Abstract Factory Pattern - How This Project Matches the Design Pattern

The Gang of Four Definition

From *Design Patterns: Elements of Reusable Object-Oriented Software* (1994):

> **Abstract Factory**: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

How Your Code Maps to the Pattern

The GoF book defines these participants:

1. Abstract Factory (`ICardFactory`)

```
ICardFactory
├── CreateCreature(cardName) → ICreature
└── CreateSpell(cardName) → ISpell
```

This is the **abstract factory interface**. It declares *what* products can be created, but not *how*. Any code that uses `ICardFactory` doesn't know (or care) whether it's getting a Red deck or Blue deck.

Your code: Interfaces/ICardFactory.cs

2. Concrete Factories (`RedDeckFactory` , `BlueDeckFactory`)

```
RedDeckFactory : ICardFactory
├── CreateCreature() → returns RedCreature
└── CreateSpell() → returns RedSpell

BlueDeckFactory : ICardFactory
├── CreateCreature() → returns BlueCreature
└── CreateSpell() → returns BlueSpell
```

These **implement** the abstract factory interface. Each concrete factory creates a *family* of related products (all Red cards, or all Blue cards).

Your code: Factories/RedDeckFactory.cs, Factories/BlueDeckFactory.cs

3. Abstract Products (`ICreature` , `ISpell`)

```
ICreature                    ISpell
├── GetName()                ├── GetName()
├── GetManaCost()            ├── GetManaCost()
├── GetPowerToughness()      ├── GetKeywords()
├── GetKeywords()            ├── GetText()
├── GetText()                └── DisplayDetails()
└── DisplayDetails()
```

These define the interface for each *type* of product. The client code works with creatures and spells through these interfaces.

Your code: Interfaces/ICreature.cs, Interfaces/ISpell.cs

4. Concrete Products (`RedCreature` , `BlueCreature` , `RedSpell` , `BlueSpell`)

These are the actual implementations. Each concrete product implements an abstract product interface.

Your code: Products/RedCreature.cs, etc.

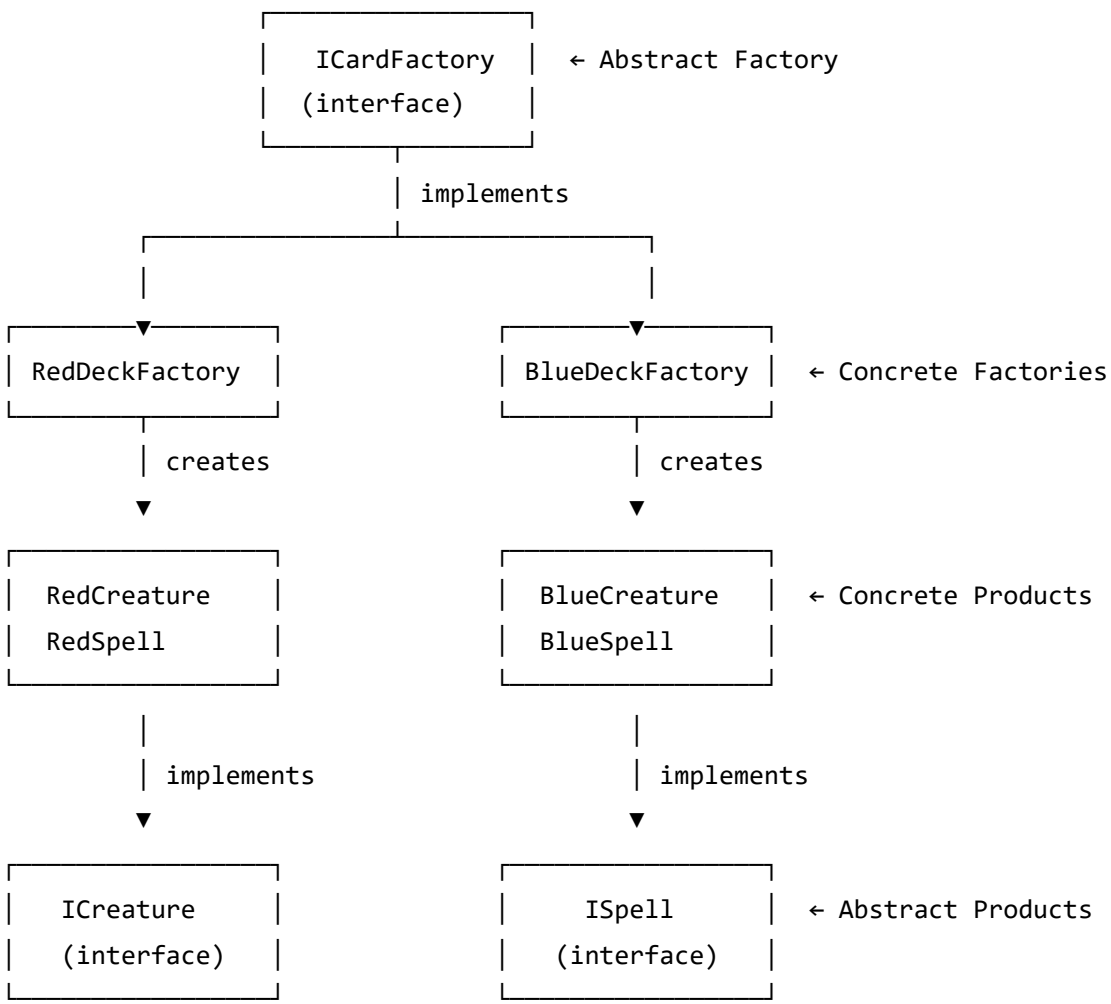
5. Client (`Program.cs` / `DemonstrateFactory`)

```
static void DemonstrateFactory(ICardFactory factory, string creatureName, string spellName)
{
    ICreature creature = factory.CreateCreature(creatureName);
    ISpell spell = factory.CreateSpell(spellName);
    // ...
}
```

The client **only** uses the abstract interfaces (`ICardFactory` , `ICreature` , `ISpell`). It never references `RedDeckFactory` or `BlueCreature` directly. This is the key benefit!

Your code: Program.cs:72-83

Visual Diagram



Why Is It "Abstract"?

The pattern is called "Abstract Factory" because:

1. **The factory is abstract** - `ICardFactory` doesn't create anything itself. It's an interface that defines *what* a factory must be able to do.
2. **The products are abstract** - `ICreature` and `ISpell` don't represent real cards. They define what any creature or spell must provide.
3. **The client code is decoupled** - Your `DemonstrateFactory` method works with abstractions, not concrete types. It can use *any* factory without changes.

The "abstraction" is the separation between:

- **What** you want (a creature, a spell)
- **How** it's made (Red version vs Blue version)

Adding a New Deck - The Easy Part

Yes! Adding a new deck (like Green, Black, or White) is straightforward:

Step 1: Create Concrete Products

```
// Products/GreenCreature.cs
public class GreenCreature : ICreature
{
    // Implement all ICreature methods
    public string GetName() { ... }
    public string GetManaCost() { ... }
    public string GetPowerToughness() { ... }
    public string GetKeywords() { ... }
    public string GetText() { ... }
    public void DisplayDetails() { ... }
}

// Products/GreenSpell.cs
public class GreenSpell : ISpell
{
    // Implement all ISpell methods
}
```

Step 2: Create Concrete Factory

```
// Factories/GreenDeckFactory.cs
public class GreenDeckFactory : ICardFactory
{
    public ICreature CreateCreature(string cardName)
    {
        return new GreenCreature(cardName);
    }

    public ISpell CreateSpell(string cardName)
    {
        return new GreenSpell(cardName);
    }
}
```

Step 3: Use It

```
ICardFactory factory = new GreenDeckFactory();
DemonstrateFactory(factory, "Llanowar Elves", "Giant Growth");
```

That's it! The `DemonstrateFactory` method works without any changes because it only depends on `ICardFactory`, not on any specific factory.

The GoF Book's Key Benefits (Applied to Your Code)

GoF Benefit	Your Implementation
Isolates concrete classes	Client uses `ICardFactory`, never sees `RedDeckFactory` directly
Makes exchanging product families easy	Switch from Red to Blue by changing one line: `new BlueDeckFactory()`
Promotes consistency among products	A `RedDeckFactory` always creates Red products; you can't accidentally mix
Supporting new kinds of products is difficult	Adding a new product type (e.g., `IEnchantment`) requires changing all factories

Summary

Question	Answer
Is "inherit" correct?	For interfaces, say "**implements**". For classes, say "inherits/extends".
Why is it "abstract"?	The factory and products are interfaces (abstractions), not concrete types.
Is adding a new deck easy?	Yes! Implement `ICreature`, `ISpell`, and `ICardFactory`, and you're done.
What makes the pattern work?	Client code depends on interfaces, so concrete types can be swapped freely.

References

- *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, Helm, Johnson, Vlissides, 1994) - Chapter 3: Creational Patterns
- Refactoring Guru - Abstract Factory