САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ ИНФОРМАТИКИ

Отчет по лабораторной работе №2

по курсу «объектно-ориентированное программирование»

Тема: Управление инвентарем в ролевой игре

Выполнила:
Буй Тхук Хуен - К3239
Проверил:
Слюсаренко С. В.

Санкт-Петербург
2025 г.

Содержание отчета

I. Задача..... 3
II. Решение..... 3
1. Листинг кода..... 3
2. Текстовое объяснение решения.....14
3. Результат работы код..... 14
III. Вывод.....14

I. Задача

Разработайте систему управления инвентарем для ролевой игры. Система должна:

1. Позволять игрокам добавлять различные типы предметов в инвентарь, такие как оружие, броня, зелья и квестовые предметы.
2. Учитывать уникальные свойства каждого предмета, например, урон для оружия или защиту для брони.
3. Предоставлять возможность использовать или экипировать предметы, а также отображать информацию о текущем состоянии инвентаря.
4. Обеспечивать возможность комбинирования или улучшения предметов.
5. Логику необходимо покрыть unit тестами
6. CLI не нужен
7. Соблюсти по каждой букве из SOLID в различных модулях проекта

Рекомендуемые паттерны:

1. Builder
2. Strategy
3. State
4. Abstract Factory

Задача – создать такую систему, которая бы поддерживала различные типы предметов и их взаимодействие, включая возможность улучшения экипировки.

Цель ЛР заключается в применение принципов ООП для создания сложной системы с множеством взаимодействующих объектов, где правильный выбор паттернов поможет оптимизировать код и соблюсти SOLID.

II. Решение

1. Листинг кода

1.1 Inventory.Core/ Interfaces

- Consume

```
namespace Inventory.Core.Interfaces
{
    public interface IConsumable
    {
        void Consume();
    }
}
```

- Equip

```
namespace Inventory.Core.Interfaces
{
    public interface IEquipable
    {
        void Equip();
        void Unequip();
    }
}
```

- Upgrade

```
namespace Inventory.Core.Interfaces
{
    public interface IUpgradable
    {
        void Upgrade();
    }
}
```

1.2 Inventory.Core/ Items

- Armor

```
using Inventory.Core.Interfaces;

namespace Inventory.Core.Items
{
    public class Armor : Item, IEquipable, IUpgradable
    {
    public int Defense { get; private set; }
    private bool isEquipped = false;

        public Armor(string name, int defense)
        {
            Name = name;
            Defense = defense;
        }

        public void Equip()
        {
            if (!isEquipped)
            {
                isEquipped = true;
                System.Console.WriteLine($"{Name} equipped!");
            }
        }
```

```
        public void Unequip()
        {
            if (isEquipped)
            {
                isEquipped = false;
                System.Console.WriteLine($"{Name} unequipped!");
            }
        }

        public void Upgrade()
        {
            Defense += 3;
                  System.Console.WriteLine($"{Name} upgraded! Defense is now
{Defense}");
        }
    }

}
```

- Item

```
namespace Inventory.Core.Items
{
    public abstract class Item
    {
        public string? Name { get; protected set; }
        public string? Description { get; protected set; }
    }
}
```

- Potion

```
using Inventory.Core.Interfaces;

namespace Inventory.Core.Items
{
    public class Potion : Item, IConsumable
    {
        public int HealAmount { get; private set; }

        public Potion(string name, int heal)
        {
            Name = name;
            HealAmount = heal;
```

```
        }

        public void Consume()
        {
            System.Console.WriteLine($"{Name} consumed! Healed {HealAmount} HP.");
        }
    }
}
```

- QuestItem

```
namespace Inventory.Core.Items
{
    public class QuestItem : Item
    {
        public QuestItem(string name, string description)
        {
            Name = name;
            Description = description;
        }
    }
}
```

- Weapon

```
using Inventory.Core.Interfaces;

namespace Inventory.Core.Items
{
    public class Weapon : Item, IEquipable, IUpgradable
    {
        public int Damage { get; private set; }
        private bool isEquipped = false;

        public Weapon(string name, int damage)
        {
            Name = name;
            Damage = damage;
        }

        public void Equip()
        {
            if (!isEquipped)
            {
```

```
                isEquipped = true;
                System.Console.WriteLine($"{Name} equipped!");
            }
        }

        public void Unequip()
        {
            if (isEquipped)
            {
                isEquipped = false;
                System.Console.WriteLine($"{Name} unequipped!");
            }
        }

        public void Upgrade()
        {
            Damage += 5;
                System.Console.WriteLine($"{Name} upgraded! Damage is now
{Damage}");
        }
    }
}
```

1.3 Inventory.Core/ Patterns

- Builder/ WeaponBuilder

```
using Inventory.Core.Items;

namespace Inventory.Core.Patterns.Builder
{
    public class WeaponBuilder
    {
        private string? name;
        private int damage = 0;

         public WeaponBuilder SetName(string? name) { this.name = name; return
this; }
          public WeaponBuilder SetDamage(int damage) { this.damage = damage;
return this; }

        public Weapon Build()
        {
            if (string.IsNullOrEmpty(name))
                throw new System.InvalidOperationException("Weapon name is
required");
```

```
            return new Weapon(name, damage);
        }
    }
}
```

- State/ EquipState

```
using Inventory.Core.Items;

namespace Inventory.Core.Patterns.State
{
    public class EquippedState : IItemState
    {
        public void Handle(Weapon weapon) => weapon.Unequip();
    }
}
```

- State/ ItemState

```
using Inventory.Core.Items;

namespace Inventory.Core.Patterns.State
{
    public interface IItemState
    {
        void Handle(Weapon weapon);
    }
}
```

- State/ UnequipState

```
using Inventory.Core.Items;

namespace Inventory.Core.Patterns.State
{
    public class UnequippedState : IItemState
    {
        public void Handle(Weapon weapon) => weapon.Equip();
    }
}
```

- State/ WeaponState

```
using Inventory.Core.Items;
```

```
namespace Inventory.Core.Patterns.State
{
    public class WeaponStateContext
    {
        private IItemState state;
        private Weapon weapon;

        public WeaponStateContext(Weapon weapon)
        {
            this.weapon = weapon;
            state = new UnequippedState();
        }

        public void ToggleEquip()
        {
            state.Handle(weapon);
                        state = state is UnequippedState ? (IItemState)new
EquippedState() : new UnequippedState();
        }
    }
}
```

- Strategy/ ConsumeStrategy

```
using Inventory.Core.Interfaces;
using Inventory.Core.Items;

namespace Inventory.Core.Patterns.Strategy
{
    public class ConsumeStrategy : IUseStrategy
    {
```

```
        public void Use(Item item)
        {
            if(item is IConsumable consumable)
                consumable.Consume();
        }
    }
}
```

- Strategy/ EquipStrategy

```
using Inventory.Core.Interfaces;
using Inventory.Core.Items;

namespace Inventory.Core.Patterns.Strategy
{
    public class EquipStrategy : IUseStrategy
    {
        public void Use(Item item)
        {
            if(item is IEquipable equipable)
                equipable.Equip();
        }
    }
}
```

- Strategy/ UseStrategy

```
using Inventory.Core.Items;

namespace Inventory.Core.Patterns.Strategy
{
    public interface IUseStrategy
    {
        void Use(Item item);
    }
}
```

1.4 Inventory.Core/ Inventory

```
using System.Collections.Generic;
using Inventory.Core.Items;
using Inventory.Core.Patterns.Strategy;

namespace Inventory.Core.Inventory
{
    public class InventoryManager
```

```
    {
        private List<Item> items = new();

        public void AddItem(Item item) => items.Add(item);
        public void RemoveItem(Item item) => items.Remove(item);

        public void UseItem(Item item, IUseStrategy strategy)
        {
            strategy.Use(item);
        }

        public void ShowInventory()
        {
            System.Console.WriteLine("Inventory:");
            foreach(var item in items)
                System.Console.WriteLine($"- {item.Name}");
        }

        public List<Item> GetItems() => new List<Item>(items);

    }
}
```

1.5 Inventory.Tests/ Tests

```
using Xunit;
using Inventory.Core.Inventory;
using Inventory.Core.Items;
using Inventory.Core.Patterns.Builder;
using Inventory.Core.Patterns.Strategy;
using Inventory.Core.Patterns.State;

namespace Inventory.Tests
{
public class InventoryTests
{
[Fact]
public void AddAllItemTypesTest()
{
var inventory = new InventoryManager();

                                                var sword = new
WeaponBuilder().SetName("Sword").SetDamage(10).Build();
        var armor = new Armor("Shield", 5);
        var potion = new Potion("Health Potion", 50);
```

```
        var questItem = new QuestItem("Key", "Opens the dungeon");

        inventory.AddItem(sword);
        inventory.AddItem(armor);
        inventory.AddItem(potion);
        inventory.AddItem(questItem);

        Assert.Contains(sword, inventory.GetItems());
        Assert.Contains(armor, inventory.GetItems());
        Assert.Contains(potion, inventory.GetItems());
        Assert.Contains(questItem, inventory.GetItems());
    }

    [Fact]
    public void EquipAndUnequipWeaponTest()
    {
                                                var      sword      =      new
WeaponBuilder().SetName("Sword").SetDamage(10).Build();
        var stateContext = new WeaponStateContext(sword);

        stateContext.ToggleEquip();
        stateContext.ToggleEquip();

        Assert.NotNull(sword);
    }

    [Fact]
    public void EquipArmorTest()
    {
        var inventory = new InventoryManager();
        var armor = new Armor("Shield", 5);
        inventory.AddItem(armor);

        inventory.UseItem(armor, new EquipStrategy());

        Assert.NotNull(armor);
    }

    [Fact]
    public void ConsumePotionTest()
    {
        var inventory = new InventoryManager();
        var potion = new Potion("Health Potion", 50);
        inventory.AddItem(potion);
```

```
        inventory.UseItem(potion, new ConsumeStrategy());

        Assert.NotNull(potion);
    }

    [Fact]
    public void UpgradeItemsTest()
    {
                                                var      sword      =      new
WeaponBuilder().SetName("Sword").SetDamage(10).Build();
        var armor = new Armor("Shield", 5);

        sword.Upgrade();
        armor.Upgrade();

        Assert.True(sword.Damage > 10);
        Assert.True(armor.Defense > 5);
    }

    [Fact]
    public void RemoveItemTest()
    {
        var inventory = new InventoryManager();
                                                var      sword      =      new
WeaponBuilder().SetName("Sword").SetDamage(10).Build();
        inventory.AddItem(sword);

        inventory.RemoveItem(sword);

        Assert.DoesNotContain(sword, inventory.GetItems());
    }

    [Fact]
    public void ShowInventoryTest()
    {
        var inventory = new InventoryManager();
                                                var      sword      =      new
WeaponBuilder().SetName("Sword").SetDamage(10).Build();
        var potion = new Potion("Health Potion", 50);
        inventory.AddItem(sword);
        inventory.AddItem(potion);

        inventory.ShowInventory(); // chỉ in ra console, không cần assert
```

```
    }
}
}
```

2. Текстовое объяснение решения

Inventory.Core/Items:

- **Items:** базовый абстрактный класс предмета. Он определяет общие свойства: Name, Description

  Так как предметы бывают разные, класс объявлен как abstract, и его нельзя создать напрямую. Все конкретные предметы наследуются от него.
  Принцип SOLID: Single Responsibility: класс отвечает только за хранение общих данных предмета.

- **Weapon:** предмет, который можно: экипировать/ снять/ улучшить. Он реализует интерфейсы:
  + IEquipable – возможность экипировки
  + IUpgradable – возможность улучшения

  Методы:
  + Equip() – экипировать оружие
  + Unequip() – снять
  + Upgrade() – увеличить урон

  Принципы SOLID:

  Open/Closed: можно расширять (добавлять новые типы оружия), не изменяя базовый код.
  Interface Segregation: класс реализует только нужные ему интерфейсы.

- **Armor:** Похож на Weapon:
  + можно надеть/снять
  + можно улучшить (увеличить защиту)
- **Potion:** Выпиваемый предмет:
  + не экипируется
  + не улучшается
  + только потребляется (Consume)

  Принцип SOLID: Interface Segregation: зелье реализует только IConsumable, другие интерфейсы ему не навязываются.

- **QuesItem:** Не имеет механики использования – только хранится.

Inventory.Core/ Interfaces:

- **Equip:** Предоставляет функционал экипировки.
- **Consume:** Для предметов, которые можно использовать (выпить или съесть).
- **Upgrade:** Реализуется предметами, которые можно улучшить.

Inventory.Core/ Patterns

- **Builder/ WeaponBuilder:** если у предмета много опциональных параметров (уровни редкости, модификаторы, веса, требуемый уровень). Builder даёт fluent-интерфейс для сборки объекта по шагам.
- **State:** Если предмет имеет сложные состояния (экипирован, повреждён, заряжен и т.д.), State позволяет инкапсулировать поведение для каждого состояния в отдельном объекте.
  WeaponStateContext держит текущее состояние и Weapon. ToggleEquip() вызывает state.Handle(weapon) и меняет состояние на противоположно
- **Strategy/ ConsumeStrategy:** Используется для зелья.
- **Strategy/ EquipStrategy:** Используется для экипировки.
- **Strategy/ UseStrategy:** Определяет общий "интерфейс поведения".

Inventory.Core/ Inventory: Этот класс реализует:

- добавление предметов
- удаление
- вывод содержания инвентаря
- использование предметов через стратегию

Inventory не знает, экипируется предмет или выпивается – это делает Strategy. Принцип SOLID - Dependency Inversion: Inventory зависит от абстракции (IUseStrategy), а не от конкретных действий.

3. Результат работы код

```
PS D:\OOP\OOP2025\Lab3> dotnet build
Restore complete (0.8s)
  Inventory.Core succeeded (0.4s) → Inventory.Core\bin\Debug\net9.0\Inventory.Core.dll
  Inventory.Tests succeeded (0.2s) → Inventory.Tests\bin\Debug\net9.0\Inventory.Tests.dll

Build succeeded in 1.9s
```

III. Вывод

В ходе выполнения лабораторной работы была разработана и протестирована программа – система управления инвентарем для ролевой игры на языке C# с использованием объектно-ориентированного подхода.

Использование паттернов **Strategy**, **State** и **Builder** позволило повысить расширяемость решения, упростить добавление новых типов предметов и обеспечить чёткое разделение обязанностей между компонентами.

- Паттерн **Strategy** обеспечил гибкое переключение логики использования, экипировки и потребления предметов без изменения их структуры.
- Паттерн **State** позволил корректно управлять состояниями экипировки оружия и предотвратить некорректные действия пользователя.
- Паттерн **Builder** упрощает процесс создания сложных объектов (оружия) и сделал его более контролируемым.

Разработанная архитектура соответствует принципам ООП и обеспечивает лёгкую масштабируемость системы. Полученный проект имеет чёткую структуру, поддерживаемый код и возможность дальнейшего расширения функциональности.