МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА № 42

ОТЧЁТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

старший преподаватель		С.Ю. Гуков
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЁТ О ЛАБОРАТОРНОЙ РАБОТЕ №2

Паттерны проектирования

по курсу:

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. №	4329		Д.С. Шаповалова
		подпись, дата	инициалы, фамилия

Санкт-Петербург 2025

ОГЛАВЛЕНИЕ

Цель работы

Ознакомиться с основными паттернами проектирования и понять их назначение для решения общих задач проектирования в конкретном контексте, научиться применять паттерны на практике при проектировании и разработке программного обеспечения.

Постановка задачи

Необходимо придумать контекст и разработать программу, используя предложенные в вариантах два паттерна проектирования. Реализация должна быть сделана в одном связанном проекте (контексте). Также требуется нарисовать UML диаграмму классов реализуемой программы. Каждый студент берет два варианта заданий: один – свой вариант по списку, второй – любой на выбор с паттерном другой группы. То есть, например, один из паттернов должен быть структурным или порождающим, а второй должен быть поведенческим – это обеспечит баланс между архитектурой и поведением. Не разрешается реализовывать в каждом конкретном паттерне такой же контекст, как был в примерах на лекциях. Проект может быть выполнен либо в качестве консольного приложения (тогда обязателен командно-текстовый интерфейс), либо иметь графический пользовательский интерфейс (User Interface, UI), а также может быть написан на любом языке программирования.

Требования к структуре проекта

- ☐ Применение принципов ООП (наследования, инкапсуляции, полиморфизма, абстракции) и SOLID
- ☐ Дружелюбный командно-текстовый, либо графический пользовательский интерфейс
- ☐ UML диаграмма классов или диаграмма последовательностей

Краткое описание хода разработки и назначение используемых технологий

Ход разработки:

1. Разработана логика работы класса кот.
2. Разработана логика классов жителей квартиры.
3. Разработан графический интерфейс - Windows-form.

Назначение используемых технологий

Используются технологии: паттерны проектирования Observer и Singleton.

Observer - наблюдатель - при изменении в экземпляре класса издателя, подписанные экземпляры классы получают уведомление.

Singleton - одиночка - закрывает классический конструктор, позволяя создавать только один экземпляр класса через метод.

А также ранее изученные технологии:

ООП: наследование, инкапсуляция (изоляция лишних данных от пользователя; разделение логики и поведения), полиморфизм (один метод умеет работать с разными данными), абстракция (представление объекта без логики).

SOLID:

S - на 1 класс 1 задача.

O - класс открыт для расширения, но не изменения.

L - вместо родителя можно (нужно) использовать наследника, программа не упадёт.

I - класс реализует только нужный ему interface.

D - верхние модули не зависят от нижних; у обоих зависимость только от абстракций, не наоборот; абстракции не зависят от деталей, детали зависят от абстракций.

Описание использованных паттернов и аргументация их выбора.

В реализованной программе - симуляторе кота, есть, собственно, кот - в единственном экземпляре => использование паттерна Singleton. Также есть кнопка “Попытка заменить кота”, которая является частью сюжета и демонстрирует применение паттерна - кот создаётся только один, даже если создать кота вновь, это будет тот же кот.

Основные функции программы реализованы с использованием паттерна Observer, так как есть необходимость оповещать объекты классов “жителей квартиры” о изменении состояния кота.

Исходный код программы (с комментариями в необходимых местах)

Cat.cs:

```
using System;

namespace Симулятор_Кота
{
   // Интерфейс подписчика
   public interface IObserver
   {
      void Update(string message);
   }

   // Интерфейс издателя
   public interface ISubject
   {
      void Subscribe(IObserver observer);
      void Unsubscribe(IObserver observer);
      void Notify(string message);
   }
   // абстракция + наследование
   public abstract class Питомец
   {
      public abstract void Notify(string message);
      public abstract void ОповеститьОГолоде();

   }
   // Кот — издатель
   public class Кот : Питомец, ISubject
   {
      private List<IObserver> _observers = new List<IObserver>();

      private static Кот _instance;
      private Кот() { }
      public static Кот GetInstance()
      {
         if (_instance == null)
            _instance = new Кот();

         return _instance;
      }

      public void Subscribe(IObserver observer)
      {
         _observers.Add(observer);
      }

      public void Unsubscribe(IObserver observer)
      {
         _observers.Remove(observer);
      }
```

```
    public override void Notify(string message)
    {
        foreach (var observer in _observers)
        {
            observer.Update(message);
        }
    }

    public override void ОповеститьОГолоде()
    {
        MessageBox.Show("Кот: Я голоден!");
        Notify("Кот голоден. Пора покормить.");
    }

    public void ОповеститьОЗакрытойДвери()
    {
        MessageBox.Show("Кот: Дверь закрыта!");
        Notify("Кот жалуется, что дверь закрыта.");
    }
}


// Жительница квартиры — подписчик
public class ЖительницаКвартиры : IObserver
{
    private string _имя;

    public ЖительницаКвартиры(string имя)
    {
        _имя = имя;
    }

    public void Update(string message)
    {
        MessageBox.Show($"{_имя} получила уведомление: {message}");
    }
}

// Жилец квартиры — подписчик
public class ЖилецКвартиры : IObserver
{
    private string _имя;

    public ЖилецКвартиры(string имя)
    {
        _имя = имя;
    }

    public void Update(string message)
    {
        MessageBox.Show($"{_имя} получил уведомление: {message}");
```

```
        }
    }
}
```

Form1.cs:

```
using System.Windows.Forms;

namespace Симулятор_Кота
{
    public partial class Form1 : Form
    {
        Кот кот = Кот.GetInstance();

        ЖительницаКвартиры мама = new ЖительницаКвартиры("Анна");
        ЖилецКвартиры папа = new ЖилецКвартиры("Борис");

        public Form1()
        {

            InitializeComponent();

        }
        private void Form1_Load(object sender, EventArgs e)
        {
        }

        // Функции кота
        private void btn_food_Click(object sender, EventArgs e)
        {
            кот.ОповеститьОГолоде();
        }

        private void btn_door_Click(object sender, EventArgs e)
        {
            кот.ОповеститьОЗакрытойДвери();
        }

        // МАМА
        private void btn_subscribe_mom_Click(object sender, EventArgs e)
        {
            кот.Subscribe(мама);
        }

        private void btn_Unsubscribe_mom_Click(object sender, EventArgs e)
        {
            кот.Unsubscribe(мама);
        }

        // ПАПА
        private void btn_subscribe_dad_Click(object sender, EventArgs e)
        {
            кот.Subscribe(папа);
```

```
        }

        private void btn_Unsubscribe_dad_Click(object sender, EventArgs e)
        {
            кот.Unsubscribe(папа);
        }

        private void btn_singleton_Click(object sender, EventArgs e)
        {
            Кот новыйКот = Кот.GetInstance();
            if (кот == новыйКот)
            {
                MessageBox.Show("Мы выгнали старого кота и взяли нового, но это оказался тот же
кот. Нам от него не избавиться...");
            }
            else
            {
                MessageBox.Show("Получилось, у нас новый кот!");
            }
        }
    }
}
```

Результаты работы программы с примерами разных сценариев (скриншоты)

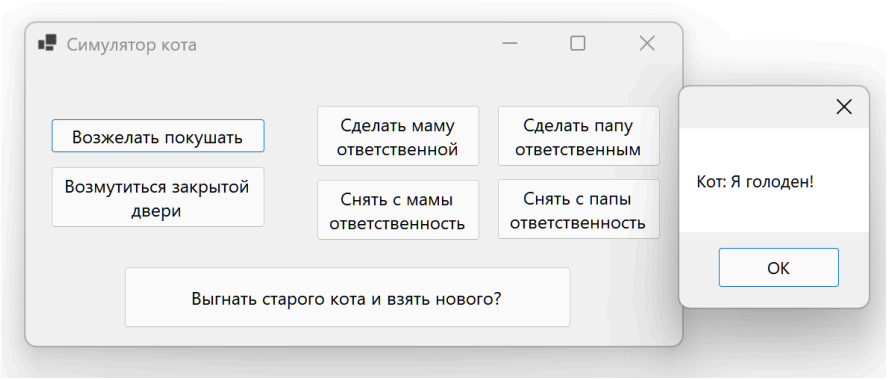


Рисунок 1.1 - Кот проголодался

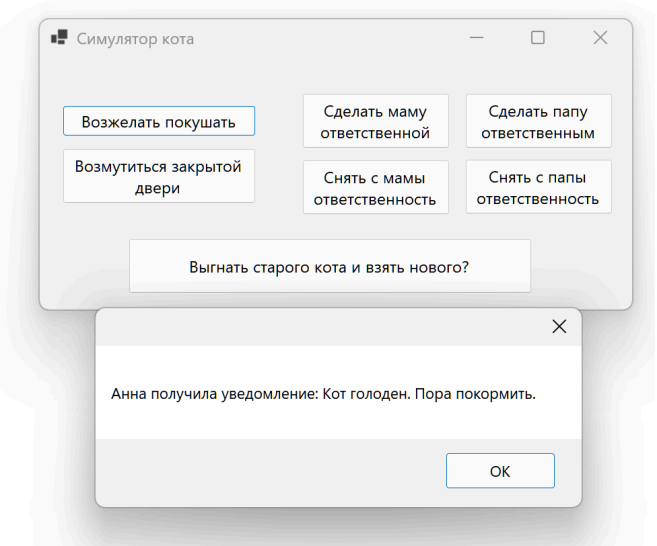


Рисунок 1.2 - Жительница получила уведомление

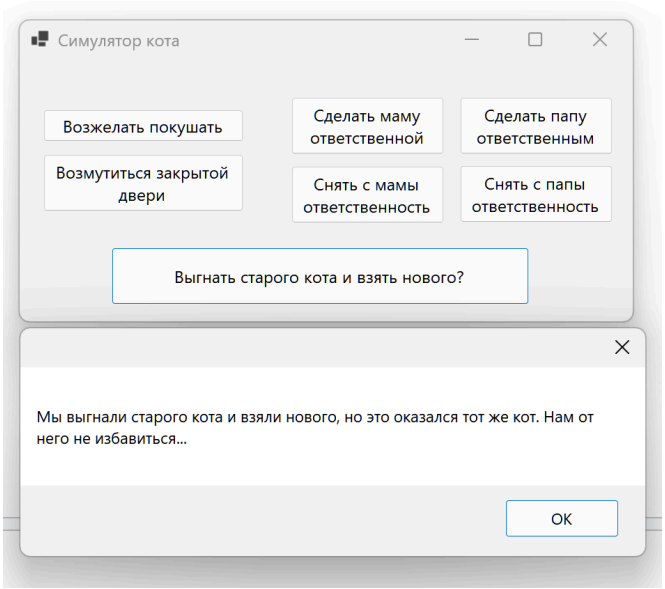
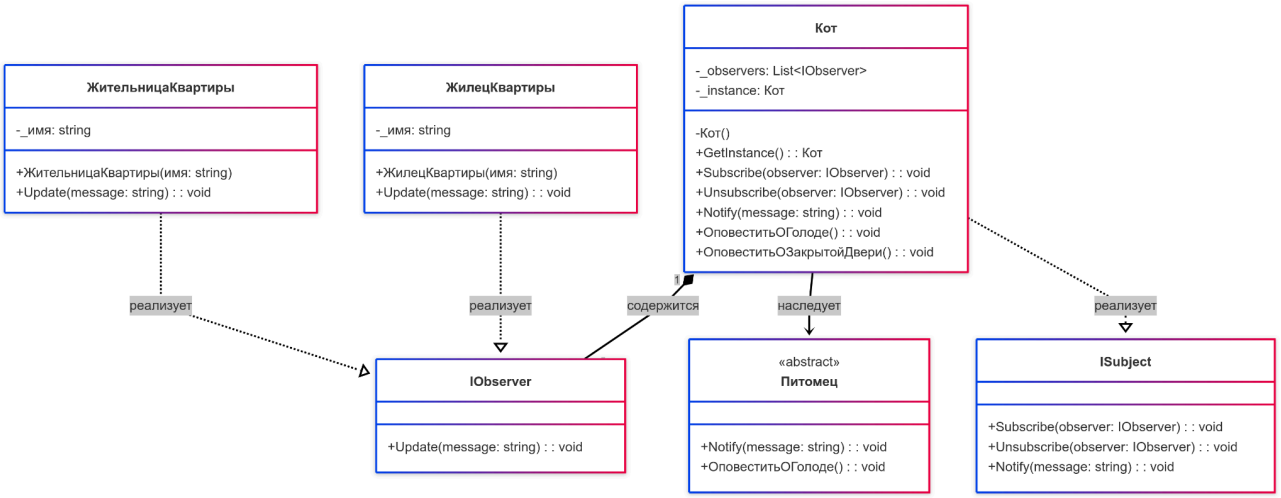


Рисунок 1.3 - Кот остался тем же

UML диаграмма классов:



Выводы

В ходе выполнения лабораторной работы мной были освоены и изучены: принципы ООП и SOLID; понятие дружелюбный интерфейс; использование паттерна проектирования Observer и Singleton. Написанная программа была протестирована, полученный результат соответствует ожиданиям.