ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ
ΣΧΟΛΗ ΟΙΚΟΝΟΜΙΑΣ, ΔΙΟΙΚΗΣΗΣ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΊΩΝ

Εργασία στο μάθημα Τεχνολογία Λογισμικού ΙΙ

Artillery Weapons

Implementation of Creational Design Patterns in Weapon Construction using C#

Επιβλέπων Καθηγητής
ΔΗΜΗΤΡΟΥΛΑΚΟΣ ΓΡΗΓΟΡΙΟΣ

Προπτυχιακός Φοιτητής
ΟΣΜΕΝΑΪ ΖΑΜΙΡ 2022201900157

Ιούνιος 2024

Implementation of Creational Design Patterns in Weapon Construction

Initialization and Blueprint Management

Overview

The program begins by creating a repository where all weapon blueprints are stored. This repository is implemented using the Singleton Design Pattern to ensure a single instance manages all blueprints.

Blueprint Repository Initialization

```
// Create RND Repository which stores all the Blueprints
var repository = BlueprintRepositoryRND.Instance;
```

The `BlueprintRepositoryRND` class contains a private list to store blueprints and ensures a single instance through a private constructor and a public static property.

```
// List to store weapon blueprints
private List<IWeaponBlueprint> _blueprintRegistry = new
List<IWeaponBlueprint>();

// Static instance of the BlueprintRepositoryRND
private static BlueprintRepositoryRND _instance;

// Private constructor to prevent instantiation from outside the class
private BlueprintRepositoryRND() { }

// Public property to access the singleton instance of the class
public static BlueprintRepositoryRND Instance {
 get {
 // Create a new instance if it doesn't already exist
 if (_instance == null) {
 _instance = new BlueprintRepositoryRND();
 }
 return _instance;
 }
}
```

Weapon Blueprint Interface

The `IWeaponBlueprint` interface, located at `Weapon Blueprints/IWeaponBlueprint.cs`, defines the characteristics of a weapon. It inherits from the `IPrototype<IWeaponBlueprint>` and `IWeaponStats` interfaces, allowing the cloning of blueprints.

```
public interface IWeaponBlueprint : IPrototype<IWeaponBlueprint>,
IWeaponStats {
    IMetalCasingBlueprint CasingBlueprint { get; set; }
    IExplosiveBlueprint ExplosiveBlueprint { get; set; }
    IDetonationBlueprint DetonationBlueprint { get; set; }
    IGuidanceKitBlueprint GuidanceKitBlueprint { get; set; }
    ILauncherBlueprint LauncherBlueprint { get; set; }
    uint WeaponVersion { get; set; }
}
```

Blueprint Repository Methods

The repository provides methods to retrieve, add, and iterate over blueprints.

- **GetBlueprint**: Retrieves a blueprint based on the specified weapon family and version.

```
public IWeaponBlueprint GetBlueprint(WeaponFamilies family, int
version) {
    return _blueprintRegistry.FirstOrDefault(x => x.WeaponFamily ==
family && x.WeaponVersion == version);
}
```

- **RegisterBlueprint**: Adds a new blueprint to the repository.

```
public void RegisterBlueprint(IWeaponBlueprint blueprint) {
    _blueprintRegistry.Add(blueprint);
}
```

- **Stats**: Iterates through all blueprints and yields those that implement `IWeaponStats`.

```
public IEnumerable<IWeaponStats> Stats() {
    foreach (var stat in _blueprintRegistry) {
        if (stat is IWeaponStats weaponStat) {
            yield return weaponStat;
        }
    }
}
```

Main Program Execution

Registering and Listing Blueprints

After initializing the repository, the program creates and registers four types of weapon blueprints (FAB, Rocket Projectile, Guided Projectile, and Conventional), each with specific characteristics.

```
// Create and register weapon blueprints
RegisterWeaponBlueprints(repository);
```

To verify that the blueprints are stored correctly, the program lists all weapon stats by printing their names.

```
// List all weapon stats in the repository
ListAllWeaponStats(repository);
```

Weapon Construction with Director and Builder

The Director acts as an orchestrator, guiding the Builder on what weapon to build based on the weapon family and version.

```
// Create a Director (orchestrator)
var director = new Director();
```

The Director identifies the appropriate Manufacturer (factory) responsible for creating the weapon family and passes it to the Builder.

Builder Assembly Process

The Builder assembles the weapon by verifying if each part (Metal Casing, Explosive, Guidance Kit, etc.) exists in the blueprint. If a part exists, the Builder calls the corresponding method (e.g., `BuildMetalCasing`). The Factory method `CreateMetalCasing` is invoked, which initializes the weapon part based on the blueprint and returns it to the Builder. The Builder stores each part locally and assembles them to create the final weapon.

```
if (_blueprint.CasingBlueprint != null) {
 _builder.BuildMetalCasing();
}
if (_blueprint.ExplosiveBlueprint != null) {
 _builder.BuildExplosives();
}
if (_blueprint.GuidanceKitBlueprint != null) {
 _builder.BuildGuidanceKit();
}
if (_blueprint.DetonationBlueprint != null) {
 _builder.BuildDetonation();
}
if (_blueprint.LauncherBlueprint != null) {
 _builder.BuildLauncher();
}

// Return the constructed weapon
return _builder.GetWeapon();
```

Implementation of the Abstract Factory Design Pattern

Overview

The Abstract Factory Design Pattern was implemented to create various weapon parts through a structured and modular approach. This process ensures that each part adheres to a specific interface, promoting consistency and reusability.

Abstract Factory Folder Structure

Within the `Abstract Factory` folder, the core logic is implemented in the `IWeaponPartsFactory.cs` interface. This interface defines the methods that each concrete factory must implement, ensuring uniformity across different types of weapon parts factories.

Concrete Factories

The concrete factories are located in the `Abstract Factory/Concrete Factories` folder. These include:

- **FAB Factory (`CFABFactory.cs`)**
- **Rocket Projectile Factory (`CRocketProjectileFactory.cs`)**
- **Guided Projectile Factory (`CGuidedProjectileFactory.cs`)**
- **Conventional Projectile Factory (`CConventionalProjectileFactory.cs`)**

Each concrete factory implements the following methods defined in the `IWeaponPartsFactory` interface: - `CreateMetalCasing()` - `CreateExplosives()` - `CreateGuidance()` - `CreateDetonation()` - `CreateLauncher()`

These methods represent the essential components required for constructing a weapon.

Weapon Parts Interface

The return types of the methods in `IWeaponPartsFactory.cs` are defined in the `IWeaponParts.cs` interface file. Each product, such as `MetalCasing`, `Explosives`, `GuidanceKit`, etc., has specific attributes. For example, the `IMetalCasingBlueprint` interface includes:

- **Metal Casing Type** (Steel, Titanium, Tungsten, Depleted Uranium)
- **Casing Shape** (Cylindrical, Spherical, Conical, Cubic)
- **Weight** (in kg)
- **Thickness** (in mm)

This structure is mirrored for other products, ensuring that each weapon part is described by its own set of attributes.

Concrete Weapon Parts

For each product, a concrete implementation exists within the `Abstract Factory/Concrete Weapon Parts` folder. These files include:

- `CFABWeaponParts.cs`
- `CRocketProjectileWeaponParts.cs`
- `CGuidedProjectileWeaponParts.cs`
- `CConventionalProjectileWeaponParts.cs`

Each file implements the `IWeaponParts.cs` interface, detailing the characteristics and internal specifications of the weapon parts for the respective factory.

Cloning with IPrototype

Each weapon part also implements the `IPrototype` interface, defined in the `Weapon Blueprints/IWeaponBlueprint.cs` folder. This interface enables each class to clone itself, which provides the advantage of reusing existing parts without recreating them from scratch.

```
// Interface for a factory to create various parts of a weapon
public interface IWeaponPartsFactory {
    IMetalCasingBlueprint CreateMetalCasing();
    IExplosiveBlueprint CreateExplosive();
    IGuidanceKitBlueprint CreateGuidance();
    IDetonationBlueprint CreateDetonation();
    ILauncherBlueprint CreateLauncher();
}

// Interface for metal casing blueprint, inheriting from IPrototype
public interface IMetalCasingBlueprint :
IPrototype<IMetalCasingBlueprint> {
    MetalCasingType MetalCasing { get; set; }
    MetalCasingShape CasingShape { get; set; }
    double WeightKG { get; set; }
    double ThicknessMM { get; set; }
}

// Generic interface IPrototype with covariant type parameter T
public interface IPrototype<out T> {
    T Clone();
}
```

Conclusion of Abstract Factory

This implementation effectively demonstrates the use of the Abstract Factory Design Pattern to manage and produce complex weapon parts. The use of interfaces and concrete classes ensures modularity and scalability, making the system robust and adaptable.

Implementation of the Builder Design Pattern

Overview

The Builder Design Pattern is implemented using a Director, which orchestrates the assembly and delivery of various weapons. This design allows for the construction of complex objects step by step.

Director and Weapon Construction

The Director is responsible for taking the weapon family (`WeaponFamily`) and its version as input to construct the weapon.

```
var weapon = director.Construct(family, version);
```

This approach supports different weapon families (FAB, Rocket Projectile, Guided Projectile, and Conventional) and their respective versions. For example, FAB bombs come in multiple versions like FAB250, FAB500, FAB5000, and FAB9000, each with unique characteristics.

Based on the provided parameters (Weapon Family and version), the Director identifies the appropriate Manufacturer (factory) to supply the parts required by the Builder.

```
var (_builder, _blueprint) = GetWeaponManufactures(family, version);
```

Blueprint Retrieval and Validation

The system first checks if the blueprint for the specified weapon family and version exists in the repository.

```
IWeaponBlueprint blueprint =
BlueprintRepositoryRND.Instance.GetBlueprint(family, version);

if (blueprint == null) {
    throw new ArgumentException("Invalid weapon version");
}
```

If the blueprint is found, the Factory responsible for that weapon is initialized with the weapon version. This Factory then knows the specifications for each part of the weapon and returns the parts to the Builder.

```
switch (family) {
    case WeaponFamilies.FAB:
        CFABFactory fabFactory = new CFABFactory(version);
        return (new CFABBuilder(fabFactory), blueprint);
    ...
}
```

Weapon Assembly

Assuming no errors are encountered, the Builder assembles each part of the weapon. The Director then combines these parts to produce the final weapon, which is delivered to the Army Command (the main program).

```
if (_blueprint.CasingBlueprint != null) {
 _builder.BuildMetalCasing();
}
if (_blueprint.ExplosiveBlueprint != null) {
 _builder.BuildExplosives();
}
if (_blueprint.GuidanceKitBlueprint != null) {
 _builder.BuildGuidanceKit();
}
if (_blueprint.DetonationBlueprint != null) {
 _builder.BuildDetonation();
}
if (_blueprint.LauncherBlueprint != null) {
 _builder.BuildLauncher();
}

return _builder.GetWeapon();
```

Code Reusability and Abstract Class Implementation

During implementation, it was observed that there was significant code reuse among the Concrete Builders: `CFABBuilder.cs`, `CRocketProjectileBuilder.cs`, `CGuidedProjectileBuilder.cs`, and `CConventionalProjectileBuilder.cs`. To address this, a base abstract class called `BaseWeaponBuilder` was created. Common code was moved to this class, and Concrete Builders inherited from it.

```
public abstract class BaseWeaponBuilder {
    public IMetalCasingBlueprint _metalCasing;
    public IExplosiveBlueprint _explosive;
    public IGuidanceKitBlueprint _guidanceKit;
    public IDetonationBlueprint _detonation;
    public ILauncherBlueprint _launcher;

    public IWeaponPartsFactory _weaponPartsFactory;

    public BaseWeaponBuilder(IWeaponPartsFactory factory) {
        _weaponPartsFactory = factory;
    }

    public virtual void BuildMetalCasing() {
        _metalCasing = _weaponPartsFactory.CreateMetalCasing();
    }

    public virtual void BuildExplosives() {
        _explosive = _weaponPartsFactory.CreateExplosive();
    }

    public virtual void BuildGuidanceKit() {
        _guidanceKit = _weaponPartsFactory.CreateGuidance();
    }

    public virtual void BuildDetonation() {
        _detonation = _weaponPartsFactory.CreateDetonation();
    }

    public virtual void BuildLauncher() {
        _launcher = _weaponPartsFactory.CreateLauncher();
    }

    public abstract IWeapon GetWeapon();
}
```

Concrete Builders and Polymorphism

Concrete Builders inherit from `BaseWeaponBuilder` and implement the `GetWeapon` method to create specific weapons.

```
public class CFABBuilder : BaseWeaponBuilder {
    public CFABBuilder(CFABFactory factory) : base(factory) { }
}

public abstract IWeapon GetWeapon();

// Return the constructed weapon
return _builder.GetWeapon();
```

This approach utilizes polymorphism to pass Concrete Factories to the superclass constructor, leveraging the Abstract Factory Design Pattern methods.

Final Product

The result of this process is the assembly of the final weapon. Each weapon type (FAB, Rocket Projectile, Guided Projectile, and Conventional) is created as a concrete product, implemented within the `Weapons` folder. An interface called `IWeapon` is used for generalization and management of the weapon's result.

Implementation of the Prototype Design Pattern

Overview

The Prototype Design Pattern is implemented in the `WeaponBlueprints` folder. This pattern allows for the cloning of existing weapon blueprints, avoiding the need to recreate them from scratch.

Weapon Blueprints

The `WeaponBlueprints` folder contains two key files: `IWeaponBlueprint.cs` and `WeaponBlueprint.cs`. These files define the characteristics a weapon should have and implement the `IPrototype` interface, enabling cloning.

IPrototype Interface

The `IPrototype` interface ensures that any implementing class can clone itself.

```
// Generic interface IPrototype with covariant type parameter T.
// This interface ensures that any implementing class can clone
itself.
public interface IPrototype<out T> {
    T Clone();
}
```

IWeaponBlueprint Interface

The `IWeaponBlueprint` interface inherits from `IPrototype<IWeaponBlueprint>` and `IWeaponStats`. It specifies the components and properties related to a weapon blueprint.

```
// Interface for weapon blueprint, inheriting from IPrototype with
IWeaponBlueprint as the type
// and also from IWeaponStats.
// This interface specifies the components and properties related to a
weapon blueprint.
public interface IWeaponBlueprint : IPrototype<IWeaponBlueprint>,
IWeaponStats {
    IMetalCasingBlueprint CasingBlueprint { get; set; }
    IExplosiveBlueprint ExplosiveBlueprint { get; set; }
    IDetonationBlueprint DetonationBlueprint { get; set; }
    IGuidanceKitBlueprint GuidanceKitBlueprint { get; set; }
    ILauncherBlueprint LauncherBlueprint { get; set; }
    uint WeaponVersion { get; set; }
}
```

Cloning Blueprints

By implementing the Prototype Design Pattern, the system can efficiently clone weapon blueprints, streamlining the process of creating new weapons without starting from scratch. This implementation enhances flexibility and reduces redundancy in the weapon creation process.

Conclusion

Through this implementation, we successfully employed all five Creational Design Patterns—Factory Method, Abstract Factory, Builder, Prototype, and Singleton. Each pattern not only addressed specific challenges in creating complex objects but also worked synergistically to form a cohesive and efficient system.

- **Factory Method**: By defining an interface for creating an object but allowing subclasses to alter the type of objects that will be created, we achieved flexibility in our weapon creation process. This was evident in the specialized factories like `CFABFactory` and `CRocketProjectileFactory`.
- **Abstract Factory**: The Abstract Factory pattern provided an interface for creating families of related or dependent objects without specifying their concrete classes. This allowed us to create varied components of a weapon, such as casings and guidance kits, through specific factories that adhered to a common interface, ensuring consistency and scalability.
- **Builder**: The Builder pattern facilitated the construction of complex objects step by step. The Director orchestrated the building process, allowing us to construct different versions of weapons systematically by assembling their parts as defined in their blueprints. This pattern also enhanced the flexibility to create different configurations of weapons.
- **Prototype**: Implementing the Prototype pattern enabled us to clone existing blueprints, thereby avoiding the redundancy of creating new blueprints from scratch. This not only improved efficiency but also allowed for quick iterations and modifications of weapon designs.
- **Singleton**: The Singleton pattern ensured that the `BlueprintRepositoryRND` was instantiated only once, maintaining a single repository of all blueprints. This centralized repository was crucial for managing and retrieving blueprints, ensuring consistency and preventing conflicts.

The interplay between these patterns was essential. The Singleton pattern provided a centralized repository, which was critical for the Builder pattern to retrieve blueprints. The Factory Method and Abstract Factory patterns provided the necessary components that the Builder used to assemble weapons. Finally, the Prototype pattern allowed for the efficient reuse of blueprints, making the entire system more flexible and scalable.

By integrating these Creational Design Patterns, we created a robust and flexible system capable of managing the complexities of weapon creation. Each pattern played a distinct role while complementing the others, resulting in a well-structured and efficient implementation. This synergy not only streamlined the process but also showcased the power of combining design patterns to tackle intricate problems in software design.