CS 247

Software Engineering Principles

Chapter 12

The Factory Pattern

Victoria Sakhnini

Table of Contents

The Problem: A Pizza Shop That Won't Stop Growing 4
What changes, what doesn't 4
The temptation: a conditional 4
First Refactor: Simple Factory 6
What the code now looks like 6
Is Simple Factory a "pattern"? 7
The Factory Method Pattern 8
Why this matters for the pizza shop 8
The three benefits 9
Dependency Inversion 9
Three concrete places are unavoidable 10
Why is it called "inversion" 10
Anatomy: Creator and Product Hierarchies 11
The Creator hierarchy 12
Designing the Franchised Pizza Store 13
Step 2: Declare createPizza as virtual on the base 13
Step 3: subclass per region 14
The runtime trace 14
Implementing the Pizza Store in C++ 15
The Pizza header 15
Pizza implementation 16
NYStyleCheesePizza 17
ChicagoStyleCheesePizza 18
PizzaStore: the abstract creator 19
PizzaStore::orderPizza 20
NYPizzaStore and ChicagoPizzaStore 21
The driver 22
Caveats and Modern C++ Refinements 23
Why std::unique_ptr is the right return type 23
Why std::string_view for the type parameter 24
enums beat strings for the product type 24
Duplicate switch chains across concrete creators 24
The relationship between factories 25

Don't reach for factories when a constructor would do 25
Abstract Factory: One Step Beyond 26
The motivating story 26
The canonical UML 27
Factory Method vs. Abstract Factory 28
Factories in the Wild 28
Cross-platform UI 29
Document and file-format selection 29
Payment gateways 29
Database drivers and ORMs 29
Logging frameworks 30
The pattern's footprint 30
Summary 31
Practice Problems 32

The Problem: A Pizza Shop That Won't Stop Growing

You own a pizza shop in Objectville. Business is good. You started by selling one kind of pizza, but customers want choice: cheese, pepperoni, veggie, and clam. As a cutting-edge object-oriented shop owner, you reach for code first. Your first version of `orderPizza` looks like this:

```
Pizza orderPizza() {
    Pizza pizza = new Pizza();
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

Read it carefully. The function takes no arguments. It hard-codes the call to `new Pizza()`. And it always returns the same kind of pizza, because there is only one kind. This is fine for a pizzeria that sells exactly one item, and very few do.

What changes, what doesn't

Step back from the code and ask what will change about a pizza shop over time. Two things, mostly:

- **The menu of pizza types is going to grow.** Today: cheese, pepperoni, veggie, clam. Next month: vegan, gluten-free, BBQ chicken, Hawaiian. Next year: whatever the chef invents.
- **The process for preparing and serving a pizza is roughly stable.** You prepare the pizza, bake it, cut it, and box it. The *steps* of `orderPizza` are the same regardless of pizza type. This routine has remained the same for years and will continue to do so.

Last chapter's design wisdom, *identify what varies, encapsulate it, and keep what is stable separate*, applies here in the cleanest possible way. The varying part is "which pizza". The stable part is "prepare-bake-cut-box". We want a design where you can change the first without touching the second.

The temptation: a conditional

The first instinct of most students is to pass the pizza type as an argument and pick a class with an `if` chain:

```
Pizza orderPizza(std::string_view type) {
    Pizza* pizza = nullptr;
    If (type == "cheese") pizza = new CheesePizza();
    else if (type == "pepperoni") pizza = new PepperoniPizza();
```

```
    else if (type == "veggie") pizza = new VeggiePizza();
    else if (type == "clam") pizza = new ClamPizza();
    pizza->prepare();
    pizza->bake();
    pizza->cut();
    pizza->box();
    return pizza;
}
```

This works. It even has a certain honest brutality. So what is wrong with it?

- **Every new pizza type requires editing orderPizza.** That violates the Open–Closed Principle from the previous chapter; the function should be closed for modification.
- **orderPizza is now a junction.** It hard-codes the names of every concrete pizza class. The function exists at the intersection of "what varies" and "what is stable", and that intersection is the worst place for change to happen.
- **Test surface area explodes.** Every test of `orderPizza` has to consider every branch. Adding a new pizza adds a new branch and forces re-testing of everything else.
- **Multiple orderPizza-like methods.** If anywhere else in the codebase has to construct a pizza, for delivery, for catering, for the loyalty-card sample-of-the-week, that other code copies the same `if` chain. Updates to the menu now have to be replicated across many files.

We need somewhere else to put the "which pizza" decision. That somewhere will be its own object, with one job: building pizzas.

> **? Question**
>
> Before reading on, sketch a class diagram for your refactored design. Where does the `if` chain go? What does `orderPizza` look like afterwards? How many classes are involved?

First Refactor: Simple Factory

The first move is the obvious one: extract the pizza-construction code into its own class. We will call this class `SimplePizzaFactory`. Its only job is to produce pizzas. Everything else, the prepare/bake/cut/box routine, stays in `PizzaStore`.

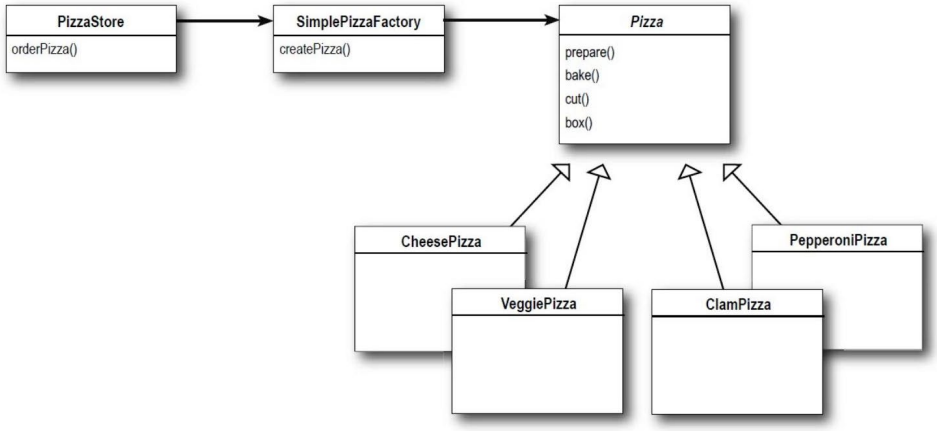

Figure 2.1: The Simple Factory refactor. PizzaStore depends on SimplePizzaFactory, which knows about all the concrete Pizza subclasses. PizzaStore itself does not.

Three classes, two associations, and a clean separation of concerns:

- **PizzaStore**, the client. It contains `orderPizza`, which is now stable: "ask the factory for a pizza of the requested type, then prepare, bake, cut, box, return".
- **SimplePizzaFactory**, the factory. Its single method `createPizza(type)` contains the `if` chain. This is the only place in the codebase that knows the concrete pizza class names.
- **Pizza** and its concrete subclasses, the products. They look exactly as they did before.

What the code now looks like

Here is the cleaned-up version of `orderPizza`:

```
std::unique_ptr<Pizza> PizzaStore::orderPizza(std::string_view type) {
    auto pizza = factory.createPizza(type);
    if (pizza) {
        pizza->prepare();
        pizza->bake();
        pizza->cut();
        pizza->box();
    }
```

```
    return pizza;
}
```

Read it next to the version with the embedded `if` chain. The structure is identical, get a pizza, run the four-step routine, but the "get a pizza" line is now a single call to `factory.createPizza`. The pizza class names have disappeared from this file entirely. The `if` chain still exists somewhere, but it is locked inside the factory, where it belongs.

Is Simple Factory a "pattern"?

Strictly speaking, no. The Gang of Four did not give it a name. The lecture describes it as "a simple way to decouple your clients from concrete classes" rather than as a pattern in its own right. Some authors call it "Simple Factory"; others, "Static Factory Method"; others, just "the factory idiom".

Why it isn't a pattern matters. Simple Factory hard-codes the *set* of products it can produce. The factory's `createPizza` method has to be edited every time a new pizza type is added, and the `if` chain grows. We have moved the problem rather than solved it. The OCP violation is now in the factory instead of in `orderPizza`, but it still exists.

What Simple Factory *is* good for: a small, fixed set of product types that you do not expect to grow. Single-region pizza shop with four pizzas? Simple Factory is perfect. Multi-region franchise with each region adding its own toppings, doughs, sauces, and pizza styles? Simple Factory falls down, and that is where the genuine Factory Method Pattern earns its keep.

> **Tip**
>
> **Static or non-static?** Simple Factory's `createPizza` method is often static. This means callers can write `SimplePizzaFactory::createPizza("cheese")` without instantiating a factory. The trade-off: a static method cannot be overridden by subclasses, which makes *Simple* Factory a dead end if you ever want different factories. Factory Method (next section) makes the create method virtual and restores flexibility.

The Factory Method Pattern

Simple Factory puts the construction logic in a *separate class*. The Factory Method Pattern goes one step further by placing the construction logic in a separate method that subclasses can override. This is what "factory method" literally means: a method whose job is to manufacture a product, where different subclasses manufacture different products.

Here is the GoF definition:

> **Tricky**
>
> **Factory Method Pattern.** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Read that twice. Each clause is doing work:

- **"Define an interface for creating an object."** There is a method, with a signature, that returns the abstract product type. Clients call this method by name.
- **"Let subclasses decide which class to instantiate."** The method is virtual (in C++) or abstract (in Java). The base class declares it; subclasses override it with a specific concrete class to construct.
- **"Defer instantiation to subclasses."** The base class never instantiates the concrete product itself. It calls the factory method and trusts the override to do the right thing.

The third clause is the structural difference from Simple Factory. In the Simple Factory, the factory class instantiates products directly using an if-else chain. In the Factory Method, the factory class is abstract; it never instantiates anything directly. Each concrete subclass overrides the factory method to produce its own kind of product.

Why this matters for the pizza shop

Imagine Objectville Pizza has been so successful that you are franchising it. New York wants a franchise. Chicago wants a franchise. And, predictably, New York's idea of a cheese pizza is not Chicago's idea of a cheese pizza. NY style is thin crust, marinara sauce, and grated Reggiano. Chicago style is extra-thick crust, plum tomato sauce, and shredded mozzarella, cut into squares instead of triangles.

With Simple Factory, you would have to write `NYSimplePizzaFactory` and `ChicagoSimplePizzaFactory`, two separate factory classes, each with its own `if` chain. And then `PizzaStore` would need to know which factory to use. The decision creeps back into the client.

With Factory Method, the answer is more elegant: make `PizzaStore` itself abstract, with a virtual `createPizza` method. `NYPizzaStore` and `ChicagoPizzaStore` inherit from `PizzaStore`, and each overrides `createPizza` to produce its regional pizzas. The `orderPizza` method stays in the base class;

it doesn't care which region it's running in, because it calls `createPizza` and lets virtual dispatch do the work.

The three benefits

- **Flexibility.** You can introduce new product types without changing existing code. A new region, Seattle, say, needs a new `SeattlePizzaStore` subclass with its own `createPizza` override. `PizzaStore::orderPizza` and every existing region's code are untouched.
- **Separation of concerns.** The creation logic ("which class do I instantiate?") is separated from the business logic ("what do I do with the product once I have it?"). These two questions live in different methods, possibly in different files.
- **Open–Closed Principle.** The base class is closed to modification (you don't edit PizzaStore::orderPizza) but open to extension (you can add new subclasses freely).

The pattern is, in effect, a structural recipe for honouring the Open–Closed Principle whenever object construction is what varies. If you find yourself wanting an *orderPizza-like* method that performs a stable routine but constructs a varying product, Factory Method is almost certainly what you want.

Dependency Inversion

> **Tricky**
>
> **Design Principle: Dependency Inversion.** Depend upon abstractions. Do not depend upon concrete classes.

"Dependency" here means: "my code mentions, by name, that other class". If `orderPizza` contains the word `CheesePizza`, then `orderPizza` depends on `CheesePizza`. Dependency Inversion says: do that with abstract types (e.g., `Pizza`) instead of concrete types (e.g., `CheesePizza`).

There are three guidelines that crystallize this:

- No variable should hold a reference to a concrete class.
- No class should derive from a concrete class.
- No method should override an implemented method of any of its base classes.

> **? Question**
> **Is this even possible?** If we follow these rules literally, who ever instantiates anything? Concrete classes must exist somewhere; concrete code must run sometime; the abstract pyramid has to bottom out at *something* real.

These are guidelines, not absolute rules. The point is to *minimize* dependencies on concrete classes, not to eliminate them. Specifically:

Three concrete places are unavoidable

- **In factories.** Some code has to say `new CheesePizza()`. The whole point of a factory is to be the *one* place that has to. The rest of your application sees only `Pizza`.
- **At the program's main entry point.** Someone has to instantiate the first concrete factory. In our pizza shop, the `main` picks between `NYPizzaStore` and `ChicagoPizzaStore`, usually based on a configuration file or a command-line argument. Once that first concrete object exists, the rest of the program can be coded against abstractions.
- **In tests.** Unit tests routinely instantiate concrete classes, both production classes and mocks/fakes/stubs. This is fine. Tests are allowed to know more than the production code they exercise.

Outside those three places, every variable, parameter, and return type should be abstract. Every time you find yourself writing `ConcreteX` elsewhere, ask whether it could be an interface or an abstract base class instead.

Why is it called "inversion"

In a naïve design, high-level code (such as `orderPizza`) depends on low-level code (such as `CheesePizza`). The flow of dependency runs downward: the business logic is at the top, and the concrete details are at the bottom; the top knows the bottom.

Dependency Inversion flips this: both the high-level and low-level code depend on an abstract type in the middle. `orderPizza` depends on `Pizza`; `CheesePizza` depends on `Pizza`. The arrows now both point up at `Pizza`, instead of one arrow pointing from the top to the bottom. The dependency has been *inverted*.

Architecturally, the consequence is that high-level code becomes *policy* and low-level code becomes *detail*. The policy never depends on the detail; the detail must conform to the policy. That is the structural shape of every well-decoupled system, from operating systems to ML frameworks.

Anatomy: Creator and Product Hierarchies

The Factory Method Pattern has two parallel hierarchies, one for the products and one for the creators.

The Product hierarchy

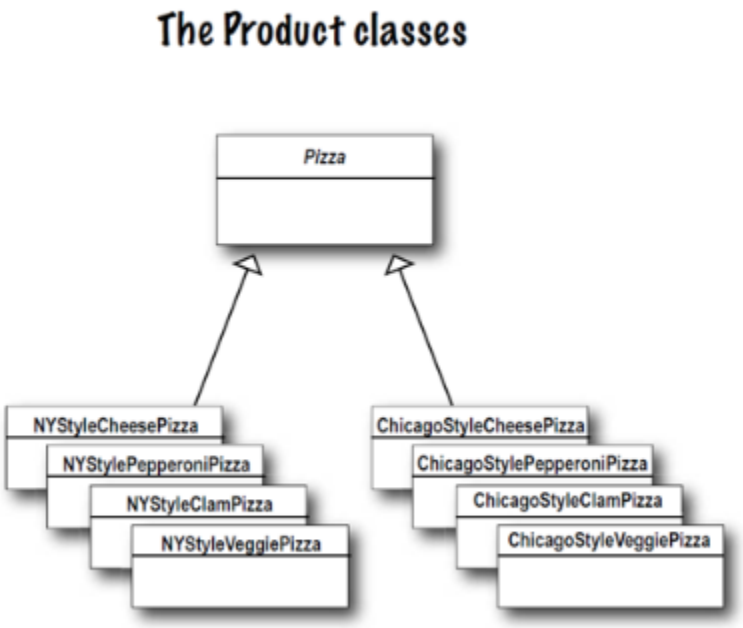

Figure 5.1: The Product hierarchy. Pizza is the abstract product. Concrete products are all the pizzas that get produced by the stores, NY style, Chicago style, in each variety.

Two layers:

- **Pizza (abstract).** The interface that the rest of the application talks to. Defines the methods every pizza must support: `prepare`, `bake`, `cut`, box, `getName`. Concrete state, name, dough, sauce, toppings, may also be declared at this level if it's shared by every concrete pizza.
- **Concrete pizzas.** The actual implementations: `NYStyleCheesePizza`, `NYStylePepperoniPizza`, `NYStyleClamPizza`, `NYStyleVeggiePizza`, and the equivalent Chicago-style four. Each concrete pizza fills in its own name/dough/sauce/toppings and may override `cut` (Chicago slices squares, not triangles).

Notice the multiplicity: with two regions and four pizza varieties, we have 2 × 4 = 8 concrete product classes. This grows linearly. Adding a fifth pizza variety adds 2 classes; adding a third region adds 4 classes. Compare with Chapter 11's combinatorial explosion under inheritance-only design: this is the well-behaved linear case.

The Creator hierarchy

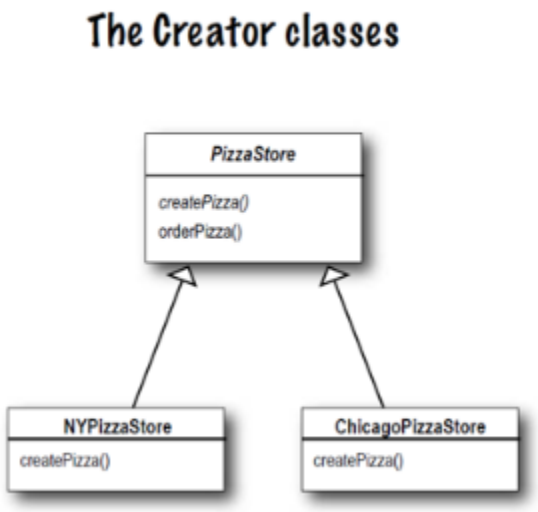

Figure 5.2: The Creator hierarchy. PizzaStore is the abstract creator; it defines an abstract factory method (createPizza) and a concrete template method (orderPizza). NYPizzaStore and ChicagoPizzaStore are concrete creators.

Also two layers, mirroring the product side:

- **PizzaStore (abstract).** The base class. Declares the factory method `createPizza` as *pure virtual*, it has no implementation here. Also defines the concrete method `orderPizza`, which calls `createPizza` and then runs the standard prepare/bake/cut/box routine. `orderPizza` is the *template method*; it sequences calls in a fixed order, and the only point of variation is which pizza it gets back from the factory method.
- **Concrete creators.** `NYPizzaStore` overrides `createPizza` to return New York-style pizzas. `ChicagoPizzaStore` overrides `createPizza` to return Chicago-style pizzas. Each franchise gets its own subclass.

Two notes are especially worth absorbing:

- **"Often the creator contains code that depends on an abstract product, which is produced by a subclass. The creator never really knows which concrete product was produced."** This is the structural payoff. `orderPizza` calls `createPizza` and returns a Pizza*. That's it. It never sees a `NYStyleCheesePizza` or a `ChicagoStyleClamPizza`. It is genuinely ignorant of the concrete type.
- **"Since each franchise gets its own subclass of PizzaStore, it's free to create its own style of pizza by implementing createPizza()."** This is the extensibility payoff. A franchise is free to invent new pizza types, Chicago might add a `ChicagoStyleStuffedCrustPizza`, without touching New York's code or the base `PizzaStore`.

Designing the Franchised Pizza Store

With the pattern in hand, let us map it onto the running pizza-shop example.

Step 1: Write the algorithm in terms of an abstract product

Look back at `orderPizza` from Section 1. We wrote it to take a `type` parameter and switch on it. The cleaned-up version keeps the type parameter and the algorithm but moves the construction out of the if-chain:

```
std::unique_ptr<Pizza> PizzaStore::orderPizza(std::string_view type) {
    auto pizza = createPizza(type);
    if (pizza) {
        pizza->prepare();
        pizza->bake();
        pizza->cut();
        pizza->box();
    }
    return pizza;
}
```

Three things have happened in this version:

- The construction line (`auto pizza = createPizza(type)`) calls a method *on this object* rather than instantiating a concrete class directly.
- The variable `pizza` has type `std::unique_ptr<Pizza>`, abstract, not concrete. The rest of the function works through this abstract type.
- `orderPizza` is now defined *entirely in terms of* the abstract Pizza interface. It compiles and works without any knowledge of a concrete pizza class.

Step 2: Declare createPizza as virtual on the base

Where does `createPizza` live? In the abstract `PizzaStore` base class, where `orderPizza` can call it. Its declaration is just:

```
virtual std::unique_ptr<Pizza> createPizza(std::string_view type) = 0;
```

Pure virtual. No implementation in the base. The base class is now abstract; you cannot instantiate `PizzaStore` directly. Every subclass must supply a `createPizza` override.

This is the structural heart of the pattern. `orderPizza` in the base class calls `createPizza` through virtual dispatch. The base class doesn't know which override will run. At runtime, when `orderPizza` is

called on (say) an `NYPizzaStore` instance, the `createPizza` call resolves to `NYPizzaStore::createPizza`, which produces NY-style pizzas.

Step 3: subclass per region

Each franchise gets its own subclass:

```
class NYPizzaStore : public PizzaStore {
public:
    std::unique_ptr<Pizza> createPizza(std::string_view type) override;
};

class ChicagoPizzaStore : public PizzaStore {
public:
    std::unique_ptr<Pizza> createPizza(std::string_view type) override;
};
```

Each override implements its own switch on `type`. `NYPizzaStore::createPizza("cheese")` returns a `NYStyleCheesePizza`. `ChicagoPizzaStore::createPizza("cheese")` returns a `ChicagoStyleCheesePizza`. Both return values are statically typed as `std::unique_ptr<Pizza>`, and `orderPizza` doesn't care which concrete type came back.

Adding a third region is trivial: write `SeattlePizzaStore : public PizzaStore`, implement its `createPizza`, done. Zero edits to any existing file. The OCP is honoured cleanly.

The runtime trace

Let us trace through what happens when `nyStore->orderPizza("cheese")` is called:

```
nyStore->orderPizza("cheese")
  └─ PizzaStore::orderPizza              // runs the inherited orderPizza
      ├─ createPizza("cheese")           // virtual dispatch, picks the override
      │   └─ NYPizzaStore::createPizza    // matched at runtime by dynamic type
      │       └─ new NYStyleCheesePizza   // construction; constructor sets fields
      │       └─< (pizza ready)
      │   └─< std::unique_ptr<Pizza>      // returned as abstract type
      ├─ pizza->prepare()                 // calls Pizza::prepare via dynamic dispatch
      ├─ pizza->bake()                    // calls Pizza::bake
      ├─ pizza->cut()                     // virtual, could resolve to override
      ├─ pizza->box()                     // calls Pizza::box
      └─< pizza                           // returned to client
```

Two things to notice. First, `createPizza` is the only point where the concrete class name appears in the trace, and even there, only inside `NYPizzaStore::createPizza`, which is in a single file. Second, `cut` calls *might* resolve to a virtual override (if Chicago wanted to cut squares instead of triangles, `ChicagoStyleCheesePizza` can override `cut`). Everywhere else, the code is identical across regions.

Implementing the Pizza Store in C++

Time to write the code. We build `Pizza` first (the abstract product), then the concrete pizzas, then `PizzaStore` (the abstract creator), then the concrete stores, then `main.cpp`. We use `std::unique_ptr` throughout for ownership and `std::string_view` for read-only string parameters.

The Pizza header

```
#ifndef PIZZA_H
#define PIZZA_H

#include <string>
#include <vector>
#include <memory>

class Pizza {
public:
    std::string name;
    std::string dough;
    std::string sauce;
    std::vector<std::string> toppings;

    void prepare();
    void bake();
    virtual void cut();
    void box();

    [[nodiscard]] std::string getName() const;
    virtual ~Pizza() = default;
};

#endif // PIZZA_H
```

Five things in this header are doing real work:

- **Public data members.** Name, dough, sauce, toppings. Each concrete pizza sets these in its constructor. Making them public is a debatable design call; a stricter version would use protected with explicit setters, but it matches the lecture deck's source and keeps the example readable.

- **Three non-virtual methods.** `prepare`, `bake`, and `box`; these are the same for every pizza. Each prints what it's doing using the data members the constructor filled in.
- **One virtual method.** `cut` is *virtual* because Chicago cuts pizzas into squares, whereas everywhere else cuts triangles. The base implementation cuts diagonal slices; `ChicagoStyleCheesePizza` will override it.
- **[[nodiscard]] on getName.** A C++17 attribute. It tells the compiler to warn if a caller ignores the return value of `getName`, e.g., writes `pizza->getName();` without using the result. Useful for getters and pure functions.
- **Virtual destructor defaulted.** `virtual ~Pizza() = default;` so that deletion through a `Pizza*` correctly destroys the concrete subclass. The compiler-generated destructor is fine; no cleanup work is needed in the abstract class.

> **Tricky**
>
> **Wait, Pizza isn't pure abstract.** Look again: every method on `Pizza` has an implementation (the cpp file). `Pizza` is not abstract; you could write `Pizza p; p.prepare();` and it would compile and run. Why is that OK? Because the *interesting* behaviour comes from the data members (name, dough, sauce, toppings) that the constructors fill in. A bare `Pizza` with default-empty data is harmless but useless.
>
> If you wanted to forbid the construction of bare `Pizza` objects, you could make one method pure virtual or make the constructor protected. We chose the simpler approach.

Pizza implementation

```
#include "pizza.h"
#include <iostream>

void Pizza::prepare() {
    std::cout << "Preparing " << name << '\n';
    std::cout << "Tossing dough..." << dough << '\n';
    std::cout << "Adding sauce..." << sauce << '\n';
    std::cout << "Adding toppings:\n";
    for (const auto& topping : toppings) {
        std::cout << "   " << topping << '\n';
    }
}

void Pizza::bake() {
    std::cout << "Bake for 25 minutes at 350\n";
```

```
}

void Pizza::cut() {
    std::cout << "Cutting the pizza into diagonal slices\n";
}

void Pizza::box() {
    std::cout << "Place pizza in official PizzaStore box\n";
}

std::string Pizza::getName() const {
    return name;
}
```

Four lines of output per pizza. `prepare` prints the name, dough, sauce, and each topping. `bake` prints "25 minutes at 350". `cut` prints "diagonal slices", but Chicago will override this. `box` prints the box message.

The whole `Pizza` class is a model of "keep the variations in data, share the logic in code". The logic, preparing, baking, cutting, and boxing is the same for everyone. The data, name, dough, sauce, and toppings are what make each pizza distinct.

NYStyleCheesePizza

Now, a concrete product. Header `nystyle.h`:

```
#ifndef NYSTYLE_H
#define NYSTYLE_H

#include "pizza.h"

class NYStyleCheesePizza : public Pizza {
public:
    NYStyleCheesePizza();
};

#endif // NYSTYLE_H
```

That's the whole header. The class adds nothing to the `Pizza` interface; it only customizes the constructor. Implementation:

```
#include "nystyle.h"

NYStyleCheesePizza::NYStyleCheesePizza() {
    name    = "NY Style Sauce and Cheese Pizza";
    dough   = "Thin Crust Dough";
    sauce   = "Marinara Sauce";
    toppings.emplace_back("Grated Reggiano Cheese");
}
```

Five lines of substance. The constructor sets the inherited data members. Note: use `toppings.emplace_back` rather than `push_back`, `emplace_back` constructs the string directly in the vector without an intermediate copy. It is the modern-C++ idiom for appending to containers when you have constructor arguments rather than an already-built object.

ChicagoStyleCheesePizza

Now the Chicago variant. Same product type, different region. Header `chicagostyle.h`:

```
#ifndef CHICAGOSTYLE_H
#define CHICAGOSTYLE_H

#include "pizza.h"

class ChicagoStyleCheesePizza : public Pizza {
public:
    ChicagoStyleCheesePizza();
    void cut() override;
};

#endif // CHICAGOSTYLE_H
```

Notice the difference from `NYStyleCheesePizza`: this one also declares `cut`, because Chicago cuts pizza into squares, not diagonals. The override comes for free thanks to the base class's `virtual void cut()` declaration. Implementation:

```
#include "chicagostyle.h"
#include <iostream>

ChicagoStyleCheesePizza::ChicagoStyleCheesePizza() {
    name    = "Chicago Style Deep Dish Cheese Pizza";
    dough   = "Extra Thick Crust Dough";
    sauce   = "Plum Tomato Sauce";
```

```
    toppings.emplace_back("Shredded Mozzarella Cheese");
}

void ChicagoStyleCheesePizza::cut() {
    std::cout << "Cutting the pizza into square slices\n";
}
```

The constructor fills in Chicago's data. `cut` overrides the base implementation to print "square slices". When `orderPizza` later calls `pizza->cut()`, virtual dispatch picks the Chicago version. The base `PizzaStore::orderPizza` doesn't notice; it just calls `cut`, and the right thing happens.

The other six concrete pizzas, `NYStylePepperoniPizza`, `NYStyleClamPizza`, `NYStyleVeggiePizza`, and their Chicago counterparts, follow the same template. The deck doesn't show them; we won't either. Each sets the four data members in its constructor, and the Chicago ones override `cut`.

PizzaStore: the abstract creator

Header `pizzastore.h`:

```
#ifndef PIZZASTORE_H
#define PIZZASTORE_H

#include <string_view>
#include <memory>
#include "pizza.h"

class PizzaStore {
public:
    // Factory Method - pure virtual function
    [[nodiscard]] virtual std::unique_ptr<Pizza>
        createPizza(std::string_view type) = 0;

    // Template method that uses the factory method
    [[nodiscard]] std::unique_ptr<Pizza>
        orderPizza(std::string_view type);

    virtual ~PizzaStore() = default;
};

#endif // PIZZASTORE_H
```

Notes:

- **createPizza is pure virtual.** The `= 0` makes `PizzaStore` itself abstract. You cannot write `PizzaStore s;`, only the subclasses can be instantiated.
- **orderPizza is concrete.** It is declared without `virtual`, which means it is a normal inherited method. The `PizzaStore` base class provides the full implementation; subclasses inherit it as-is. This is the Template Method Pattern: a concrete algorithm in the base class that calls an abstract step (the factory method).
- **std::string_view for the type parameter.** Read-only view into a string, no allocation. Standard modern C++ for short-lived string parameters that just need to be compared. The alternative, passing `const std::string&`, would force a `std::string` allocation if the caller passes a string literal.
- **unique_ptr return types.** The factory method and the template method both return `std::unique_ptr<Pizza>`. This makes ownership unambiguous: the caller of `orderPizza` gets a pizza they own, and the caller's pizza is destroyed when their `unique_ptr` is destroyed. No raw pointers, no manual deletes.

PizzaStore::orderPizza

The template method, in just 12 lines:

```
#include "pizzastore.h"
#include <iostream>

std::unique_ptr<Pizza> PizzaStore::orderPizza(std::string_view type) {
    auto pizza = createPizza(type);

    if (pizza) {
        pizza->prepare();
        pizza->bake();
        pizza->cut();
        pizza->box();
    }

    return pizza;
}
```

Look at line 5: `auto pizza = createPizza(type)`. This is the factory method call. The base class calls a function *on itself* via the implicit `this` pointer, and because `createPizza` is virtual, dispatch goes to the most-derived override. At runtime, when `orderPizza` is called on an `NYPizzaStore` instance, the `createPizza` line resolves to `NYPizzaStore::createPizza` and produces an NY-style pizza. The `PizzaStore` base never has to know which subclass is running.

Lines 7–12: the standard prepare/bake/cut/box routine. The `if (pizza)` check guards against the case where `createPizza` returned `nullptr`, for example, if the customer asked for a pizza type that doesn't exist in this region (e.g., Chicago asking for sushi). The function still returns a `unique_ptr<Pizza>`; the caller has to check it.

NYPizzaStore and ChicagoPizzaStore

The concrete stores. Header `nystore.h`:

```
#ifndef NYSTORE_H
#define NYSTORE_H

#include <string_view>
#include <memory>
#include "pizzastore.h"

class NYPizzaStore : public PizzaStore {
public:
    [[nodiscard]] std::unique_ptr<Pizza>
        createPizza(std::string_view type) override;
};

#endif // NYSTORE_H
```

The class adds nothing except the `createPizza` override. The keyword `override` is the modern C++ idiom; it tells the compiler, "this is supposed to be overriding a virtual function from the base class". If you misspell the function name or get the signature wrong (forget `string_view`, say), the compiler errors instead of silently introducing a new non-virtual member.

And implementation `nystore.cpp`:

```
#include "nystore.h"
#include "nystyle.h"

std::unique_ptr<Pizza> NYPizzaStore::createPizza(std::string_view type) {
    if (type == "cheese") {
        return std::make_unique<NYStyleCheesePizza>();
    }
    // Add other pizza types as needed
    return nullptr;
}
```

This is the place, and the only place, where the concrete class name `NYStyleCheesePizza` appears. `std::make_unique` constructs the pizza on the heap and wraps it in a `unique_ptr` in one call. If the type isn't recognized, the function returns `nullptr`, which `orderPizza` will handle with its guard check.

Chicago is exactly parallel, different switch body, different concrete class:

```
#include "chicagostore.h"
#include "chicagostyle.h"

std::unique_ptr<Pizza> ChicagoPizzaStore::createPizza(std::string_view
type) {
    if (type == "cheese") {
        return std::make_unique<ChicagoStyleCheesePizza>();
    }
    // Add other pizza types as needed
    return nullptr;
}
```

In real production code, the `if` chain would extend to handle the other three pizza types, pepperoni, clam, and veggie, for each store. We omit these for brevity. Notice that each store has its own `if` chain, which *does* mean some duplication if both stores support all four types. We will discuss this later, a known weakness of the basic Factory Method.

The driver

Finally `main.cpp`:

```
#include <iostream>
#include <memory>
#include "pizzastore.h"
#include "nystore.h"
#include "chicagostore.h"

int main() {
    // Create pizza stores using smart pointers
    auto nyStore      = std::make_unique<NYPizzaStore>();
    auto chicagoStore = std::make_unique<ChicagoPizzaStore>();

    // Order from NY store
    auto pizza = nyStore->orderPizza("cheese");
    std::cout << "Ethan ordered a " << pizza->getName() << "\n\n";

    // Order from Chicago store
```

```
    pizza = chicagoStore->orderPizza("cheese");
    std::cout << "Joel ordered a " << pizza->getName() << "\n\n";

    // Smart pointers automatically clean up - no delete needed!
    return 0;
}
```

Two stores constructed via `std::make_unique`. Two orders, each calling `orderPizza("cheese")`. Notice that the call sites for `nyStore` and `chicagoStore` are *identical*, same method, same argument. The difference in behaviour comes entirely from which concrete store is on the receiving end of the call. That is virtual dispatch doing the work the pattern promised.

Also notice `pizza = chicagoStore->orderPizza("cheese")`, assigns to `pizza` overwriting the previous unique_ptr's contents, which means the first (NY) pizza is destroyed at that point. No memory leak; no manual delete. The unique_ptr machinery handles it.

Caveats and Modern C++ Refinements

Factory Method is one of the cleaner patterns to implement in C++, but it has a few practical concerns. Let us cover them.

Why std::unique_ptr is the right return type

- **Ownership is explicit.** A `unique_ptr` return value tells the caller: "this is yours; you own it; deleting it is automatic when your unique_ptr dies". Compared to returning a raw `Pizza*`, you can't tell from the signature whether the caller is supposed to delete it. The discipline is only in the documentation.
- **No leaks if the caller forgets.** With raw pointers, `auto* p = nyStore->orderPizza("cheese")` without a matching `delete` is a leak. With unique_ptr, the leak is impossible; when p goes out of scope, the pizza is freed.
- **Exception safety.** If anything between construction and use throws, unique_ptr ensures the partially constructed pizza is destroyed. Raw pointers would leak in this case.

Other smart-pointer choices have their place. `std::shared_ptr<Pizza>` is appropriate when the pizza is shared by multiple owners, say, two threads watching the same order. `std::weak_ptr<Pizza>` is rarely useful as a return type. The default and right answer is `unique_ptr`.

Why std::string_view for the type parameter

The factory method takes a `std::string_view` rather than a `std::string`. This is also deliberate.

- **No allocation.** A `string_view` is a pointer-plus-length, not a heap-allocated string. Passing `"cheese"` as a parameter does not allocate.
- **Accepts everything.** A `string_view` can be constructed from a `std::string`, a `char*`, a string literal, or another `string_view`. The caller has freedom.
- **Lifetime caveat.** A `string_view` does not own the data it points to. You *must not* store one as a member and use it after the original string has been destroyed. The factory method's pattern of "use it once during the function body and discard" is exactly the use case `string_view` was designed for.

enums beat strings for the product type

The lecture's design uses strings (`"cheese"`, `"pepperoni"`) to select the product. This is easy to read and easy to extend, but it sacrifices compile-time safety. A typo, `"chees"`, turns into a runtime `nullptr` instead of a compile error.

A stricter design uses an enum class:

```
enum class PizzaType { Cheese, Pepperoni, Clam, Veggie };

[[nodiscard]] virtual std::unique_ptr<Pizza>
    createPizza(PizzaType type) = 0;
```

Now, `createPizza(PizzaType::Cheese)` is the only way to ask for cheese. Typos are compile errors. The trade-off: enums are harder to extend in a header-only environment, and they make string-based configuration ("read pizza type from command line") slightly less direct (you need a string→enum lookup). For production code, the safety usually wins; for lecture code, the simplicity of strings is more readable.

Duplicate switch chains across concrete creators

The most visible weakness of basic Factory Method is that each concrete creator carries its own switch on the product type. `NYPizzaStore::createPizza` switches on the type and returns one of four NY pizzas. `ChicagoPizzaStore::createPizza` switches on the type and returns one of four Chicago pizzas. The *switch structure* is duplicated.

Two ways to attack this duplication:

- **Registry-based factory.** Instead of an `if` chain, hold a `std::unordered_map<std::string, std::function<std::unique_ptr<Pizza>()>>` inside each store. Map keys are pizza-type

strings; values are factory lambdas. To create a pizza of a given type, look up the map. Adding a new pizza is just inserting a new map entry, no if-chain to extend.

- **Template-based factory method.** If your product types are known at compile time, you can templatize: `template<typename PizzaT> std::unique_ptr<Pizza> create();`. This eliminates the switch entirely at the cost of forcing the choice of pizza type to be known at compile time, a poor fit if the type comes from user input.

The lecture's basic version with an `if` chain is the pedagogical sweet spot, clearer than either alternative, with a small enough surface area that the duplication is acceptable. In production code with twelve regions and forty pizza types, you would reach for the registry version.

The relationship between factories

A subtle question: should `NYPizzaStore` and `ChicagoPizzaStore` share any code? Today, no, each just has its own switch. But there's room for genuine sharing: both stores might log every order, both might enforce a maximum number of pizzas per order, both might compute a regional sales-tax adjustment.

If you find such shared logic, the right home for it is `PizzaStore::orderPizza` itself. The template method can incorporate any per-store-but-uniform logic before or after the factory method call. The whole point of Template Method (which lives inside the Factory Method Pattern) is to centralize this logic.

If shared logic varies between regions in a small way, the standard escape valves are: (a) more virtual methods on the base, each defaulted but overridable, and (b) protected helper methods that subclasses can call.

Don't reach for factories when a constructor would do

Factory Method is a good answer when you have multiple subclasses and the choice between them is dynamic. It is overkill when there is one product class, one creator, and no axis of variation.

In particular, do not introduce a factory method to "hide" a constructor that has a complicated argument list. The Builder pattern (which you may meet later in the course) is the right tool for that, not Factory Method.

Abstract Factory: One Step Beyond

Factory Method handles the case where "a class needs to create one kind of object, and subclasses pick which kind". Abstract Factory handles a richer problem: "a class needs to create a *whole family* of related objects, and a single decision picks the entire family".

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

Three phrases matter:

- **"Families."** Multiple kinds of objects, not just one. Buttons, scroll bars, *and* menus. Bread, pasta, *and* sauce. Ingredients of many kinds, all needing to be consistent with each other.
- **"Related."** The objects belong together. NY-style dough and NY-style sauce belong on an NY pizza; Chicago-style dough and Chicago-style sauce belong on a Chicago pizza. Mixing them is wrong.
- **"Without specifying concrete classes."** The client picks a factory, and the factory picks all the concrete classes. The client never says `new NYStyleDough`; it says `nyIngredients->createDough()` and gets one back without knowing or caring which concrete class implements it.

The motivating story

Your pizza shop has been franchised, and each franchise has different ingredient suppliers. New York's red sauce is not Chicago's red sauce; they come from different vendors, have different consistencies, and different prices. You need to ship the right ingredients to each franchise.

Without the pattern, every concrete pizza would either hard-code its ingredient brand ("NY Cheese Pizza uses Brand-A marinara") or include a region switch ("if NY then Brand-A else Brand-B"). Both options scatter the regional decision throughout the pizza classes. The maintenance nightmare is obvious: change vendors in one region and you touch every pizza class.

Abstract Factory's answer: an `IngredientFactory` interface with one method per ingredient type, `createDough`, `createSauce`, `createCheese`, and so on. Two concrete factories: `NYIngredientFactory` and `ChicagoIngredientFactory`, each producing the regional variant of every ingredient. The pizza classes hold an `IngredientFactory*` and ask it for ingredients; they never know which region they're in.

The canonical UML

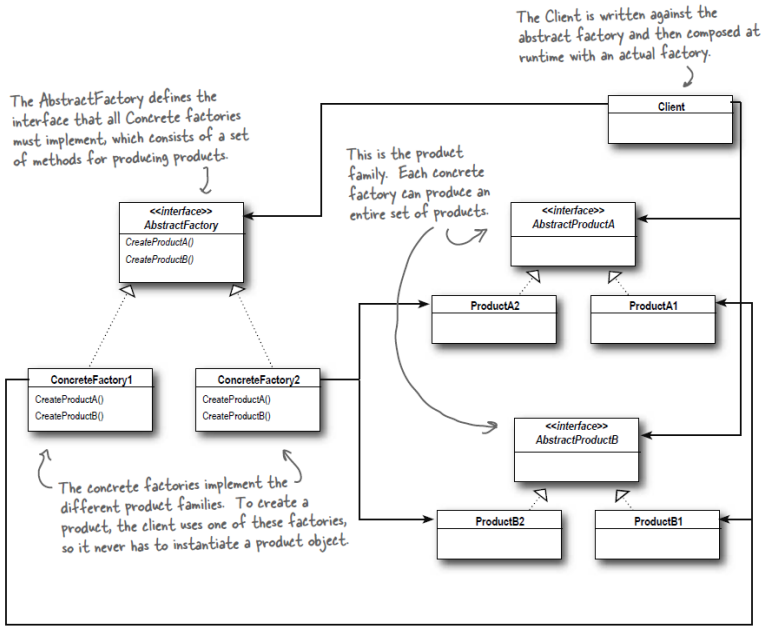

Figure 9.1: Abstract Factory. The AbstractFactory interface declares one creation method per product type. ConcreteFactory1 and ConcreteFactory2 each implement those methods to produce a coherent family of products. The Client depends only on the AbstractFactory interface and the abstract product interfaces.

- **AbstractFactory**, the interface. Declares `createProductA()`, `createProductB()`, etc. Each method returns an abstract product type.
- **ConcreteFactory1, ConcreteFactory2**, the family implementations. Each one implements every create-method on AbstractFactory, returning products from *its* family. ConcreteFactory1 returns ProductA1 and ProductB1; ConcreteFactory2 returns ProductA2 and ProductB2.
- **AbstractProductA, AbstractProductB**, the product interfaces. The client interacts with these.
- **Client**, written against the abstract factory and the abstract products. At construction time, it is handed a concrete factory; from that point on, it has no idea which family it is using.

The key insight, embodied in the annotation: "The Client is written against the abstract factory and then composed at runtime with an actual factory." That last clause, "composed at runtime", is what gives Abstract Factory its power. The decision of which family to use can come from a configuration file, a command-line argument, a UI selection, or anything.

Factory Method vs. Abstract Factory

Students routinely conflate the two. Here is the head-to-head comparison:

Aspect	Factory Method	Abstract Factory
How many products?	One product type	A family (multiple product types)
Mechanism	Inheritance, subclass overrides method	Composition, the client holds a factory object
Where the choice lives	Which subclass is instantiated	Which factory object is held
Number of methods	One factory method per Creator	One method per product type
When to use	One varying axis ("which class")	Multiple varying axes producing related objects
Pizza-shop analogue	Each region makes its own pizzas	Each region uses its own ingredient suppliers

Notes:

- **"Factory Method relies on inheritance: object creation is delegated to subclasses, which implement the factory method to create objects."**
- **"Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface."**

The two patterns often appear together. A real pizza franchise system might use Factory Method to pick a `PizzaStore` subclass per region, and an Abstract Factory within each pizza to pick the regional ingredient family. They are complementary, not redundant.

Factories in the Wild

Factories are everywhere, possibly more so than any other creational pattern.

Logistics and transport

Consider a logistics company. A base `LogisticsCompany` class has a factory method `createTransport()`. A `RoadLogistics` subclass implements it to return `Truck` objects; `SeaLogistics` returns `Ship` objects. The main application code works with the abstract `Transport` interface and never knows whether it is dispatching trucks or ships.

This pattern shows up in any routing/dispatch system: ride-sharing apps with cars vs. bikes vs. scooters; food-delivery platforms with couriers vs. drones; shipping providers that route automatically between truck, rail, sea, and air.

Cross-platform UI

Java Swing/AWT, the older Windows Forms framework, and most cross-platform GUI toolkits use Factory Method extensively. Different operating systems need different button implementations. Windows draws a native Windows button, macOS draws a Cocoa button, and Linux draws a Qt or GTK button. The framework provides an abstract `Button` class, and the platform-specific toolkit overrides `createButton` to produce the right kind. Your application code calls `createButton` and gets a working native widget without writing platform-specific code.

This is exactly an Abstract Factory in the GoF sense: the toolkit factory produces a *family* of native widgets (button, text field, scroll bar, menu, ...) that need to be visually consistent. Every UI framework that wants to feel native on multiple operating systems uses this pattern.

Document and file-format selection

Microsoft Office and similar suites use the Factory Method when you click "New Document." The application creates different document types, Word doc, Excel spreadsheet, PowerPoint presentation, based on which menu the user clicked or which application is running. Each document type is a concrete product, and the host application is the concrete creator.

LibreOffice, Adobe Creative Suite, and the older Borland/Inprise toolset all have similar factory-shaped "New X" flows. The chapter's practice problem at the end of this chapter is a slimmed-down version of exactly this: a document export tool that selects an output format at runtime.

Payment gateways

Payment processing libraries like Stripe, Braintree, and Adyen use factories to construct payment-method handlers. The client says "the user picked credit card" or "the user picked digital wallet," and the library returns a `PaymentMethod` object that knows how to talk to the underlying gateway. The set of supported methods grows over time, new providers add Apple Pay, then Google Pay, then crypto, then BNPL, and the factory pattern lets the client code stay the same.

Database drivers and ORMs

Almost every database access library uses factories. The Java JDBC `DriverManager.getConnection` call is a factory method that returns a `Connection`, implemented differently by MySQL, Postgres, Oracle, SQLite drivers. Python's DB-API is the same. Object-relational mappers (ORM) like SQLAlchemy and Hibernate use factories to construct query objects and session handles.

Logging frameworks

Logging libraries like log4j, Serilog, and Python's `logging` module use factories for logger objects. `Logger::get("com.app.module")` is a factory call that returns either a freshly-constructed logger or a cached one, but the caller can't tell which. This is the Factory Method Pattern combined with the Singleton pattern (which you may meet later).

The pattern's footprint

Once you have the Factory Method shape internalized, "a virtual method on an abstract base, overridden in subclasses to construct concrete products", you will spot it constantly. Every time an application has to support a *pluggable* subsystem (database, file format, protocol, codec, payment method, UI toolkit), the chances are excellent that the pluggability is implemented with a factory. That is why the GoF book describes Factory Method as "perhaps the most used" pattern. Quietly, it is the structural glue holding most extensible software together.

Summary

Factories isolate *object construction* from the rest of the application, so the rest of the application can work entirely through abstractions. This chapter covered three related variants:

- **Simple Factory** (not a formal GoF pattern). A class with one method that switches on a type tag and returns the right concrete product. Decouples clients from concrete product classes, but the factory itself still has to be modified when products are added.
- **Factory Method**. An abstract method on a Creator base class that subclasses override to produce specific concrete products. The base class is closed for modification; new product types are added via new Creator subclasses.
- **Abstract Factory**. An interface with multiple creation methods that together produce a *family* of related products. Implemented by composition rather than inheritance.

All three are built around the same Dependency Inversion principle: *depend upon abstractions, not concrete classes*.

Things to remember when you implement Factory Method in C++:

- Return `std::unique_ptr<Product>`, never a raw pointer. Ownership becomes explicit and leaks become impossible.
- Mark the factory method `virtual` (and usually `= 0`) so subclasses can override it.
- Use `override` on every override so the compiler catches mistakes.
- Use `std::string_view` or `enum class` for type tags; prefer enums for stricter type safety where extensibility is not an issue.
- Use `std::make_unique<Concrete>(...)` inside the factory method body.
- The base class's template method calls the factory method through `this`, relying on dynamic dispatch.

Things to remember about the pattern's role in design:

- Factories are the canonical structural answer to the Open–Closed Principle for object construction.
- Factory Method and Abstract Factory often appear together, one selects the family, the other selects the members.
- Concrete class names should appear in exactly one place: the factory. Everywhere else, code against abstractions.
- Don't reach for factories when there is no axis of variation. They are most valuable when the set of products is open-ended or chosen at runtime.

Practice Problems

Problem 1.

For each system below, identify the abstract Creator, at least two concrete Creators, the abstract Product, and at least two concrete Products. If the system uses Abstract Factory rather than Factory Method, identify which products belong to the same family.

(a) A document-export tool that can output PDF, HTML, or plain text based on a command-line flag.

(b) A web framework's request handler. The framework supports JSON, XML, and form-encoded request bodies; the parser is chosen by the Content-Type header.

(c) A cross-platform GUI library that supports Windows, macOS, and Linux. The library provides Buttons, TextFields, and Sliders, each rendered natively on the host OS.

(d) A multiplayer game with three character classes (Warrior, Mage, Archer), each with their own weapon types (Sword, Staff, Bow), armor sets, and special abilities.

(e) A database connection pool that supports MySQL, Postgres, and SQLite. The user supplies a connection string; the library detects the database type and returns the right kind of connection object.

Problem 2.

Suppose the pizza-shop code from Section 7 has been extended so that `ChicagoPizzaStore::createPizza` supports both "cheese" and "pepperoni". Trace what happens for the following snippet:

```
auto chicagoStore = std::make_unique<ChicagoPizzaStore>();
auto pizza = chicagoStore->orderPizza("pepperoni");
std::cout << pizza->getName() << '\n';
```

(a) List, in order, every method call that occurs between `orderPizza("pepperoni")` being invoked and control returning to main. Be explicit about which class's version of each method runs.

(b) At the line `auto pizza = createPizza(type)` inside `PizzaStore::orderPizza`, explain how the C++ runtime decides *which* `createPizza` implementation to call.

(c) Suppose someone removes the `virtual` keyword from `createPizza` in `pizzastore.h`. The code still compiles. What goes wrong at runtime, and why?

(d) Suppose instead they remove `override` from `ChicagoPizzaStore::createPizza`'s signature in the header. Does anything change at runtime? What is the value of keeping the `override` keyword anyway?

Problem 3.

For each scenario, decide whether Simple Factory or full Factory Method is the right tool, and justify in two or three sentences.

(a) A graphics editor needs to instantiate one of three shape classes (Circle, Square, Triangle) based on a button the user clicks. The set of shapes is fixed and will not grow.

(b) A compiler needs to construct AST nodes for any of dozens of syntactic constructs. Adding a new construct is a routine extension that happens many times per year.

(c) A test framework needs to construct a fresh "fixture" object for every test method. The framework is shipped as a library; user test files define their own fixture subclasses.

(d) An online shop has exactly four payment methods and the law requires explicit code-review when adding a fifth. Payment processing varies per method.

(e) Identify which one of the five answers above will most likely *change* over a five-year horizon. Does your factory choice still hold for the longer time scale?

Problem 4.

For each of the following code snippets, decide whether it respects the Dependency Inversion principle. If not, identify the violation and rewrite the snippet to fix it.

(a)

```
void handleOrder(NYPizzaStore* store) {
    auto pizza = store->orderPizza("cheese");
    pizza->box();
}
```

(b)

```
std::vector<CheesePizza> cart;
cart.push_back(CheesePizza{});
```

(c)

```
class OrderTracker {
    SimplePizzaFactory factory;
public:
    void recordOrder(std::string_view type) {
        auto pizza = factory.createPizza(type);
        log << pizza->getName();
    }
};
```

(d)

```
Pizza* createPizza(std::string_view type) {
    if (type == "cheese") return new CheesePizza{};
    if (type == "pepperoni") return new PepperoniPizza{};
    return nullptr;
}
```

(e) Of the four snippets above, which one is most defensibly OK as-is? Argue for the most permissive reading of DIP.

Problem 5.

Starting from the pizza-shop code in Section 7, add a new franchise: `SeattlePizzaStore`, which sells "cheese" pizzas in a regional Seattle style.

(a) Write the new concrete product class `SeattleStyleCheesePizza`, with appropriate dough, sauce, name, and topping(s) of your choice. Override `cut` to print whatever cutting style Seattle uses (be creative).

(b) Write the new concrete creator class `SeattlePizzaStore`, with a `createPizza` override that produces SeattleStyleCheesePizza when asked for "cheese".

(c) Modify `main.cpp` to also order a cheese pizza from the Seattle store.

(d) List every other file in the existing codebase that you had to edit. The list should be empty, that's the point.

(e) Predict the exact console output the new section of main will produce. Don't compile it first.

Problem 6.

Section 8.4 mentioned that the if-chain inside each `createPizza` can be replaced with a registry-based factory using `std::unordered_map<std::string, std::function<std::unique_ptr<Pizza>()>>`.

(a) Rewrite `NYPizzaStore` so it holds a map from pizza-type strings to factory lambdas. Register entries in the constructor for at least "cheese" and "pepperoni". The `createPizza` method should look up the type in the map and call the lambda.

(b) What does the new `createPizza` do when an unknown type is passed? Compare the runtime behaviour to the if-chain version.

(c) Suppose you want to add a *third* pizza type to an existing `NYPizzaStore` *after* it has been constructed. With the if-chain version, this is impossible without editing the source. With the registry version, sketch the public method you would add to `NYPizzaStore` to allow runtime registration of new pizza types.

(d) Compare the registry-based factory and the if-chain factory along three axes: readability, runtime cost per createPizza call, and extensibility. State which axis you would weight most heavily for the pizza-shop and why.

Problem 7.

Implement the Abstract Factory pattern described in Section 9 for pizza *ingredients*. (This is the deck's own task.)

(a) Sketch a UML class diagram with: an `IngredientFactory` abstract interface; `NYIngredientFactory` and `ChicagoIngredientFactory` as concrete factories; and at least three product hierarchies, Dough, `Sauce`, and `Cheese`, each with NY and Chicago concrete variants.

(b) Write the C++ headers for the four interfaces (IngredientFactory, Dough, Sauce, Cheese) and at least one NY concrete variant of each ingredient.

(c) Modify the existing `NYStyleCheesePizza` to hold an `IngredientFactory*` and use it in the constructor instead of hard-coding strings. Show the new constructor body.

(d) Discuss: With this design in place, how would a customer mistakenly cause an NY pizza to be made with Chicago ingredients? How does the design make such a mistake hard?

Problem 8.

Decorator (Chapter 12) and Factory Method (this chapter) play very well together. Consider a Starbuzz-style coffee shop where each region has its own coffee bases (`Espresso`, `HouseBlend`, `DarkRoast`) and condiment lineup (regional milks, sweeteners, and toppings).

(a) Sketch a design that uses Factory Method to choose between an `NYCoffeeShop` and a `SeattleCoffeeShop`, and inside each shop uses Decorator (from Chapter 12) to assemble the condiment chain on top of the base coffee.

(b) Where does the inheritance live in your design? Where does the composition live?

(c) Where does each variation axis (region, coffee type, condiment type) get its own pattern? Justify why your design hands each one to the right pattern.

(d) How would the design change if condiments were a *family* of related ingredients (regional milk + regional sweetener + regional topping) that always travel together? Which pattern would you reach for instead of Decorator?

Problem 9.

Design a cross-platform UI library that supports three operating systems, Windows, macOS, and Linux, and three widget types, Button, TextField, and `Slider`. Each widget on each OS draws differently natively; the application code should be able to construct widgets without caring about the OS.

(a) Decide: Factory Method or Abstract Factory? Justify the choice in two or three sentences.

(b) Sketch a UML class diagram showing all interfaces and concrete classes. Use the role labels (Creator, ConcreteCreator, Product, ConcreteProduct, etc.) where applicable.

(c) Write the C++ headers for: the abstract widget interfaces; one full concrete OS implementation (Windows, including all three widgets); and the abstract factory interface plus the Windows concrete factory.

(d) Show how a client function `buildLoginScreen` takes an abstract factory parameter and produces a screen with one of each widget type, with no OS-specific knowledge.

(e) Sketch how main() would detect the running OS (assume `getCurrentOS()` is given) and construct the right concrete factory.

(f) Stretch: discuss the testability of your design. If you needed to unit-test `buildLoginScreen` without spawning real native widgets, what would you do? Hint: a *Mock* implementation of the abstract factory.

Problem 10. ★

Program behaviour:

Your executable must be named `exporter` and run like this:

```
./exporter pdf  "CS247 notes"
./exporter html "Hello world"
./exporter txt  "Week 3 summary"
```

The first argument is an output format (`pdf`, `html`, or `txt`). The second argument is the document content (assume quoted if it contains spaces). The program creates an appropriate "export pipeline" at runtime and prints the exported result to standard output.

Required design (Factory Method focus).

Implement a Creator hierarchy and a Product hierarchy.

Product interface. Create an abstract base class `Document` with a virtual destructor and a pure virtual method `std::string render() const`. Then implement concrete products `PdfDocument`, `HtmlDocument`, and `TextDocument`. Each concrete class must store the original content and return a distinct formatted output in `render()`. Keep the formatting simple, PDF adds a fake header/footer string; HTML wraps in `<html><body>...</body></html>`; Text returns the content unchanged.

Creator (Factory Method) interface. Create an abstract base class `Exporter` (the Creator) that provides a public method `exportDoc` performing the overall export process, and a protected pure virtual factory method `createDocument`. The key requirement is that `Exporter` must not know which concrete `Document` it gets; it only uses the `Document` interface.

Concrete Creators. Implement `PdfExporter` that overrides `createDocument` to return a `PdfDocument`; `HtmlExporter` that overrides it to return an `HtmlDocument`; and `TextExporter` that overrides it to return a `TextDocument`.

Runtime selection requirement. `main()` must choose which `Exporter` subclass to instantiate based on the first command-line argument. After that, `main()` must call the *same* interface (`exportDoc`) regardless of which exporter was chosen. `main()` must *not* directly instantiate any `Document` classes, only `Exporter` subclasses may create `Document` objects via the factory method.

(a) Sketch the UML class diagram for your design. Use the standard role labels (Product, ConcreteProduct, Creator, ConcreteCreator).

(b) Implement the full program. Use `std::unique_ptr` throughout. Show all headers, all .cpp files, and main.cpp.

(c) Verify your implementation produces something like (lay out the exact output yourself):

```
$ ./exporter pdf "CS247 notes"
```

```
=== PDF Document ===
CS247 notes
=== End PDF ===

$ ./exporter html "Hello world"
<html><body>Hello world</body></html>

$ ./exporter txt "Week 3 summary"
Week 3 summary
```

(d) Adding `markdown` as a fourth output format should require touching exactly two new files (a `MarkdownDocument` and a `MarkdownExporter`) plus a small change in `main` to recognise the new format string. List the exact changes.

(e) Reflect: how would your program look *without* the Factory Method Pattern, perhaps using one massive `if/else` chain that handles every (format × content) combination? What advantages does the pattern offer when you go from three formats to twenty?