

Bienvenue au cours “Design Pattern-1”

Nom de l’enseignant : Fabrice Meynckens

Promo : 2025-2026

Classe : B2CDEV



Objectifs de la session

Les Fondations et l'Art de la Construction

- Pourquoi les Design Patterns ?
- SOLID
- Anti-Patterns

Les Patterns de Création

- Singleton
- Factory Method
- Builder

Les Patterns de Structure (Partie 1)

- Adapter

Design Patterns

Bienvenue dans ce module consacré aux **Design Patterns** (ou Patrons de Conception). Puisque vous maîtrisez déjà les bases du C++, la programmation orientée objet, le polymorphisme et la surcharge... , nous n'allons pas perdre de temps sur la syntaxe. Nous allons nous concentrer sur l'**architecture** et l'**élégance** du code.

L'objectif de ces trois jours est simple : **transformer votre façon de penser**.

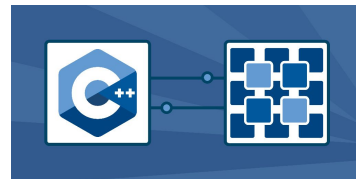
Vous ne coderez plus seulement pour que "ça marche", mais pour que votre code soit **robuste**, **maintenable** et **évolutif**.

Pour commencer ce cours, nous allons poser les valises...

Avant d'écrire la moindre ligne de code, **il faut changer votre état d'esprit**.

Jusqu'ici, en C++, vous avez appris à utiliser le marteau, la scie et le tournevis (syntaxe, boucles, classes).

Aujourd'hui, nous allons apprendre à lire et dessiner des plans d'architecte.



Qu'est-ce qu'un Design Pattern ?

Ne confondez pas "Design Pattern" et "Algorithme".

- **Un Algorithme** est une recette de cuisine précise pour résoudre un problème de calcul (ex: trier une liste, trouver le plus court chemin). C'est du "pas à pas".
- **Un Design Pattern (Patron de Conception)** est une description de haut niveau d'une solution à un problème de conception récurrent. C'est un schéma, un plan.

Analogie : Si vous construisez une maison, l'algorithme serait la méthode exacte pour mélanger le ciment. Le Design Pattern serait le plan standardisé pour installer le circuit électrique ou la plomberie afin qu'ils soient accessibles pour les réparations futures.

Les 4 éléments essentiels d'un Pattern :

1. **Le Nom :** Pour en parler rapidement.
2. **Le Problème :** Dans quel contexte cela s'applique-t-il ?
3. **La Solution :** Les classes, les interfaces et leurs relations (souvent via un diagramme UML).
4. **Les Conséquences :** Les avantages (ex: flexibilité) et les inconvénients (ex: complexité accrue).

Pourquoi s'embêter avec ça ?

Vous pourriez dire : *"Mais je peux coder ça avec trois `if` et une variable globale, ça marche !"*.

Oui, ça marche... aujourd'hui. Mais dans 6 mois ?

Voici pourquoi les Design Patterns sont vitaux pour votre carrière :

- **1. Un Vocabulaire Commun (Le plus important)**

- Imaginez que vous deviez expliquer à votre collègue : *"Alors ici, j'ai créé une classe qui notifie une liste d'autres classes quand sa variable change..."* C'est long.
- Avec les patterns, vous dites : *"J'ai utilisé un **Observer** ici."*
- **Boom !**

En 3 mots, votre collègue a visualisé la structure, les classes, les connexions et les implications mémoire. C'est un gain de temps colossal.

- **2. Ne pas réinventer la roue**

- Les problèmes que vous rencontrez (comment créer cet objet complexe ? comment découpler l'interface de l'implémentation ?) ont déjà été résolus par des milliers d'ingénieurs avant vous. Profitez de leur expérience.

- **3. Architecture et Maintenabilité**

- Les patterns forcent souvent le **découplage**. En C++, cela signifie moins de dépendances entre vos fichiers d'en-tête (`.h`), une compilation plus rapide, et surtout, la possibilité de changer une brique sans casser tout le château.

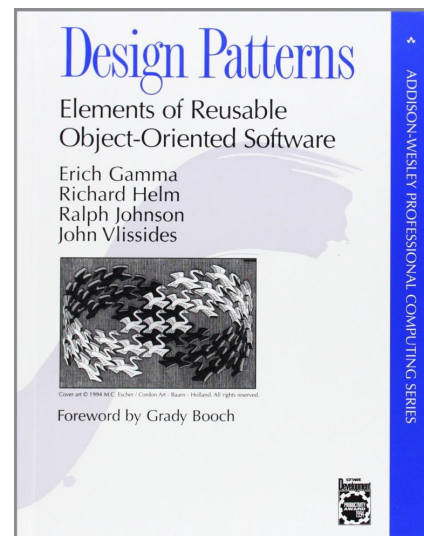
L'Histoire : Le "Gang of Four" (GoF)

L'origine du concept vient de l'architecture de bâtiments (Christopher Alexander, 1977), mais c'est en **1994** que tout a basculé pour l'informatique.

Quatre auteurs, **Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides**, ont publié le livre : *"Design Patterns: Elements of Reusable Object-Oriented Software"*.

Ce livre a catalogué 23 patterns fondamentaux. On appelle ces auteurs le **Gang of Four (GoF)**.

Même si le C++ a évolué (C++11/14/20), ces 23 patterns restent la référence absolue.



La Taxonomie : Les 3 Familles

Pour s'y retrouver, nous classons les patterns en trois catégories distinctes selon leur but.

A. Les Patterns de Création (Creational)

- **Le problème** : L'opérateur `new` en C++ est rigide. Si vous écrivez `new MaClasse()`, votre code est soudé à `MaClasse` pour toujours.
- **Le but** : Abstraire le processus d'instanciation. Rendre le système indépendant de la manière dont ses objets sont créés.
- *Exemples* : Singleton, Factory, Builder.

B. Les Patterns de Structure (Structural)

- **Le problème** : Comment faire travailler ensemble des classes qui n'ont pas été prévues pour ça ? Comment assembler des objets simples pour créer des structures complexes ?
- **Le but** : Utiliser l'héritage et l'aggrégation pour composer des interfaces et définir la structure des objets.
- *Exemples* : Adapter, Decorator, Facade.

C. Les Patterns de Comportement (Behavioral)

- **Le problème** : Qui fait quoi ? Comment les objets communiquent-ils sans être trop liés les uns aux autres ?
- **Le but** : Se concentrer sur les algorithmes et l'assignation des responsabilités entre objets.
- *Exemples* : Observer, Strategy, Command.

Problème racine

Avant de plonger dans le catalogue des patterns, il est crucial de comprendre **le problème racine** que les patterns de création cherchent à résoudre :

– le couplage fort induit par l'opérateur `new`. –

En C++, quand vous écrivez `new MaClasse()`, vous soudez votre code à `MaClasse`.

Si vous voulez changer d'implémentation plus tard, vous devez modifier le code, recompiler et re-tester.

Pour la suite du cours, vous pouvez télécharger le fichiers "Design-Pattern-1.zip" disponible sur GDrive à l'adresse suivante :

<https://drive.google.com/drive/folders/1LU0w7KuCH9INQam87Ss23CmLSNOMBFjL>

Mauvais code : Couplage fort

Imaginez une classe `Game` qui doit intégrer (en plus de la mécanique du jeu) la sauvegarde de la partie.

Ici, le développeur a décidé de sauvegarder dans un fichier texte.

Si demain on veut sauvegarder dans le Cloud ou en Base de Données (SQL), on est obligé de modifier la classe `Game` et de casser le code existant.

La classe `Game` doit se focaliser sur les mécaniques du jeu, il n'est pas bon de devoir la modifier si les conditions de sauvegarde changent.

Aujourd'hui, on a peut être uniquement besoin de sauvegarder sur fichier, mais il faut penser à l'avenir, à l'évolution des besoins et des techniques

(fichier : `general/bad-code.cpp`)

```
// --- MAUVAIS CODE ---

#include <string>
#include <iostream>

class TextFileSaver {
public:
    void save(const std::string& data) {
        std::cout << "Sauvegarde dans un fichier texte : " << data << std::endl;
    }
};

class Game {
public:
    void saveGame() {
        // PROBLÈME ICI :
        // La classe Game crée elle-même sa dépendance.
        // Elle est "mariée" de force avec TextFileSaver.
        TextFileSaver* saver = new TextFileSaver();
        saver->save("PlayerLvl:10");
        delete saver;
    }
};

// Si demain on veut sauvegarder dans le Cloud ou en Base de Données (SQL),
// on est obligé de modifier la classe Game et de casser le code existant.
```

Le "Bon" Code : Découplage via Interface

Pour corriger cela, on applique le principe d'**Inversion de Dépendance** (le "D" de SOLID). La classe `Game` ne doit pas dépendre d'une implémentation concrète (`TextFileSaver`), mais d'une abstraction (`ISaver`).

Rappel : destructeur virtuel par défaut

```
virtual ~ISaver() = default;
```

1. virtual

Rend le destructeur polymorphe

Permet la destruction correcte des objets via un pointeur vers la classe de base

Essentiel quand on utilise l'héritage avec des pointeurs de base

2. ~ISaver()

C'est le destructeur de la classe `ISaver`

Appelé automatiquement quand un objet est détruit

3. = default

Demande au compilateur de générer automatiquement le destructeur par défaut

Plus moderne et explicite que d'écrire {}

```
// PROBLÈME sans virtual :
```

```
ISaver* saver = new CloudSaver();  
delete saver; // ❌ Appelle seulement ~ISaver(), pas ~CloudSaver()
```

```
// SOLUTION avec virtual :
```

```
ISaver* saver = new CloudSaver();  
delete saver; // ✅ Appelle d'abord ~CloudSaver(), puis ~ISaver()
```

```
// 1. On définit un contrat (Interface)  
class ISaver {  
public:  
    virtual void save(const std::string& data) = 0;  
    virtual ~ISaver() = default;  
};  
  
// 2. Les implémentations respectent le contrat  
class TextFileSaver : public ISaver {  
public:  
    void save(const std::string& data) override {  
        std::cout << "Sauvegarde Fichier : " << data << std::endl;  
    }  
};  
  
class CloudSaver : public ISaver {  
public:  
    void save(const std::string& data) override {  
        std::cout << "Envoi vers le Cloud : " << data << std::endl;  
    }  
};  
  
// 3. La classe Game ne connaît que l'interface  
class Game {  
    ISaver* saver; // Pointeur vers l'abstraction  
public:  
    // On "injecte" la dépendance à la construction  
    Game(ISaver* s) : saver(s) {}  
  
    void saveGame() {  
        // Game ne sait pas (et ne veut pas savoir) si c'est du fichier ou du cloud.  
        saver->save("PlayerLvl:10");  
    }  
};
```

Pourquoi est-ce mieux ? Les patterns de création (Factory, Builder, Singleton) servent justement à gérer cette étape de création (`new CloudSaver()`) de manière intelligente, pour que votre code reste aussi propre que l'exemple 2.
(fichier : general/good-code.cpp)

SOLID

SOLID est un peu la "Constitution" du développement orienté objet. Les Design Patterns ne sont, finalement, que des solutions techniques qui permettent de respecter ces lois.

Si l'Orienté Objet est le **mécanisme** (les outils), Et les Design Patterns sont les **plans** (l'architecture), Alors **SOLID** est le **Code de la Construction** (les normes de sécurité et de durabilité).

Voici une présentation des 5 piliers **SOLID** et comment ils justifient l'utilisation des patterns que nous voyons.

S - Single Responsibility Principle (Responsabilité Unique)

"Une classe ne doit avoir qu'une seule raison de changer."

Le Concept : Si vous avez une classe **Facture** qui calcule la TVA **ET** qui imprime la facture en PDF, vous violez le principe. Si la loi sur la TVA change, vous touchez à la classe. Si le format du PDF change, vous touchez à la même classe. C'est dangereux (risque de casser le calcul en changeant l'impression).

En C++ : Séparez le code ! Une classe **CalculateurTVA** et une classe **ImprimantePDF**.

Lien avec les Patterns :

- Le **Decorator** sert à ajouter des responsabilités visuelles sans polluer la classe métier.
- L'**Observer** sépare la donnée (le Modèle) de son affichage (la Vue).

O - Open/Closed Principle (Ouvert/Fermé)

"Les entités logicielles doivent être ouvertes à l'extension, mais fermées à la modification."

Le Concept : C'est le Saint Graal. Vous devez pouvoir ajouter une fonctionnalité (ex: un nouveau monstre dans le jeu) **sans modifier** le code source existant des autres classes.

En C++ : On utilise le polymorphisme et l'héritage. On ne touche pas au **if/else** du moteur de jeu, on crée juste un nouveau fichier **.cpp** avec une nouvelle classe.

Lien avec les Patterns :

- C'est la raison d'être de la **Factory Method** (ajouter un produit sans changer le créateur).
- C'est le cœur du **Strategy** (changer d'algorithme sans toucher au client).
- C'est la base du **Decorator** (étendre le comportement sans toucher la classe).

L - Liskov Substitution Principle (Substitution de Liskov)

"Les objets d'une classe enfant doivent pouvoir remplacer les objets de la classe parent sans que le programme ne plante."

Le Concept : Si votre code attend une **Voiture** et que je lui passe une **Ferrari** (qui hérite de Voiture), tout doit marcher.

Le Piège Classique (Rectangle vs Carré) : En maths, un carré est un rectangle. En C++, **NON**. Si j'ai une fonction **redimensionner(Rectangle& r)** qui change la largeur sans changer la hauteur... Et que je lui passe un **Carré** (dont la largeur et la hauteur sont liées)... Le **Carré** n'est plus un carré. Le comportement est cassé.

Lien avec les Patterns :

- Les patterns structurels comme **Adapter** aident parfois à contourner des interfaces mal conçues qui violent ce principe.

I - Interface Segregation Principle (Ségrégation des Interfaces)

"Mieux vaut plusieurs interfaces spécifiques qu'une seule interface générale."

Le Concept : Imaginez une interface **ITravailleur** avec les méthodes **coder()**, **manger()**, **dormir()**.

Si je crée une classe **Robot**, je suis obligé d'implémenter **manger()** et **dormir()** (même vides) pour que ça compile. C'est sale.

En C++ : Créez des petites classes abstraites pures : **ICodeur**, **IMangeur**. Le Robot n'héritera que de **ICodeur**. L'Humain héritera des deux.

Lien avec les Patterns :

- L'**Adapter** est souvent utilisé pour faire correspondre une "grosse" interface avec une "petite" interface attendue par le client.

D - Dependency Inversion Principle (Inversion des Dépendances)

"Dépendre des abstractions, pas des implémentations concrètes."

Le Concept : C'est ce que nous avons vu au tout début du cours (l'exemple **Game** vs **FileSaver**). Le module de haut niveau (Logique Métier) ne doit pas dépendre du module de bas niveau (Disque dur, Réseau). Les deux doivent dépendre d'une Interface.

En C++ : Utilisez des interfaces (**virtual pure**) et passez des pointeurs/références vers ces interfaces, jamais les objets concrets.

Lien avec les Patterns :

- **Factory Method, Abstract Factory, Builder** : Ils servent tous à *injecter* des dépendances concrètes tout en gardant le code couplé uniquement à des interfaces abstraites.

Anti Pattern

L'Ombre des Design Patterns : Les Anti-Patterns

Jusqu'à présent, nous avons parlé des "Meilleures Pratiques" (Best Practices). Mais il existe dans le monde du développement un côté obscur.

Un **Anti-Pattern**, c'est une solution à un problème récurrent qui semble séduisante au premier abord, mais qui s'avère être **contre-productive** et **dangereuse** à long terme.

La définition simple : Si le Design Pattern est une recette de chef étoilé, l'Anti-Pattern est la "recette de grand-mère" qui donne mal au ventre à tous les coups.

Pourquoi les étudier ? Pour développer votre "odorat".

Un bon développeur doit savoir "sentir" quand un code commence à puer (on parle de *Code Smell*).

Pourquoi je vous en parle maintenant ?

Parce que le plus grand danger qui vous guette à la sortie de ce cours, c'est le **Marteau d'Or**.

Vous allez avoir envie de mettre des Factory, des Builder et des Strategy partout.

Attention : Un Design Pattern ajoute de la complexité (plus de classes). Il ne doit être utilisé que si le problème de flexibilité est réel.

Règle d'or : N'appliquez pas un pattern "au cas où". Appliquez-le quand vous sentez la douleur du couplage fort. On garde ça en tête pour éviter de créer des monstres.

Voici les 4 Anti-Patterns les plus célèbres que vous devez fuir comme la peste...

The Golden Hammer (Le Marteau d'Or)

"Pour celui qui ne possède qu'un marteau, tout ressemble à un clou."

Abraham Maslow (psychologue américain, 1908-1970)

- **Le Symptôme** : Vous venez d'apprendre un Design Pattern (par exemple le *Singleton* ou l'*Observer*) et vous le trouvez génial. Résultat : vous l'utilisez PARTOUT, même quand ce n'est pas nécessaire.
- **L'Exemple** : Utiliser un *Singleton* pour passer une variable d'une fonction A à une fonction B, juste parce qu'on a la flemme de la passer en paramètre.
- **Le Remède** : Toujours se demander : "Est-ce que la solution la plus simple (KISS - Keep It Simple, Stupid) ne suffit pas ici ?".

The God Object (L'Objet Dieu / The Blob)

C'est l'Anti-Pattern n°1 en Programmation Orientée Objet.

- **Le Symptôme** : Une classe unique qui fait TOUT. Elle possède 3000 lignes de code, 50 méthodes et contrôle l'ensemble du programme. Les autres classes ne sont que des coquilles vides contenant des données.
- **Pourquoi c'est tentant** : C'est facile au début. On met tout dans `GameController` ou `SystemManager`. Pas besoin de réfléchir à l'architecture.
- **La Réalité** : C'est un cauchemar à maintenir. Vous violez le principe **S** (Single Responsibility) de **SOLID** de manière spectaculaire. Si vous touchez à une ligne, vous risquez de casser une fonctionnalité qui n'a rien à voir.
- **Le Remède** : Découpez !
Utilisez les patterns (Strategy, State, Command) pour déléguer les responsabilités.

Poltergeist (L'Esprit Frappeur)

- **Le Symptôme** : Des classes qui ne servent à rien.
Elles apparaissent, lancent une méthode dans une autre classe, et disparaissent. Elles n'ont pas d'état, pas de responsabilité propre.
- **Exemple** : Une classe `ManagerDeConnexionController` qui ne fait qu'appeler `Connexion::connect()`.
- **Le Remède** : Supprimez ces intermédiaires inutiles.
Si une classe ne fait rien d'autre que passer le plat, elle est probablement superflue (sauf si c'est un Proxy ou un Adapter justifié).

Spaghetti Code (Code Spaghetti)

Celui-là, tout le monde le connaît, mais il a une définition précise.

- **Le Symptôme** : Un flux d'exécution impossible à suivre. Des `goto` (ou leur équivalent moderne : des exceptions utilisées pour contrôler le flux), des boucles imbriquées sur 5 niveaux, des fonctions qui font 500 lignes.
- **La Réalité** : Si vous devez dessiner le schéma de votre code et que ça ressemble à un plat de pâtes, c'est perdu.
- **Le Remède** : Refactoring. Découpage en petites fonctions.
Utilisation du pattern **State** pour gérer les machines à états complexes au lieu de `if/else` géants.

Le Singleton

Passons à notre premier pattern officiel. C'est le plus simple à comprendre, **mais paradoxalement le plus controversé**.

La Philosophie

Le **Singleton** répond à une question simple : **Comment m'assurer qu'une classe n'a qu'une seule et unique instance dans tout mon programme, et y accéder facilement ?**

Exemples d'utilisation typiques :

- Le gestionnaire de configuration (vous ne voulez pas charger le fichier `config.ini` 50 fois).
- Le système de Logging.
- Le gestionnaire de connexion à la base de données.

En C++, pour garantir l'unicité, il faut "verrouiller" les manières habituelles de créer des objets.

1. **Constructeur privé** : Personne ne peut faire `new Singleton()`.
2. **Constructeur de copie supprimé (`delete`)** : Personne ne peut cloner l'objet.
3. **Opérateur d'affectation supprimé (`delete`)** : Idem.
4. **Méthode statique publique** : Le seul point d'entrée pour récupérer l'instance.

Singleton pour log

Voici la version moderne (C++11 et +), souvent appelée le **Meyers' Singleton**. Elle est simple et *Thread-Safe* (sûre en multithreading) par défaut.

```
// --- UTILISATION ---
void fonctionQuelconque() {
    // On ne fait pas "new Logger()", on demande l'instance
    Logger::getInstance().log("Message depuis une fonction");
}

int main() {
    // Premier appel : le constructeur est appelé ici
    Logger::getInstance().log("Démarrage du programme");

    fonctionQuelconque();

    // Deuxième appel : on réutilise la même instance (pas de constructeur)
    Logger::getInstance().log("Fin du programme");

    return 0;
}
```

Les "Do" et "Don't" du Singleton

Attention, le Singleton est souvent appelé un **"Anti-Pattern"** s'il est mal utilisé.

- **L'avantage** : C'est pratique. Vous avez accès à votre `Logger` de n'importe où sans avoir à le passer en paramètre à 15 fonctions.
- **Le piège** : C'est une **variable globale déguisée**.
 - Si votre Singleton garde un état (des variables) modifiable, cela crée des dépendances cachées entre des parties de votre code qui ne devraient pas se connaître.
 - Cela rend les **Tests Unitaires** très difficiles (impossible de "reset" le Singleton proprement entre deux tests).

Règle d'or : Utilisez le Singleton pour des objets qui sont vraiment uniques par nature (Service de Log, Driver Audio), mais n'en abusez pas pour stocker des données partagées (ex: ne faites pas un Singleton `CurrentPlayer`).

```
class Logger {
public:
    // La méthode magique pour accéder à l'instance
    static Logger& getInstance() {
        // Cette variable statique est initialisée à la première exécution
        // de la fonction, et détruite à la fin du programme.
        static Logger instance;
        return instance;
    }

    // Une méthode métier normale
    void log(const std::string& msg) {
        std::cout << "[LOG] " << msg << std::endl;
    }

    // --- Verrouillage de la création ---
    // 1. On interdit la copie
    Logger(const Logger&) = delete;
    // 2. On interdit l'assignation (ex: log1 = log2)
    void operator=(const Logger&) = delete;

private:
    // 3. Le constructeur est privé !
    Logger() {
        std::cout << "Construction du Logger (une seule fois !)" << std::endl;
    }

    // Destructeur privé ou public, selon si vous voulez gérer la destruction manuellement
    ~Logger() {
        std::cout << "Destruction du Logger" << std::endl;
    }
};
```

Exercice : Singleton

Exercice : Le Gestionnaire de Configuration

Contexte : Vous développez un jeu vidéo. Ce jeu possède des paramètres globaux (Volume audio, Résolution écran, Difficulté) qui doivent être chargés au démarrage et rester accessibles par n'importe quel module du jeu (le moteur audio, le moteur graphique, l'IA).

Il est interdit de recharger ces paramètres depuis le disque dur à chaque fois qu'on en a besoin (trop lent), et il est interdit d'avoir deux versions différentes de la configuration en mémoire.

Votre mission : Implémentez une classe `GameConfig` en utilisant le pattern **Singleton**.

Cahier des charges technique :

1. La classe doit stocker des paires Clé/Valeur (ex: `std::map<std::string, int>`).
2. On doit pouvoir faire `GameConfig::getInstance().setValue("volume", 50);`.
3. On doit pouvoir faire `int v = GameConfig::getInstance().getValue("volume");`.
4. **Important :** Dans votre `main()`, prouvez que le Singleton fonctionne en appelant `getInstance()` deux fois via deux variables différentes, et affichez leurs adresses mémoires. Elles doivent être identiques.
5. **Sécurité :** Assurez-vous (via le code) qu'il est **impossible** de copier cette classe.

Factory Method

C'est probablement le pattern le plus utilisé dans les frameworks modernes.

Si le Singleton répondait à "Comment avoir une seule instance ?", la **Factory Method** répond à : "**Comment créer des objets sans connaître leur classe exacte à l'avance ?**".

C'est ici que le **Polymorphisme** prend tout son sens.

Le Scénario Catastrophe (Le Problème)

Imaginez que vous codez une application de logistique. Au début, votre client ne gère que le transport routier par camions. Partout dans votre code, vous avez :

```
Transport* t = new Camion();  
t->livrer();
```

Un an plus tard, l'entreprise explose et se lance dans le fret maritime.

Vous devez gérer des **Bateaux**. **Le drame** : Vous devez parcourir tout votre code pour remplacer les `new Camion()` par une logique conditionnelle horrible :

```
Transport* t;  
if (type == "MER") {  
    t = new Bateau();  
} else if (type == "ROUTE") {  
    t = new Camion();  
}  
// Et si demain on ajoute des Avions ? On modifie encore le code.
```

Ceci viole le principe **Open/Closed** (Ouvert à l'extension, Fermé à la modification).

La Solution Factory Method

L'idée est d'arrêter d'utiliser `new` directement dans la logique métier. On déplace cet appel dans une méthode spéciale (la "Fabrique"). On définit une interface pour créer un objet, mais on laisse les sous-classes décider quelle classe instancier.

Factory Method : implementation

Nous allons utiliser `std::unique_ptr` pour gérer la mémoire automatiquement (fini les `delete` oubliés).

L'architecture :

Produits Concrets : `Camion`, `Bateau`

```
// --- 2. LES PRODUITS CONCRETS ---
class Camion : public ITransport {
public:
    void livrer() const override {
        std::cout << "Livraison par la route dans un carton." << std::endl;
    }
};

class Bateau : public ITransport {
public:
    void livrer() const override {
        std::cout << "Livraison par la mer dans un conteneur." << std::endl;
    }
};
```

Créateurs Concrets : `LogistiqueRoute`, `LogistiqueMer`.

```
// --- 4. LES CRÉATEURS CONCRETS ---
class LogistiqueRoute : public Logistique {
public:
    // Implémentation de la fabrique : on retourne un Camion
    std::unique_ptr<ITransport> creerTransport() const override {
        return std::make_unique<Camion>();
    }
};

class LogistiqueMer : public Logistique {
public:
    // Implémentation de la fabrique : on retourne un Bateau
    std::unique_ptr<ITransport> creerTransport() const override {
        return std::make_unique<Bateau>();
    }
};
```

Produit (Interface) : `ITransport`

```
// --- 1. L'INTERFACE DU PRODUIT ---
class ITransport {
public:
    virtual void livrer() const = 0;
    virtual ~ITransport() = default; // Toujours un destructeur virtuel !
};
```

Créateur (Abstrait) : `Logistique` (C'est lui qui contient la méthode fabrique virtuelle).

```
// --- 3. LE CRÉATEUR (La classe métier) ---
class Logistique {
public:
    virtual ~Logistique() = default;

    // VOICI LA FACTORY METHOD
    // Elle est "virtual pure" ici, forçant les sous-classes à l'implémenter.
    virtual std::unique_ptr<ITransport> creerTransport() const = 0;

    // LA LOGIQUE MÉTIER
    // Notez bien : cette fonction ne sait pas si elle utilise un camion ou un bateau.
    // Elle travaille uniquement avec l'interface ITransport.
    void planifierLivraison() const {
        // Appel de la fabrique pour obtenir un objet
        std::unique_ptr<ITransport> transport = creerTransport();

        // Utilisation de l'objet
        std::cout << "Logistique : Début du processus..." << std::endl;
        transport->livrer();
    }
};
```

Fichier : `creational/factory.cpp`

Analyse Factory Method

Pourquoi ce code est-il "mieux" ?

1. **Découplage** : La méthode `planifierLivraison()` (le cœur de votre application) ne dépend plus des classes `Camion` ou `Bateau`. Elle dépend de `ITransport`.
2. **Extensibilité** : Si demain vous devez ajouter `Avion` :
 - Vous créez `class Avion : public ITransport`.
 - Vous créez `class LogistiqueCiel : public Logistique`.
 - **Vous ne touchez pas une seule ligne du code existant** (`Logistique`, `Camion`, etc.). C'est propre.

Quand l'utiliser ?

- Quand vous ne savez pas à l'avance quels types d'objets votre code devra manipuler.
- Quand vous voulez permettre aux utilisateurs de votre bibliothèque d'étendre vos composants internes (ex: un Framework UI qui laisse le développeur créer ses propres boutons).

Factory Method : Exercice

l'enjeu est de comprendre comment déléguer la création d'objets aux sous-classes pour respecter le principe d'ouverture/fermeture (Open/Closed Principle).

Voici un exercice classique mais très parlant, orienté "Jeu Vidéo"

Exercice : Les Factions du Jeu de Stratégie

Contexte : Vous développez le moteur d'un jeu de stratégie en temps réel (RTS) type *Warcraft* ou *Age of Empires*. Le jeu possède plusieurs factions (par exemple : les **Humains** et les **Robots**).

Chaque faction possède un bâtiment principal (une Caserne) qui permet de produire une unité de combat de base. Le moteur du jeu (le code principal) doit pouvoir dire "Caserne, entraîne une unité !" sans savoir s'il s'agit d'une caserne humaine ou robotique, et sans savoir quel type de soldat va en sortir.

Votre mission : Implémentez le pattern **Factory Method** pour gérer cette production d'unités.

Cahier des charges technique :

1. **Produit (Interface) :** Créez une interface Unite avec une méthode combattre().
2. **Produits Concrets :** Créez deux classes :
 - Archer (pour les Humains) -> affiche "L'archer tire une flèche."
 - Terminator (pour les Robots) -> affiche "Le robot tire au laser."
3. **Créateur (Abstrait) :** Créez une classe Caserne qui possède :
 - Une méthode virtuelle pure creerUnite() (La Factory Method).
 - Une méthode concrète entraîner() qui appelle la fabrique pour obtenir une unité, puis lui ordonne de combattre.
4. **Créateurs Concrets :** Créez CaserneHumaine (produit des Archers) et CaserneRobot (produit des Terminators).
5. **Main :** Dans le programme principal, utilisez un pointeur de type Caserne pour manipuler indifféremment une caserne humaine ou robot.

Le Builder (Le Monteur)

Le Problème : Le "Constructeur Télescopique"

Imaginez que vous deviez créer une classe `Pizza`. Une pizza peut avoir une taille, de la pâte fine ou épaisse, de la sauce tomate ou crème, du fromage (ou pas), du pepperoni, des champignons, des olives...

En C++, sans pattern, on finit souvent avec une horreur pareille :

```
// L'ENFER DU DÉVELOPPEUR
Pizza* p = new Pizza(30, true, false, true, false, true, false, "Tomate");
```

Pourquoi est-ce mauvais ?

Illisible : Que signifie le 3ème `true` ? Et le 5ème `false` ? Il faut sans cesse vérifier la documentation.

Rigide : Si vous voulez ajouter un ingrédient "Ananas" (sacrilège !), vous devez modifier le constructeur et casser tous les appels existants.

Inutile : La plupart des paramètres sont optionnels, mais vous êtes obligés de passer des valeurs par défaut (`null` ou `false`).

La Solution : La Construction "Pas à Pas"

L'idée du Builder est de sortir la logique de construction de la classe `Pizza` et de la mettre dans une classe dédiée : le `PizzaBuilder`. Au lieu d'appeler un constructeur géant, on appelle des méthodes nommées pour chaque étape.

Il faudrait pouvoir écrire quelque chose comme :

```
Pizza maPizza = PizzaBuilder("Margaritha").pateFine().tomate().pepperonni().build();
```

et qu'il y ait un automatisme pour mettre des valeurs par défaut si on ne les spécifie pas (genre `fromage=true` par défaut)

Builder : implementation

L'Implémentation "Fluent Interface" en C++

C'est la version la plus populaire en C++. On fait en sorte que chaque méthode du Builder retourne une référence au Builder lui-même (`return *this`). Cela permet de chaîner les appels.

Le produit complexe (Computer)

C'est le produit final attendu.

Dans tous les cas, on veut éviter de créer directement un Computer en lui passant tous ses paramètres.

Le Builder

C'est ce qui va nous permettre de créer un Computer et d'appeler des méthodes chaînées de paramétrage.

```
// --- 1. LE PRODUIT COMPLEXE ---
class Computer {
public:
    // Beaucoup d'attributs, certains obligatoires, d'autres optionnels
    std::string cpu;
    std::string gpu;
    int ram_gb;
    int storage_gb;
    bool hasWifi;
    bool hasCooling;

    void afficherSpecs() const {
        std::cout << "--- PC SPECS ---" << std::endl;
        std::cout << "CPU: " << cpu << " | GPU: " << gpu << std::endl;
        std::cout << "RAM: " << ram_gb << "GB | SSD: " << storage_gb << "GB" << std::endl;
        std::cout << "WiFi: " << (hasWifi ? "Oui" : "Non") << std::endl;
        std::cout << "Watercooling: " << (hasCooling ? "Oui" : "Non") << std::endl;
        std::cout << "-----" << std::endl;
    }
};

// --- 2. LE BUILDER ---
class ComputerBuilder {
private:
    // Le Builder contient une instance temporaire du produit
    Computer computer;
    // Constructeur par défaut inaccessible
    ComputerBuilder() = delete;
    // On interdit la copie du Builder pour éviter des erreurs
    ComputerBuilder(const ComputerBuilder&) = delete;
    void operator=(const ComputerBuilder&) = delete;

public:
    // On peut forcer les paramètres obligatoires dans le constructeur du Builder
    ComputerBuilder(const std::string& cpu, int ram) {
        computer.cpu = cpu;
        computer.ram_gb = ram;
        // Valeurs par défaut
        computer.gpu = "Integrated";
        computer.storage_gb = 256;
        computer.hasWifi = false;
        computer.hasCooling = false;
    }
}
```

Builder : implementation

Méthodes de construction

L'astuce est ici : chaque méthode renvoie un pointeur sur l'objet "this", ce qui permettra ensuite de chaîner les appels à ces méthodes (`methode1().methode2()...`)

Le code client

C'est ce qui va nous permettre de créer un Computer et d'appeler des méthodes chaînées de paramétrage.
Il est simple et lisible.

```
// --- 3. CODE CLIENT ---
int main() {
    std::cout << "Construction d'un PC Bureau : " << std::endl;
    // Syntaxe fluide et lisible
    Computer officePC = ComputerBuilder("Intel i3", 8)
        .setStorage(512)
        .withWifi()
        .build();
    officePC.afficherSpecs();

    std::cout << "\nConstruction d'un PC Gamer : " << std::endl;
    Computer gamerPC = ComputerBuilder("AMD Ryzen 9", 32)
        .setGPU("Nvidia RTX 4090")
        .setStorage(2000)
        .withWaterCooling()
        .build();
    gamerPC.afficherSpecs();

    return 0;
}
```

```
// --- MÉTHODES DE CONSTRUCTION (CHAINABLES) ---

ComputerBuilder& setGPU(const std::string& gpuModel) {
    computer.gpu = gpuModel;
    return *this; // L'astuce est ici : on retourne l'objet lui-même
}

ComputerBuilder& setStorage(int gb) {
    computer.storage_gb = gb;
    return *this;
}

ComputerBuilder& withWifi() {
    computer.hasWifi = true;
    return *this;
}

ComputerBuilder& withWaterCooling() {
    computer.hasCooling = true;
    return *this;
}

// --- ÉTAPE FINALE : RÉCUPÉRATION DU PRODUIT ---
Computer build() {
    return computer;
}
```

Fichier : `creational/Builder.cpp`

Builder : concept du “Director”

Dans le pattern complet (GoF), il y a souvent une classe supplémentaire appelée le **Director** (Directeur).

- Le **Builder** fournit les briques (setGPU, setRAM...).
- Le **Director** connaît la recette.

Exemple : Si vous fabriquez souvent le même type de PC, vous pouvez avoir une classe **DirecteurDeMagasin** qui possède une méthode **construirePCGamerStandard(Builder& builder)**.

```
void construirePCGamerStandard(ComputerBuilder& builder) {  
    builder.setGPU("RTX 4070")  
        .setStorage(1024)  
        .withWaterCooling();  
}
```

Cela permet de séparer complètement la *recette* (Director) de la *fabrication technique* (Builder).

```
std::cout << "\nConstruction d'un PC Gamer Standard via fonction : " << std::endl;  
ComputerBuilder builder("AMD Ryzen 7", 16);  
construirePCGamerStandard(builder);  
Computer standardGamerPC = builder.build();  
standardGamerPC.afficherSpecs();
```

Résumé Builder

- **Utilisez le Builder quand** : Un objet a plus de 3 ou 4 paramètres dans son constructeur, surtout s'ils sont optionnels.
- **L'avantage clé** : Votre code client devient lisible comme une phrase en anglais (`builder.setThis().withThat()`).
- **En C++** : N'oubliez pas le `return *this;` dans les méthodes du builder pour permettre le chaînage.



Exercice : Le Générateur de Requêtes SQL

Contexte : Vous écrivez un petit ORM (Object Relational Mapper). Vos collègues en ont assez de concaténer des chaînes de caractères à la main avec des erreurs d'espaces ou de virgules. Ils veulent écrire du code C++ qui ressemble à du SQL, de manière fluide.

Votre mission : Implémentez une classe `SqlBuilder` qui permet de construire une requête `SELECT` proprement.

Cahier des charges technique :

1. **Classe `SqlBuilder`** : Elle doit stocker les morceaux de la requête (table, colonnes, conditions WHERE).
2. **Méthodes chaînables (Fluent Interface)** :
 - `from(string table)` : Définit la table (Obligatoire).
 - `select(vector<string> colonnes)` : Définit les colonnes. Si non appelé, par défaut c'est `*`.
 - `where(string condition)` : Ajoute une condition. **Attention** : Si on appelle `where` plusieurs fois, les conditions doivent être reliées par un `AND`.
 - `limit(int limit)` : Ajoute une limite (Optionnel).
3. **Méthode `build()`** : Elle assemble le tout et retourne une `std::string` prête à être exécutée.

Code client cible (ce que l'utilisateur veut écrire) :

Ce pattern rend le code client **indépendant de la syntaxe SQL**. Si demain vous changez de base de données (ex: syntaxe Oracle vs MySQL pour le `LIMIT`), vous ne changez que la méthode `build()`, pas le code dans tout le projet.

```
SqlBuilder builder;
std::string req1 = builder.select({"nom", "email"})
    .from("clients")
    .where("actif = 1")
    .where("solde < 0") // Ajoute un AND automatiquement
    .limit(5)
    .build();
```

Patterns de structure

Si les patterns de *Création* consistent à fabriquer les briques, les patterns de *Structure* consistent à **cimenter** ces briques ensemble.

Le mot d'ordre ici est : **Composition**.

Le piège à éviter : L'héritage excessif.

Nous allons voir comment faire collaborer des classes qui n'étaient pas prévues pour ça, et comment ajouter des fonctionnalités sans modifier le code existant.

Voici les deux premiers incontournables : l'**Adapter** et le **Decorator**.

Adapter (L'Adaptateur)

Le Problème

Vous avez une classe existante (souvent une vieille librairie ou un code tiers) qui fait exactement ce que vous voulez, mais son **interface** (ses noms de méthodes) ne correspond pas à ce que votre application attend.

Exemple :

- Votre application attend une méthode `dessiner()`.
- La vieille librairie possède une méthode `afficherCarre()`.

Vous ne pouvez pas modifier la librairie (pas le code source, ou risque de bugs).

2. La Solution

Comme un adaptateur de prise électrique (EU vers US), on crée une classe intermédiaire qui "traduit" les appels.

3. Implémentation en C++

On utilise la **Composition** : l'Adaptateur *possède* une instance de l'objet à adapter et traduit les appels.

À **retenir** : L'Adapter permet de sauver du code "Legacy" sans polluer votre nouveau code propre.

```
// --- 1. TARGET (Ce que votre code attend) ---
class Forme {
public:
    virtual void dessiner() const = 0;
    virtual ~Forme() = default;
};
```

```
// --- 2. ADAPTEE (L'ancienne classe incompatible) ---
// Imaginez que c'est une librairie tierce non modifiable
class VieuxCarre {
public:
    void afficherCarre(int x, int y, int cote) const {
        std::cout << "VieuxCarre: Coin(" << x
            << ", " << y << ") Côté: " << cote << std::endl;
    }
};
```

```
// --- 3. ADAPTER (Le traducteur) ---
class CarreAdapter : public Forme {
private:
    // L'adaptateur "enveloppe" l'objet incompatible
    std::unique_ptr<VieuxCarre> vieuxCarre;
    int size;

public:
    CarreAdapter(int w) : size(w) {
        vieuxCarre = std::make_unique<VieuxCarre>();
    }

    void dessiner() const override {
        // TRADUCTION : On convertit l'appel 'dessiner' en 'afficherCarre'
        // On décide arbitrairement des coordonnées 0,0 car Forme n'en a pas
        std::cout << "[Adapter] Traduction de l'appel..." << std::endl;
        vieuxCarre->afficherCarre(0, 0, size);
    }
};
```

```
// --- CLIENT ---
void lancerGraphismes(const Forme& f) {
    f.dessiner();
}

int main() {
    CarreAdapter monCarre(10);

    // La fonction lancerGraphismes croit manipuler une "Forme" standard.
    // En coulisses, c'est le VieuxCarre qui bosse.
    lancerGraphismes(monCarre);

    return 0;
}
```



Exercice : Le Choc des Générations (Legacy vs Modern)

Voici un exercice qui simule un cas très fréquent en entreprise : l'intégration d'une vieille bibliothèque ("Legacy Code") dans un projet moderne.

Contexte : Vous travaillez sur un logiciel de dessin vectoriel ultra-moderne. Votre moteur graphique gère tous les objets via une interface propre IRenderable. Votre chef de projet vous demande d'intégrer une classe VieuxRectangle écrite par un stagiaire en 1998. Cette classe fonctionne, mais son interface est incompatible avec la vôtre.

Le Défi Mathématique : En plus des noms de méthodes différents, la logique de coordonnées est différente !

- **Votre système moderne** dessine en utilisant : x, y (point haut-gauche), largeur, hauteur.
- **Le vieux système** dessine en utilisant : x1, y1 (haut-gauche), x2, y2 (bas-droite).

Votre mission : Créez un **Adapter** pour pouvoir utiliser VieuxRectangle dans votre moteur moderne sans modifier la classe VieuxRectangle.

Cahier des charges technique :

1. **Target (Moderne) :** Interface IRenderable avec une méthode `void draw()`.
2. **Adaptee (Vieux) :** Classe VieuxRectangle. Elle possède une méthode `affiche(int x1, int y1, int x2, int y2)`.
3. **Adapter :** Classe RectangleAdapter.
 - Elle doit implémenter IRenderable.
 - Elle doit posséder une instance de VieuxRectangle.
 - Son constructeur doit prendre les paramètres modernes (x, y, w, h).
 - Sa méthode `draw()` doit calculer les coordonnées x2 et y2 avant d'appeler la méthode du vieux rectangle.

Indice : $x2 = x + w$