплаты сотрудника.

Получившийся код приведен ниже:

#include <iostream>

#include <vector>

#include <string>

#include <list>

#include <map>

#include "algorithm"

using namespace std;

**class Observer**

**{**

**public:**

**virtual ~Observer() {}**

**virtual void update(int salary, const string &workerName) = 0;**

**};**

class Worker

{

public:

Worker(const string &name) : name\_(name) {}

**void attach(Observer \*observer)**

**{**

**observers\_.push\_back(observer);**

**}**

**void detach(Observer \*observer)**

**{**

**observers\_.remove(observer);**

**}**

void paySalary(int salary)

{

salaryHistory\_.push\_back(salary);

cout << "Worker " << name\_ << " received salary: " << salary << endl;

**for (auto &observer : observers\_)**

**{**

**observer->update(salary, name\_);**

**}**

}

void showSalaryHistory()

{

cout << "Salary history for worker " << name\_ << ":" << endl;

for (const auto &salary : salaryHistory\_)

{

cout << salary << endl;

}

}

const string &getName() const

{

return name\_;

}

private:

string name\_;

list<int> salaryHistory\_;

**list<Observer \*> observers\_;**

};

**class SalaryObserver : public Observer**

**{**

**public:**

**void update(int salary, const string &workerName) override**

**{**

**cout << "SalaryObserver: Worker " << workerName << " received salary: " << salary << endl;**

**}**

**};**

class WorkerManager

{

public:

vector<Worker> workers\_;

void addWorker(const string &name)

{

workers\_.emplace\_back(name);

}

void removeWorker(const string &name)

{

workers\_.erase(remove\_if(workers\_.begin(), workers\_.end(),

[&name](const Worker &worker)

{ return worker.getName() == name; }),

workers\_.end());

cout << "Worker: " << name << " removed" << '\n';

}

void paySalary(const string &name, int salary)

{

for (auto &worker : workers\_)

{

if (worker.getName() == name)

{

worker.paySalary(salary);

break;

}

}

}

void showAllWorkers()

{

for (const auto &worker : workers\_)

{

cout << "Worker: " << worker.getName() << endl;

}

}

};

int main()

{

WorkerManager workerManager;

workerManager.addWorker("Ivan");

workerManager.addWorker("Petr");

workerManager.addWorker("Sidor");

SalaryObserver salaryObserver;

**for (auto &worker : workerManager.workers\_)**

**{**

**worker.attach(&salaryObserver);**

**}**

workerManager.paySalary("Ivan", 50000);

workerManager.paySalary("Petr", 60000);

workerManager.paySalary("Sidor", 70000);

workerManager.paySalary("Petr", 15000);

workerManager.showAllWorkers();

workerManager.removeWorker("Ivan");

workerManager.showAllWorkers();

workerManager.workers\_[0].showSalaryHistory();

return 0;

}

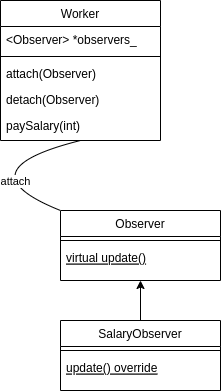


Рисунок 1 – Диаграмма классов

**ВЫВОДЫ**

В ходе выполнения лабораторной работы были исследованы возможности использования поведенческих паттернов проектирования. Получены практические навыки применения паттернов поведения при объектно-ориентированном проектировании и рефакторинге ПО.

Был применен поведенческий паттерн «Наблюдатель», позволяющим объектам подписываться на обновление состояния некоторого другого объекта. Такой подход позволяет реагировать на изменение состояния объекта и выполнять необходимый функционал.