Object Oriented programming and software engineering – Lab 18

Adam Korytowski – 2025

1. Factory design pattern.

The Factory Pattern is a design pattern that provides an interface for creating objects without specifying their concrete classes.

Instead of using new directly in your code (e.g., new Goblin()), the object creation is delegated to a factory class or method. This approach encapsulates the logic of object creation, making your code more flexible, scalable, and decoupled.

When Should You Use the Factory Pattern? Use the Factory Pattern when:

- You want to decouple object creation from the main logic – for example, a battle system creates enemies, but it doesn't need to know *which specific enemy class* is instantiated.
- You have a family of related objects (e.g., Orc, Goblin, Troll) and want to create them dynamically.
- You want to encapsulate object creation logic, especially if: It's complex (e.g., setting up stats, loading assets) or when it depends on configuration or runtime conditions.
- You want to introduce new types in the future without changing existing client code. – Just create a new factory subclass — no need to touch core gameplay logic.

Key Benefits:

Feature	Benefit
Decoupling	Client code depends on interfaces, not classes.
Flexibility	Easily add new product types.
Centralized object creation	Manage how and when objects are created.

Testability	Easier to mock object creation for testing.

2. Example:

Having base and derived classes:

```
class Character
{
public:
    virtual void speak() = 0;
    virtual ~Character() {}
};

class Enemy
{
public:
    virtual void attack() = 0;
    virtual ~Enemy() {}
};

class Warrior : public Character
{
public:
    void speak() override
    {
        std::cout << "I am a Warrior." << std::endl;
    }
};
```

```cpp
class Mage : public Character
{
public:
    void speak() override
    {
        std::cout << "I am a Mage." << std::endl;
    }
};


class Goblin : public Enemy
{
public:
    void attack() override
    {
        std::cout << "Goblin attacks!" << std::endl;
    }
};

class Orc : public Enemy
{
public:
    void attack() override
    {
        std::cout << "Orc attacks!" << std::endl;
    }
};
```

We need to create factory classes for them:

```cpp
class CharacterFactory
{
public:
    virtual Character* createCharacter() = 0;
    virtual ~CharacterFactory() {}
};

class EnemyFactory
{
public:
    virtual Enemy* createEnemy() = 0;
    virtual ~EnemyFactory() {}
};
```

```
class WarriorFactory : public CharacterFactory
{
public:
    Character* createCharacter() override
    {
        return new Warrior();
    }
};

class MageFactory : public CharacterFactory
{
public:
    Character* createCharacter() override
    {
        return new Mage();
    }
};
```

```
class GoblinFactory : public EnemyFactory
{
public:
    Enemy* createEnemy() override
    {
        return new Goblin();
    }
};

class OrcFactory : public EnemyFactory
{
public:
    Enemy* createEnemy() override
    {
        return new Orc();
    }
};
```

3. Tasks.

- Use your code from past laboratories (containing base and derived classes) and implement the Factory design pattern. Use it for creation of your objects, replacing existing implementations (5 pts), e. g.:

```
EnemyFactory* goblinFactory = new GoblinFactory();
EnemyFactory* orcFactory = new OrcFactory();

CharacterFactory* warriorFactory = new WarriorFactory();
CharacterFactory* mageFactory = new MageFactory();

Enemy* goblin = goblinFactory->createEnemy();
Enemy* orc = orcFactory->createEnemy();
Character* warrior = warriorFactory->createCharacter();
Character* mage = mageFactory->createCharacter();
```

- create a template class AutoFactory deriving from one of your base factory classes, for the purpose of creating factories for derived classes, the use it in main() for two different classes (2 pts)