Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра Информатики
Дисциплина «Программирование»

ОТЧЕТ

к лабораторной работе №9

на тему:

«МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ.»

БГУИР 6-05-0612-02 17

Выполнил студент группы 353502
ХАРИТОНЧИК Денис Сергеевич

_____

(дата, подпись студента)

Проверил ассистент каф.
Информатики
РОМАНЮК Максим Валерьевич

_____

(дата, подпись преподавателя)

Минск 2024

1 ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ

Задание 1. Вариант 7. Описать семейство классов, имеющих общий функционал (), при этом в каждом классе присутствует дополнительно свой функционал. Набор дополнительных функций в разных классах может быть произвольным. Дополнительный функционал описать в виде набора интерфейсов. Одна из общих функций должна быть реализована по-своему в каждом классе. Одна из общих функций должна быть реализована в других классах (например, изменение скорости, использование оружия, доставка груза). При этом должно быть несколько вариантов реализации (несколько классов), например, персонажам игры доступны разные инструменты – каждый инструмент может использоваться разными персонажами. Конкретный вариант реализации выбирается при создании объекта (применить шаблон проектирования «Мост» («Bridge»)). Для создания объектов использовать шаблон проектирования «Абстрактная фабрика» (Abstract factory) или «Построитель» (Builder). В классе Program создать коллекцию разных объектов (см. п.1). Затем для каждого элемента коллекции вызвать все методы, доступные для данного объекта. Предметная область: Медперсонал. Пример общих свойств: имя, профиль (хирург, терапевт, медсестра …). Общие функции: GetInfo (в каждом классе реализуется по своему), Diagnose (варианты описаны в других классах, примеры: осмотр, МРТ, рентген…). Примеры дополнительных функций: сделать укол, выписать больничный, сделать перевязку …).

2 ВЫПОЛНЕНИЕ РАБОТЫ

В рамках проекта была создана директория MedicalStaff, в которой разместились 6 файлов с классами Doctor, Nurse, MedicalStaffBase, MedicalStaffFactory, MRIDiagnoseMethod, XRayDiagnoseMethod и 4 интерфейса IDiagnosable, IDiagnoseMethod, IMovable, IWorker. Все эти классы и интерфейсы тесно взаимосвязаны и представляют основные компоненты для системы мониторинга медицинским персоналом.

Класс Doctor представляет медицинского работника с профилем врача. Он имеет имя и профиль, методы для перемещения Walk и Run, выполнения работы DoWork, и диагностики Diagnose. Когда вызывается метод GetInfo, он выводит информацию о враче, включая его имя и профиль. Ниже приведен листинг кода класса Doctor.

```
internal class Doctor : MedicalStaffBase, IMovable, IWorker, IDiagnosable
{
    private readonly IDiagnoseMethod _diagnoseMethod;

      public Doctor(string name, IDiagnoseMethod diagnoseMethod) : base(name,
"Doctor")
    {
        _diagnoseMethod = diagnoseMethod;
    }

    public void Walk()
    {
        Console.WriteLine($"{Name} is walking");
    }
    public void Run()
    {
        Console.WriteLine($"{Name} is running");
    }

    public void DoWork()
    {
        Say("I am examining patients");
    }

    public void Diagnose()
    {
        _diagnoseMethod.Diagnose();
    }
    public override void GetInfo()
    {
        Say($"Doctor {Name}, profile: {Profile}");
    }
}
```

Класс Nurse представляет медицинского работника с профилем медсестры. Он также имеет имя и профиль, аналогично врачу. Методы для перемещения Walk и Run, выполнения работы DoWork, и диагностики Diagnose также присутствуют. При вызове метода GetInfo выводится информация о медсестре, включая ее имя и профиль. Ниже приведен листинг кода класса Nurse.

```
internal class Nurse : MedicalStaffBase, IMovable, IWorker, IDiagnosable
{
    public readonly IDiagnoseMethod _diagnoseMethod;
      public Nurse(string name, IDiagnoseMethod diagnoseMethod) : base(name,
"Nurse")
    {
        _diagnoseMethod = diagnoseMethod;
    }
    public void Walk()
    {
```

```
        Console.WriteLine($"{Name} is walking");
    }
    public void Run()
    {
        Console.WriteLine($"{Name} is running");
    }
    public void DoWork()
    {
        Say("I'm assisting the doctor");
    }
    public void Diagnose()
    {
        _diagnoseMethod.Diagnose();
    }
    public override void GetInfo()
    {
        Say($"Nurse {Name}, profile: {Profile}");
    }
}
```

Класс MedicalStaffBase является абстрактным базовым классом для всех медицинских работников. Он содержит общие свойства и методы, которые наследуются классами Doctor и Nurse. Класс имеет свойства Name и Profile, которые представляют имя и профиль медицинского персонала соответственно. Класс также содержит абстрактный метод GetInfo, который должен быть реализован в производных классах для вывода информации о медицинском персонале. Кроме того, класс имеет метод Say, который выводит сообщение с именем медицинского персонала. Ниже приведен листинг кода класса MedicalStaffBase.

```
internal abstract class MedicalStaffBase
{
    protected string Name;
    protected string Profile;
    protected MedicalStaffBase(string name, string profile)
    {
        Name = name;
        Profile = profile;
    }

    // вывод информации о персонале
    public abstract void GetInfo();

    // вывод сообщения с именем медперсонала
    public void Say(string message)
    {
        Console.WriteLine($"{Name} says: {message}");
    }
}
```

Класс MedicalStaffFactory является фабрикой для создания объектов медицинского персонала. Он предоставляет методы для создания объектов классов Doctor и Nurse. Фабрика принимает имя медицинского работника и профиль в качестве параметров при создании объектов. Это позволяет создавать различных медицинских работников с разными профилями. Класс MedicalStaffFactory также отвечает за создание объектов различных методов диагностики, которые используются медицинским персоналом для выполнения диагностических процедур. Это обеспечивает гибкость выбора метода диагностики при создании медицинского персонала. Ниже приведен листинг кода класса MedicalStaffFactory.

```
internal class MedicalStaffFactory
{
    public Nurse CreateNurse(string name, IDiagnoseMethod diagnoseMethod)
    {
        return new Nurse(name, diagnoseMethod);
    }
    public Doctor CreateDoctor(string name, IDiagnoseMethod diagnoseMethod)
    {
        return new Doctor(name, diagnoseMethod);
    }
}
```

Класс MRIDiagnoseMethod представляет метод диагностики с использованием МРТ (магнитно-резонансная томография). Он реализует интерфейс IDiagnoseMethod, который определяет метод Diagnose, используемый медицинским персоналом для выполнения диагностических процедур. Класс MRIDiagnoseMethod содержит логику для выполнения диагностики с помощью МРТ. Этот метод может быть использован медицинским персоналом, таким как врачи и медсестры, для получения более подробной информации о состоянии пациента. Создание объекта MRIDiagnoseMethod позволяет использовать этот метод диагностики при создании медицинского персонала, что обеспечивает гибкость выбора метода диагностики в зависимости от потребностей клиники или условий пациента. Ниже приведен листинг кода класса MRIDiagnoseMethod.

```
internal class MRIDiagnoseMethod : IDiagnoseMethod
{
    public void Diagnose()
    {
        Console.WriteLine("Performing MRI examination");
    }
}
```

Класс XRayDiagnoseMethod представляет метод диагностики с использованием рентгеновских лучей. Он также реализует интерфейс IDiagnoseMethod, предоставляющий метод Diagnose для выполнения диагностических процедур. Класс XRayDiagnoseMethod содержит логику для проведения диагностики с использованием рентгеновского оборудования. Этот метод может быть применен медицинским персоналом для получения изображений внутренних структур пациента, что помогает в определении возможных заболеваний или повреждений. Создание объекта XRayDiagnoseMethod позволяет использовать этот метод диагностики при создании медицинского персонала, что дает возможность выбора метода диагностики в соответствии с потребностями конкретной ситуации. Ниже приведен листинг кода класса XRayDiagnoseMethod.

```
internal class XRayDiagnoseMethod : IDiagnoseMethod
{
    public void Diagnose()
    {
        Console.WriteLine("Performing X-ray examination");
    }
}
```

Интерфейс IDiagnosable определяет контракт для классов, которые могут выполнять диагностику. Он содержит единственный метод Diagnose, который должен быть реализован в классах, использующих этот интерфейс. Метод Diagnose предназначен для выполнения диагностических процедур, таких как осмотр, проведение МРТ, рентгенография и другие. Классы, реализующие интерфейс IDiagnosable, должны обеспечивать реализацию этого метода в соответствии с конкретными потребностями и целями диагностики в контексте приложения. Использование интерфейса IDiagnosable позволяет объединить различные методы диагностики под общим интерфейсом, что способствует гибкости и расширяемости кода. Ниже приведен листинг кода интерфейса IDiagnosable.

```
internal interface IDiagnosable
{
    void Diagnose();
}
```

Интерфейс IDiagnoseMethod определяет контракт для классов, представляющих методы диагностики в медицинском приложении. Он включает в себя единственный метод Diagnose, который должен быть реализован в классах, использующих этот интерфейс. Метод Diagnose

предназначен для выполнения конкретной диагностической процедуры, такой как МРТ, рентгенография. Классы, реализующие интерфейс IDiagnoseMethod, предоставляют логику для выполнения соответствующих диагностических процедур. Это позволяет разделить абстракцию метода диагностики от его конкретной реализации, что обеспечивает гибкость и расширяемость приложения. Использование интерфейса IDiagnoseMethod также позволяет легко добавлять новые методы диагностики в будущем, не затрагивая существующий код. Ниже приведен листинг интерфейса IDiagnoseMethod.

```
internal interface IDiagnoseMethod
{
    void Diagnose();
}
```

Интерфейс IMovable определяет контракт для классов, которые могут перемещаться. Он содержит методы Walk и Run, предназначенные для описания движения объекта. Классы, реализующие интерфейс IMovable, должны обеспечивать реализацию этих методов в соответствии с конкретными потребностями и особенностями перемещения в контексте приложения. Использование интерфейса IMovable позволяет абстрагировать поведение перемещения от конкретных реализаций объектов, что способствует гибкости и расширяемости кода. Таким образом, этот интерфейс полезен для создания модульных и легко масштабируемых приложений, где перемещение объектов является важной частью функциональности. Ниже приведен листинг интерфейса IMovable.

```
internal interface IMovable
{
    void Walk();
    void Run();
}
```

Интерфейс IWorker определяет контракт для классов, которые могут выполнять работу. Он включает в себя метод DoWork, предназначенный для выполнения определенной работы или действия. Классы, реализующие интерфейс IWorker, должны обеспечивать реализацию этого метода в соответствии с конкретными требованиями и задачами, которые должны быть выполнены в контексте приложения. Использование интерфейса IWorker позволяет абстрагировать работу от конкретных реализаций объектов, что способствует гибкости и расширяемости кода. Таким образом, этот интерфейс полезен для создания модульных и легко масштабируемых

приложений, где выполнение работы играет важную роль в функциональности. Ниже приведен листинг кода интерфейса IWorker.

```
internal interface IWorker
{
    void DoWork();
}
```

Класс Program представляет точку входа в ваше приложение. Он содержит метод Main, который является точкой запуска приложения. В методе Main происходит инициализация и запуск основной логики программы. В вашем случае, в методе Main создается экземпляр фабрики MedicalStaffFactory, затем создается коллекция медицинского персонала с помощью этой фабрики, и для каждого элемента коллекции вызываются различные методы, такие как GetInfo, DoWork и Diagnose, чтобы продемонстрировать функционал каждого объекта. Кроме того, в методе Main используются созданные объекты методов диагностики MRIDiagnoseMethod и XRayDiagnoseMethod, которые передаются при создании медицинского персонала с помощью фабрики. Все это позволяет показать работу вашей программы и взаимодействие между различными компонентами. Ниже приведен листинг кода класса Program. На рисунке 1 изображена работа консольного приложения. На рисунке 2 изображена UML-диаграмма приложения.

```
public class Program
{
    static void Main(string[] args)
    {
        // создание фабрики
        MedicalStaffFactory factory = new MedicalStaffFactory();
        // создание объектов диагностики
        IDiagnoseMethod mriDiagnoseMethod = new MRIDiagnoseMethod();
        IDiagnoseMethod xRayDiagnoseMethod = new XRayDiagnoseMethod();
        // Создание коллекции медперсонала
        List<MedicalStaffBase> medicalStaff = new List<MedicalStaffBase>
        {
            factory.CreateNurse("Alice", mriDiagnoseMethod),
            factory.CreateDoctor("Bob", xRayDiagnoseMethod)
        };
        // вызов методов для каждого элемента коллекции
        foreach (var staff in medicalStaff)
        {
            staff.GetInfo();

            // проверяем тип объекта и вызываем соответствующие методы
            if (staff is Doctor)
```

```
            {
                var doctor = (Doctor)staff;
                doctor.Walk();
                doctor.Run();
                doctor.DoWork();
                doctor.Diagnose();
            }
            else if (staff is Nurse)
            {
                var nurse = (Nurse)staff;
                nurse.Walk();
                nurse.Run();
                nurse.DoWork();
                nurse.Diagnose();
            }
            Console.WriteLine();
        }
    }
}
```

```
/Users/denisharitonciksergeevic/RiderProjects/Lab_9/Lab_9/bin/Debug/net6.0/Lab_9
Alice says: Nurse Alice, profile: Nurse
Alice is walking
Alice is running
Alice says: I'm assisting the doctor
Performing MRI examination

Bob says: Doctor Bob, profile: Doctor
Bob is walking
Bob is running
Bob says: I am examining patients
Performing X-ray examination
```

Рисунок 1 – Работа консольного приложения

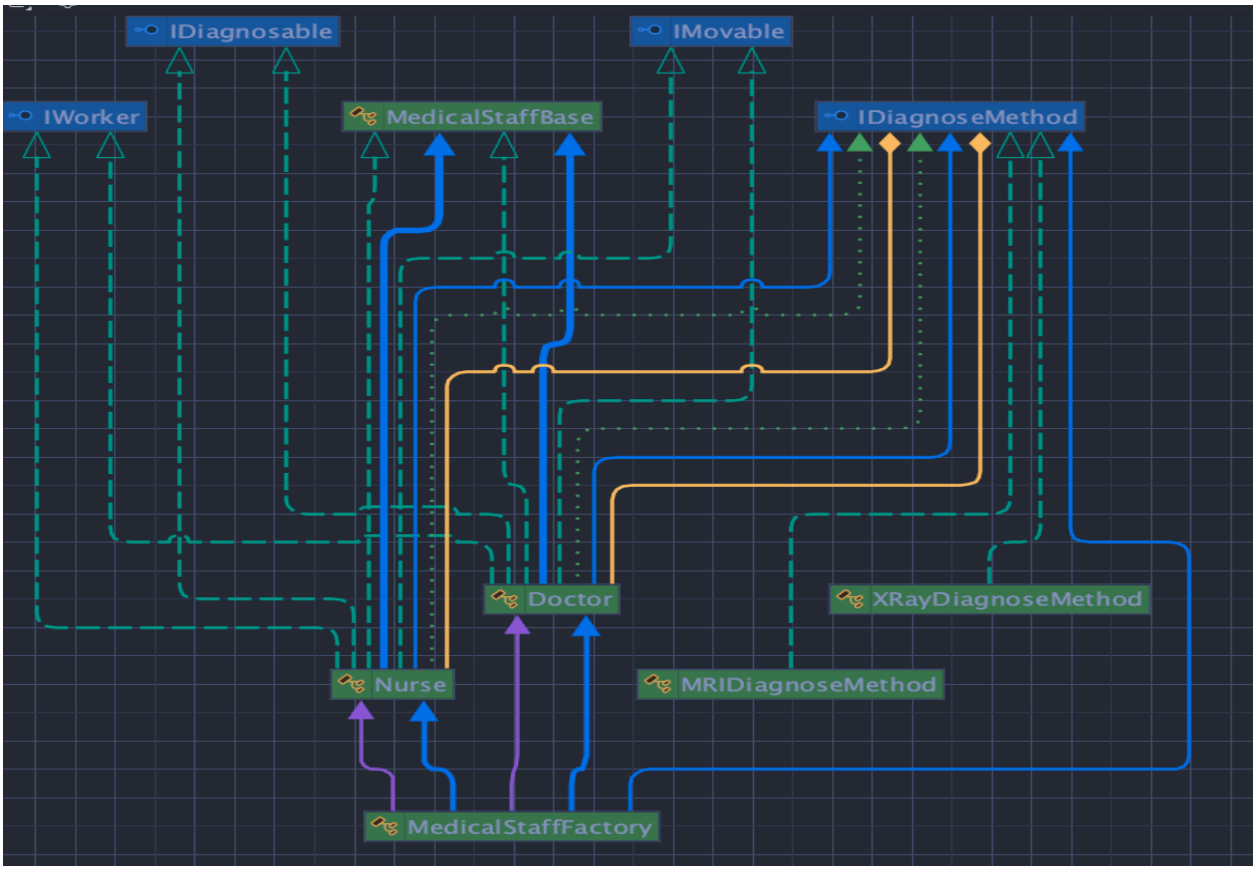

Рисунок 2 – UML-диаграмма приложения

ВЫВОД

В ходе лабораторной работы были получены навыки проектирования приложения, состоящего из нескольких взаимосвязанных классов и интерфейсов. Были применены шаблоны проектирования Мост (Bridge) и Абстрактная фабрика (Abstract factory). Закреплены и отточены все механизмы и принципы работы ООП в C#. Закончено обучение в работе с интерфейсами. Закреплены навыки построения UML-диаграмм.