Bienvenue au cours “Design Pattern-2”

Nom de l’enseignant : Fabrice Meynckens
Promo : 2025-2026
Classe : B2CDEV

Objectifs de la session

Les Patterns de Structure (Partie 2)

- **Decorator**
- **Composite**

Les Patterns de comportement

- **Observer**
- **Strategy**
- **Command**

https://github.com/swanee-31/Design-Patterns

Decorator (Le Décorateur)

Le Problème : L'Explosion de Classes

Imaginez que vous gérez des fenêtres d'interface graphique. Vous avez une classe `Fenetre`. Vous voulez des fenêtres avec bordure : `FenetreAvecBordure`. Vous voulez des fenêtres avec barre de défilement : `FenetreAvecScroll`. Et si vous voulez les deux ? `FenetreAvecBordureEtScroll` ? Et si on ajoute une ombre ? `FenetreAvecBordureEtScrollEtOmbre` ?

C'est l'enfer combinatoire. L'héritage statique ne marche pas ici.

La Solution : Les "Poupées Russes"

Le **Decorator** permet d'ajouter des comportements à un objet *dynamiquement* (à l'exécution) en l'enveloppant dans un autre objet.

L'astuce géniale : Le Décorateur implémente l'interface `Widget` ET contient un `Widget`. Il est donc à la fois contenant et contenu.

Cela permet “ d’envelopper “ les appels dans un service de base.

Decorator (Le Décorateur)

Interface commune

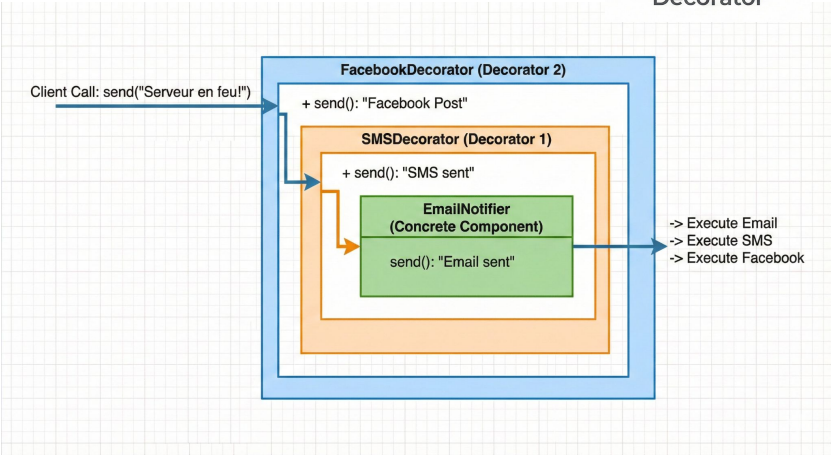
```cpp
// --- 1. COMPONENT (Interface Commune) ---
class INotifier {
public:
    virtual void send(const std::string& msg) const = 0;
    virtual ~INotifier() = default;
};

// --- 2. CONCRETE COMPONENT (L'objet de base) ---
class EmailNotifier : public INotifier {
public:
    void send(const std::string& msg) const override {
        std::cout << "Email envoyé : " << msg << std::endl;
    }
};
```

Décorateur de base

```cpp
// --- 3. BASE DECORATOR (La coquille vide) ---
class NotifierDecorator : public INotifier {
protected:
    // Il possède un pointeur vers le composant qu'il décore
    // (Note: on utilise un pointeur brut ou shared_ptr ici pour l'exemple de chainage,
    // unique_ptr est plus complexe à gérer dans des chaînes dynamiques)
    INotifier* wrappee;

public:
    NotifierDecorator(INotifier* n) : wrappee(n) {}

    void send(const std::string& msg) const override {
        // Par défaut, il délègue juste le travail au suivant
        wrappee->send(msg);
    }
};
```

Décorateur additionnels (options)

```cpp
// --- 4. CONCRETE DECORATORS (Les options) ---

class SMSDecorator : public NotifierDecorator {
public:
    SMSDecorator(INotifier* n) : NotifierDecorator(n) {}

    void send(const std::string& msg) const override {
        // 1. On fait le travail de base (Email ou autre décorateur)
        NotifierDecorator::send(msg);
        // 2. On ajoute notre comportement
        std::cout << "SMS envoyé : " << msg << std::endl;
    }
};

class FacebookDecorator : public NotifierDecorator {
public:
    FacebookDecorator(INotifier* n) : NotifierDecorator(n) {}

    void send(const std::string& msg) const override {
        NotifierDecorator::send(msg);
        std::cout << "Facebook Post : " << msg << std::endl;
    }
};
```



Utilisation Client

```cpp
int main() {
    // 1. On crée le service de base
    EmailNotifier* email = new EmailNotifier();

    // 2. On veut ajouter les SMS
    SMSDecorator* emailAndSMS = new SMSDecorator(email);

    // 3. On veut AUSSI Facebook (on emballe/enveloppe le précédent)
    FacebookDecorator* allNotifiers = new FacebookDecorator(emailAndSMS);

    std::cout << "--- Envoi d'une alerte critique ---" << std::endl;
    // L'appel traverse toute la chaîne : Facebook -> SMS -> Email
    allNotifiers->send("Serveur en feu !");
```

Exercice : La Forge Légendaire

Contexte & Défi :

Vous développez le système de combat d'un jeu type Skyrim ou Elden Ring. Le joueur commence avec une arme de base et peut appliquer des enchantements (Rune de Feu, Poison...).

Le problème : L'héritage classique mène à l'explosion combinatoire (EpeeDeFeu, EpeeDeFeuEtGlace...). EpeeDeFeuEtGlace...).

Vous devez utiliser le **Decorator** pour empiler les effets dynamiquement.

Votre mission :

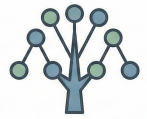
Créez un système pour calculer les dégâts totaux et la description complète.

Cahier des charges technique :

1. **Interface Commune (Arme) :**
   - getDescription() (string), getDegats() (int)
2. **Composant Concret (EpeeBasique) :**
   - Description : "Épée rouillée", Dégâts : 10
3. **Décorateur de Base (DecorateurArme) :**
   - Pointeur vers Arme, délègue les appels
4. **Décorateurs Concrets :**
   - RuneDeFeu (+5 dégâts, "de Feu")
   - Poison (+2 dégâts, "Empoisonnée")
5. **Main :**
   1. Créez une Épée.
   2. Transformez-la en "Épée de Feu".
   3. Ajoutez du poison pour en faire une "Épée une "Épée de Feu Empoisonnée".
   4. Affichez le résultat final (Description et Dégâts).

Attention à la gestion de la mémoire si vous utilisez des pointeurs bruts, sinon `std::unique_ptr` est votre ami.

Pattern de structure : Composite



Le Problème : La Boîte dans la Boîte

Imaginez que vous concevez un système pour une entreprise de livraison.

- Vous avez des **Produits** (un téléphone, un marteau).
- Vous avez des **Boîtes** qui contiennent des produits... mais qui peuvent aussi contenir d'autres petites **Boîtes**.

On vous demande : **"Calcule le prix total de cette grosse caisse d'expédition."**

L'approche naïve (l'enfer) : Vous ouvrez la caisse. Vous regardez ce qu'il y a dedans.

- Si c'est un produit -> J'ajoute le prix.
- Si c'est une boîte -> Euh... je dois ouvrir cette boîte, regarder dedans, et si je trouve encore une boîte...

En code, cela donnerait des boucles imbriquées et des vérifications de type (`if (objet is Boite)`) horribles.

Le code client devient dépendant de la structure interne.

La Solution Composite

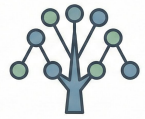
Le pattern Composite propose de traiter **l'élément simple** (la Feuille / Leaf) et **le conteneur** (le Composite) de la même manière, via une interface commune.

Pour le client, une Boîte **EST** un Produit comme un autre. Elle a un prix. Sauf que son prix se calcule en additionnant le contenu. Le client n'a pas besoin de savoir qu'il y a une boucle.

Implémentation en C++ (Système de Fichiers)

L'exemple le plus parlant est celui de votre disque dur : Fichiers vs Dossiers. Un Dossier contient des Fichiers ET d'autres Dossiers. Si je demande `getTaille()` à un Dossier, il doit interroger ses enfants.

Pattern de structure : Composite

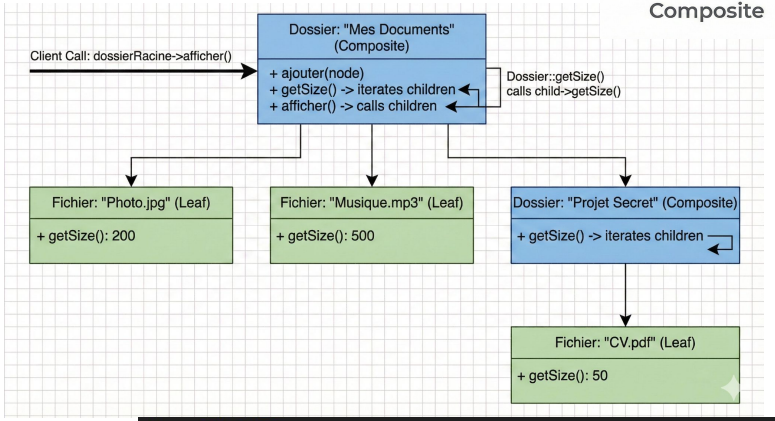


Interface commune

```cpp
// --- 1. COMPONENT (L'interface commune) ---
class FileSystemNode {
public:
    virtual int getSize() const = 0;
    virtual void afficher(int indentation = 0) const = 0;
    virtual ~FileSystemNode() = default;

protected:
    // Utilitaire pour afficher des jolis espaces
    void printIndent(int indent) const {
        for (int i = 0; i < indent; ++i) std::cout << "  ";
    }
};
```

Objet feuille

```cpp
// --- 2. LEAF (La feuille : L'objet simple) ---
class Fichier : public FileSystemNode {
private:
    std::string nom;
    int taille;

public:
    Fichier(std::string n, int t) : nom(n), taille(t) {}

    int getSize() const override {
        return taille;
    }

    void afficher(int indentation) const override {
        printIndent(indentation);
        std::cout << "- Fichier: " << nom << " (" << taille << " ko)" << std::endl;
    }
};
```



Le conteneur

Contient la logique (récursivité)

```cpp
class Dossier : public FileSystemNode {
private:
    std::string nom;
    // LA CLÉ EST ICI : Le dossier contient une liste de l'INTERFACE (pas des classes concrètes)
    // Il peut donc contenir des Fichiers ET des Dossiers indifféremment.
    std::vector<std::shared_ptr<FileSystemNode>> enfants;

public:
    Dossier(std::string n) : nom(n) {}

    // Méthode spécifique pour ajouter des éléments
    void ajouter(std::shared_ptr<FileSystemNode> composant) {
        enfants.push_back(composant);
    }

    // Le calcul récursif de la taille
    int getSize() const override {
        int total = 0;
        for (const auto& enfant : enfants) {
            // L'appel polymorphique magique :
            // Si l'enfant est un fichier -> renvoie sa taille
            // Si l'enfant est un dossier -> renvoie la somme de ses enfants (récursion)
            total += enfant->getSize();
        }
        return total;
    }

    void afficher(int indentation) const override {
        printIndent(indentation);
        std::cout << "+ Dossier: " << nom << " (Total: " << getSize() << " ko)" << std::endl;

        // On demande aux enfants de s'afficher (avec un cran d'indentation en plus)
        for (const auto& enfant : enfants) {
            enfant->afficher(indentation + 1);
        }
    }
};
```

```cpp
// --- CODE CLIENT ---
int main() {
    // 1. On crée des feuilles (Fichiers)
    auto f1 = std::make_shared<Fichier>("Photo.jpg", 200);
    auto f2 = std::make_shared<Fichier>("CV.pdf", 50);
    auto f3 = std::make_shared<Fichier>("Musique.mp3", 500);

    // 2. On crée des composites (Dossiers)
    auto dossierRacine = std::make_shared<Dossier>("Mes Documents");
    auto sousDossier = std::make_shared<Dossier>("Projet Secret");

    // 3. On assemble l'arbre
    // Le sous-dossier contient un fichier
    sousDossier->ajouter(f2); // CV.pdf

    // La racine contient deux fichiers et le sous-dossier
    dossierRacine->ajouter(f1); // Photo
    dossierRacine->ajouter(f3); // Musique
    dossierRacine->ajouter(sousDossier); // Dossier imbriqué

    // 4. Appel unique
    std::cout << "--- Structure du Disque ---" << std::endl;
    // Le client n'a pas besoin de savoir qu'il y a une structure complexe en dessous.
    // Il appelle juste 'afficher'.
    dossierRacine->afficher(1);

    return 0;
}
```

Exercice : Le Menu du Fast-Food

Contexte & Défi

Vous développez le logiciel de caisse d'une grande chaîne de fast-food. Le menu est complexe. Les clients peuvent commander :

- Des **plats individuuls** (un Burger seul, une Frite seule, un Soda seul).
- Des **menus "Combo"** qui regroupent plusieurs plats (ex: Menu Best-Of = Burger + Frite + Soda).
- Des **offres "Maxi Box"** qui peuvent contenir des plats individuels ET des menus Combo entiers.

Le problème :

Le système de caisse doit pouvoir demander le prix à n'importe quel article ajouté au panier, que ce soit une simple frite ou une "Maxi Box" complexe, sans avoir à vérifier son type exact avec des `if (type == BOX) ... else if (type == FRITE)....`

Votre mission :

Utilisez le pattern **Composite** pour modéliser cette structure et calculer le prix total d'une commande complexe de manière récursive.

Cahier des charges technique:

1. **Composant (Interface Commune)** : Créez une classe abstraite `ElementMenu`.
   - Méthode `double getPrix() const`.
   - Méthode `void afficher(int indentation) const` (pour visualiser le ticket de caisse).
2. **Feuille (Leaf)** : Créez une classe `PlatIndividuel`.
   - Elle a un nom et un prix fixe.
3. **Composite** : Créez une classe `MenuCombo`.
   - Elle a un nom.
   - Elle contient une liste de `std::list<std::shared_ptr<ElementMenu>>`.
   - Sa méthode `getPrix()` doit retourner la somme des prix de ses enfants.
   - Sa méthode `afficher()` doit afficher son nom, puis demander à ses enfants de s'affants avec une indentation supplémentaire.
4. **Dans le `main` :**
   - Créez des plats : BigBurger (8.50€), PetiteFrite (2.50€), Soda (2.00€), Café (1.50€).
   - Créez un menu 'Menu Classique' contenant : BigBurger + PetiteFrite + Soda.
   - Créez une grosse commande 'Panier Famille' contenant : Le 'Menu Classique' + un Café + une autre PetiteFrite.
   - Affichez le détail du 'Panier Famille' et son prix total final en un seul appel.

Résumé des Patterns Structure

Nous avons vu comment :

1. **Adapter** : Faire rentrer un carré dans un rond (connecter deux interfaces incompatibles).

2. **Decorator** : Ajouter des couches de vêtements (fonctionnalités) dynamiquement.

3. **Composite** : Créer des structures en arbre (le tout et la partie sont traités pareil).

Patterns de comportement (Behavioral)

Philosophie : L'Art de la Communication

Jusqu'ici, nous avons été très statiques :

- **Création :** "Comment je fais naître mes objets ?" (Factory, Singleton...)
- **Structure :** "Comment j'assemble mes objets ?" (Composite, Decorator...)

Maintenant, la question est : **"Comment mes objets discutent-ils entre eux et qui est responsable de quoi ?"**

Dans une application complexe, il ne suffit pas d'avoir de belles classes. Il faut qu'elles interagissent. Le risque ? Le **Couplage Fort de Communication**. Si l'objet A appelle directement l'objet B, qui appelle C, qui rappelle A... vous obtenez un plat de spaghettis indémêlable. Impossible de modifier A sans casser B et C.

Le but des Patterns de Comportement est de :

1. **Fluidifier la communication :** Les objets peuvent s'envoyer des messages sans forcément connaître l'identité précise du destinataire.
2. **Isoler les algorithmes :** Séparer la logique métier (le "comment on fait") de la classe qui l'utilise.
3. **Gérer le flux de contrôle :** Définir clairement qui décide de l'étape suivante.

Analogie : Imaginez une salle de marché boursier.

- Sans pattern : Tout le monde crie vers tout le monde. C'est le chaos.
- Avec pattern : Il y a des règles. "Si le prix monte, je lève la main", "Si je veux acheter, je remplis ce formulaire papier (Command)".

Patterns de comportement (Behavioral)

Les "Big Three" que nous allons étudier

Le catalogue du *Gang of Four* en contient 11, mais pour ce cours, nous allons nous concentrer sur les 3 plus vitaux, ceux que vous croiserez dans presque tous les frameworks modernes (Qt, Angular, .NET, moteurs de jeux).

1. **Observer (L'Observateur) :**

   - ***Le concept*** *:* "Ne m'appelle pas, je t'appellerai."
   - ***L'usage*** *:* Quand une donnée change, tout le monde est mis au courant automatiquement (ex: Excel met à jour le graphique quand vous changez une case).

2. **Strategy (La Stratégie) :**

   - ***Le concept*** *:* Le caméléon algorithmique.
   - ***L'usage*** *:* Changer la façon dont un objet travaille *pendant* l'exécution (ex: Un GPS qui bascule du calcul "Plus rapide" au calcul "Plus court" ou "Sans péage" en un clic).

3. **Command (La Commande) :**

   - ***Le concept*** *:* Transformer une action en un objet.
   - ***L'usage*** *:* Permet de mettre des actions en file d'attente, de les planifier, et surtout de faire le fameux **Ctrl+Z (Undo)**.

Pattern Observer

Observer (L'Observateur)

1. Le Problème : "C'est quand qu'on arrive ?"
Imaginez une station météo qui mesure la température. Imaginez que vous avez 3 écrans différents pour afficher cette température (un sur votre téléphone, un sur le mur du salon, un sur votre montre).

L'approche naïve (Le "Polling")
Vos 3 écrans demandent à la station toutes les 10 millisecondes : *"La température a changé ? La température a changé ?"*

- **Résultat :** La station est surchargée, le processeur chauffe, et 99% des appels sont inutiles car la température ne change pas si vite.

L'approche Observer (Le "Push")
On inverse le contrôle. Les écrans disent à la station : *"Tiens, voici mon numéro. Appelle-moi quand tu as du nouveau."* Quand la température change, la station envoie le message à tout le monde d'un coup.

2. La Structure

Ce pattern définit une relation **"Un-à-Plusieurs"**.

- **Le Sujet (Subject/Observable) :** C'est la donnée, la célébrité. Il tient une liste de ses fans.
- **L'Observateur (Observer) :** C'est le fan. Il s'inscrit sur la liste et attend la notification.

3. Implémentation en C++

Le défi en C++ est de gérer la liste des abonnés proprement.

Pattern Observer

🎯 Le Concept du Pattern Observer

Le pattern Observer implémente une relation **1-à-N** où :

- **1 Sujet** (StationMeteo) notifie automatiquement
- **N Observateurs** (EcranTelephone, EcranMur) quand son état change

C'est comme un **système d'abonnement** à des notifications !

🏗️ Les 4 Composants

1. **Interface Observer (IObserver)**
2. **Le Sujet Observable (StationMeteo)**

- **Maintient** une liste d'abonnés (std::vector<IObserver*>)
- **Gère** les inscriptions (`attach()`) et désinscriptions (`detach()`)
- **Notifie** tous les observateurs quand son état change (notify())

```cpp
// --- 1. L'INTERFACE OBSERVER ---
// Tous ceux qui veulent écouter la station météo doivent implémenter ça.
class IObserver {
public:
    virtual void update(float temperature) = 0;
    virtual ~IObserver() = default;
};
```

```cpp
// --- 2. LE SUJET (La Station) ---
class StationMeteo {
private:
    float temperature;
    // La liste de contacts (carnet d'adresses)
    std::vector<IObserver*> observateurs;

public:
    // Méthode pour s'abonner (Subscribe)
    void attach(IObserver* obs) {
        observateurs.push_back(obs);
    }

    // Méthode pour se désabonner (Unsubscribe)
    void detach(IObserver* obs) {
        // Astuce C++ pour supprimer un élément d'un vecteur
        observateurs.erase(
            std::remove(observateurs.begin(), observateurs.end(), obs),
            observateurs.end()
        );
    }

    // La méthode magique qui prévient tout le monde
    void notify() {
        for (IObserver* obs : observateurs) {
            obs->update(temperature);
        }
    }

    // Méthode métier qui déclenche le processus
    void setTemperature(float temp) {
        std::cout << "\n--- La station mesure une nouvelle temperature : "
        << temp << "C ---" << std::endl;
        this->temperature = temp;
        notify(); // Hop, on prévient tout le monde !
    }
};
```

Pattern Observer

3. Les Observateurs Concrets

- EcranTelephone : Affiche une notification mobile
- EcranMur : Met à jour l'affichage mural

4. Le Client : Orchestre les abonnements

🔄 Flux d'Exécution

✅ Avantages du Pattern

1. **Faible couplage** : La station ne connaît pas les types concrets d'écrans
2. **Extensibilité** : Facile d'ajouter de nouveaux types d'observateurs
3. **Dynamisme** : Abonnement/désabonnement à l'exécution
4. **Principe ouvert/fermé** : Ouvert à l'extension, fermé à la modification

```cpp
// --- 3. LES OBSERVATEURS CONCRETS ---
class EcranTelephone : public IObserver {
public:
    void update(float temperature) override {
        std::cout << ">> Telephone : Nouvelle notif ! Il fait "
        << temperature << " degres." << std::endl;
    }
};

class EcranMur : public IObserver {
public:
    void update(float temperature) override {
        std::cout << ">> Ecran Mur : Mise a jour affichage... [ "
        << temperature << "C ]" << std::endl;
    }
};
```

```cpp
// --- 4. CODE CLIENT ---
int main() {
    StationMeteo station;

    EcranTelephone monTel;
    EcranMur monMur;

    // 1. Inscription
    station.attach(&monTel);
    station.attach(&monMur);

    // 2. Événement 1
    station.setTemperature(25.5);

    // 3. Désinscription dynamique
    std::cout << "\n(Le telephone se deconnecte du reseau...)" << std::endl;
    station.detach(&monTel);

    // 4. Événement 2
    station.setTemperature(26.0); // Seul le mur reçoit l'info

    return 0;
}
```

Exercice : Le Backend de YouTube

Contexte & Défi : Vous développez le module de notification d'une plateforme vidéo. Lorsqu'un créateur de contenu (YouTuber) publie une nouvelle vidéo, tous ses abonnés doivent être notifiés instantanément. Cependant, les abonnés peuvent se désabonner à tout moment (ils ne doivent plus rien recevoir).

Votre mission : Implémentez ce système avec le pattern **Observer**.

Cahier des charges technique :

1. **L'Observateur (Interface)** : Créez une classe abstraite `IAbonne`.
   - Elle doit avoir une méthode `reagir(std::string nomChaine, std::string titreVideo)`.
2. **Le Sujet (La classe concrète)** : Créez une classe `ChaineYouTube`.
   - Elle possède un nom (ex: "JoueurDuGrenier").
   - Elle gère une liste d'abonnés.
   - Elle a les méthodes `sAbonner(IAbonne* a)` et `seDesabonner(IAbonne* a)`.
   - Elle a une méthode `posterVideo(std::string titre)` qui déclenche la notification générale.
3. **Les Observateurs Concrets** : Créez une classe `Utilisateur`.
   - Un utilisateur a un prénom.
   - Quand il reçoit une notif (`reagir`), il affiche sur la console : " *[Prénom] a vu la notif de [Chaine] : 'Titre de la vidéo' ".*

Main :

1. Créez une chaîne "Mister Beast".
2. Créez 3 utilisateurs (Alice, Bob, Charlie).
3. Alice et Bob s'abonnent.
4. Mister Beast poste "J'offre une île !". (Alice et Bob reçoivent la notif).
5. Charlie s'abonne. Bob se désabonne (trop de clickbait).
6. Mister Beast poste "Je mange un burger en or". (Alice et Charlie reçoivent la notif).

Pattern Strategy

C'est le pattern "Caméléon".

Si vous avez compris le principe du polymorphisme, vous allez trouver ce pattern très naturel. C'est l'arme absolue contre les blocs `if / else if / else` ou les `switch / case` qui font 200 lignes.

Strategy (La Stratégie)

1. Le Problème : L'Enfer des `if / else`

Imaginez que vous codez un GPS. Au début, vous ne gérez que la voiture.

```
void calculerItineraire() {
    // Algorithme complexe pour la route...
}
```

Puis on vous demande d'ajouter les piétons, puis les vélos, puis les transports en commun. Votre classe `GPS` va commencer à ressembler à ça :

```
void calculerItineraire(String mode) {
    if (mode == "VOITURE") {
        // 50 lignes de code pour la route
    } else if (mode == "PIETON") {
        // 40 lignes de code pour les trottoirs
    } else if (mode == "VELO") {
        // 45 lignes pour les pistes cyclables
    }
    // ... et ça continue
}
```

Le souci :

1. Votre classe `GPS` devient énorme.
2. Si vous voulez modifier l'algo "Vélo", vous risquez de casser l'algo "Piéton" par erreur en touchant au fichier.
3. C'est difficile de changer de comportement *pendant* l'exécution proprement.

2. La Solution Strategy

On prend chaque algorithme (chaque bloc du `if`), et on l'enferme dans sa propre classe. Le `GPS` (le Contexte) ne contiendra plus la logique, mais juste un pointeur vers l'algorithme actuel.

Pattern Strategy

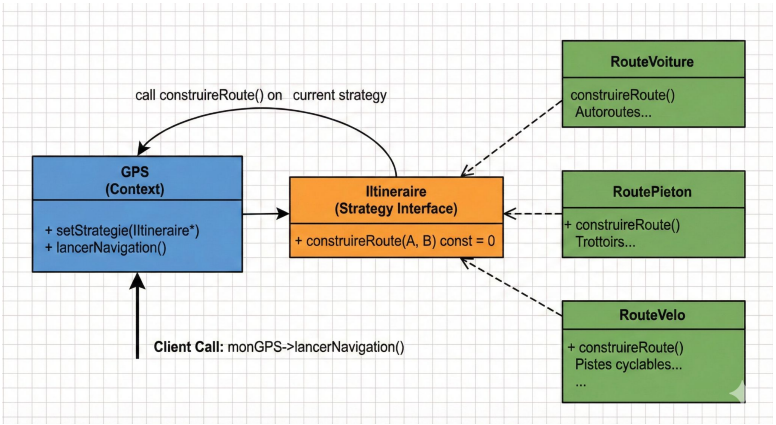
L'interface STRATEGY (IItineraire)
Elle définit "Ce qu'on veut faire" (aller d'un point A à B), mais pas "Comment on le fait".

```cpp
class IItineraire {
public:
    virtual void construireRoute(const std::string& A, const std::string& B) const = 0;
    virtual ~IItineraire() = default;
};
```

Les stratégies concrètes
Elles implémentent IItineraire, on peut en ajouter sans toucher au code de GPS

```cpp
class RouteVoiture : public IItineraire {
public:
    void construireRoute(const std::string& A, const std::string& B) const override {
        std::cout << "Itineraire Voiture : Autoroutes et nationales entre "
        << A << " et " << B << "." << std::endl;
    }
};

class RoutePieton : public IItineraire {
public:
    void construireRoute(const std::string& A, const std::string& B) const override {
        std::cout << "Itineraire Pieton : Trottoirs et parcs entre "
        << A << " et " << B << "." << std::endl;
    }
};
```

Le contexte (GPS)
Il possède une stratégie mais ne sait pas laquelle
On peut changer la stratégie à la volée




```cpp
class GPS {
private:
    // Le GPS possède une stratégie, mais il ne sait pas laquelle (polymorphisme)
    std::unique_ptr<IItineraire> strategieActuelle;

public:
    // On peut définir une stratégie par défaut à la construction
    GPS(std::unique_ptr<IItineraire> strategie) : strategieActuelle(std::move(strategie)) {}

    // LE POINT CLÉ : On peut changer de cerveau à la volée !
    void setStrategie(std::unique_ptr<IItineraire> nouvelleStrategie) {
        std::cout << "\n>> Changement de mode de transport..." << std::endl;
        strategieActuelle = std::move(nouvelleStrategie);
    }

    void lancerNavigation(std::string depart, std::string arrivee) {
        if (strategieActuelle) {
            strategieActuelle->construireRoute(depart, arrivee);
        }
    }
};
```

Pattern Strategy

Implémentation en C++ (Le Navigateur)

Nous allons utiliser `std::unique_ptr` pour gérer la stratégie courante, car le GPS *possède* sa stratégie active.

Analyse et Avantages

1. **Open/Closed Principle (Respecté) :** Si demain on veut un routage par avion, vous créez une classe `RouteAvion`. Vous ne touchez pas une seule ligne de la classe `GPS`.
2. **Séparation des préoccupations :** La classe `GPS` gère l'interface utilisateur ou les coordonnées. La classe `RouteVoiture` gère l'algorithme A* ou Dijkstra. Chacun son métier.

Composition vs Héritage :

- *Approche Héritage (Mauvaise)* : Faire `class GPSVoiture`, `class GPSVelo`. Impossible de changer de mode sans détruire l'objet GPS et en recréer un autre.
- *Approche Strategy (Bonne)* : Le GPS *a* une stratégie. On peut la changer.

```cpp
// --- 4. CODE CLIENT ---
int main() {
    // 1. On démarre en Voiture
    std::unique_ptr<GPS> monGPS = std::make_unique<GPS>(std::make_unique<RouteVoiture>());
    monGPS->lancerNavigation("Paris", "Toulouse");

    // 2. On arrive en ville, on se gare, on passe en mode Piéton
    // Notez qu'on garde le MÊME objet GPS, on change juste son "cerveau"
    monGPS->setStrategie(std::make_unique<RoutePieton>());
    monGPS->lancerNavigation("Gare Matabiau", "Place du Capitole");

    // 3. On trouve un vélo libre-service
    monGPS->setStrategie(std::make_unique<RouteVelo>());
    monGPS->lancerNavigation("Place du Capitole", "Stadium de Toulouse");

    return 0;
}
```

Dijkstra** et **A (prononcez "A-Star") sont les deux algorithmes les plus célèbres au monde pour résoudre le problème du "Plus Court Chemin" (Shortest Path) dans un graphe.*

Exercice : Le Maître d'Armes

Strategy

Contexte & Défi

Vous codez un personnage de jeu vidéo (un Chevalier). Ce chevalier peut ramasser différentes armes au sol. Quand le joueur appuie sur le bouton "X", de chevalier", le chevalier attaque. L'animation et les dégâts dépendent entièrement de l'arme qu'il tient re qu'il tient en main à ce moment-là.

Le Défi :

La classe `Chevalier` ne doit contenir **aucun** `if` ou `switch` pour vérifier le type d'arme. Elle doit juste dire "Arme, active-toi !".

Scénario du Main

1. Créez "Arthur".
2. Donnez-lui une Épée -> Attaquez.
3. L'ennemi est loin ! Changez pour l'Arc -> Attaquez.
4. L'ennemi est insensible au physique ! Changez pour le Sceptre -> Attaquez.

Cahier des charges technique

1. **L'Interface Stratégie (`IArme`) :**
   - Une méthode virtuelle pure `void utiliser()`.
2. **Stratégies Concrètes :**
   - `Epee` : Affiche *"Vlan ! Un coup d'épée tranchant."*
   - `Arc` : Affiche *"Pan ! Une flèche en pleine tête."*
   - `SceptreMagique` : Affiche *"Boum ! Une boule de feu."*
3. **Le Contexte (`Chevalier`) :**
   - Il possède un nom.
   - Il possède une `arme` (via `std::unique_ptr`).
   - Il a une méthode `equiper(std::unique_ptr<IArme> nouvelleArme)` pour changer de stratégie dynamiquement.
   - Il a une méthode `attaquer()` qui délègue le travail à l'arme.
4. **Main :**
   - Créez "Arthur".
   - Donnez-lui une Épée -> Attaquez.
   - L'ennemi est loin ! Changez pour l'Arc -> Attaquez.
   - L'ennemi est insensible au physique ! Changez pour le Sceptre -> Attaquez.

Pattern Command

C'est le dernier pattern "Must Have" de notre formation, et il est important car il débloque une fonctionnalité que tous les utilisateurs réclament : le **Ctrl+Z (Annuler/Rétablir)**.

Command (La Commande)

1. Le Problème : "Je regrette mon action !"

Imaginez que vous codez un éditeur de texte ou une télécommande domotique. L'utilisateur clique sur un bouton "Allumer la lumière". Le code naïf serait : `bouton.onClick = [] () { lumiere.allumer(); };`

C'est direct et rapide. Mais... Si l'utilisateur dit "Oups, je voulais pas", comment faites-vous ? Une fois la fonction exécutée, l'ordinateur a "oublié" ce qui s'est passé. Il ne sait pas qu'avant, la lumière était éteinte.

La Solution : L'Action en Objet

Le pattern **Command** propose de ne pas exécuter l'action tout de suite, mais de l'encapsuler dans un objet. Au lieu d'appeler une méthode, on crée un objet `CommandeAllumer`.

L'analogie du Restaurant :

1. **Le Client (Vous)** veut manger.
2. Vous ne courez pas en cuisine pour hurler au chef. Vous donnez votre ordre (**Commande**) au Serveur (**Invoker**).
3. Le Serveur note la commande sur un papier. Il ne sait pas cuisiner, il sait juste transmettre le papier.
4. Il donne le papier au Chef (**Receiver**) qui cuisine.

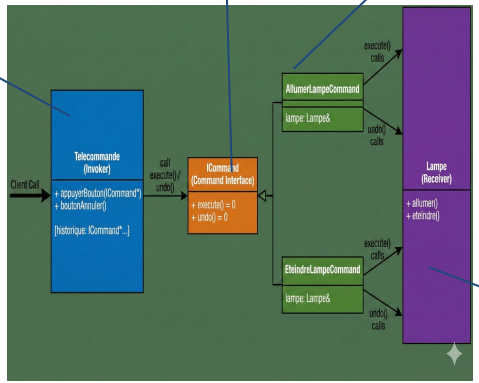
L'avantage ? Le serveur garde le papier (la commande). À la fin du repas, il peut reprendre tous les papiers pour calculer l'addition (Historique).

Pattern Command


```cpp
// --- 4. INVOKER (La Télécommande) ---
class Telecommande {
private:
    // Historique des commandes exécutées
    std::vector<std::shared_ptr<ICommand>> historique;

public:
    // Le bouton ne sait pas ce qu'il fait, il exécute juste une commande qu'on lui donne
    void appuyerBouton(std::shared_ptr<ICommand> cmd) {
        cmd->execute();
        // On stocke la commande dans l'historique
        historique.push_back(cmd);
    }

    void boutonAnnuler() {
        if (!historique.empty()) {
            std::cout << ">> [CTRL+Z] Annulation de la derniere action..." << std::endl;
            // On récupère la dernière commande
            std::shared_ptr<ICommand> derniereCmd = historique.back();
            // On l'annule
            derniereCmd->undo();
            // On la retire de l'historique
            historique.pop_back();
        } else {
            std::cout << "Rien a annuler !" << std::endl;
        }
    }
};
```

```cpp
// --- CODE CLIENT ---
int main() {
    // Le matériel
    Lampe salon("Salon");
    Lampe cuisine("Cuisine");

    // Les commandes (les objets actions)
    auto cmdSalonOn = std::make_shared<AllumerLampeCommand>(salon);
    auto cmdCuisineOn = std::make_shared<AllumerLampeCommand>(cuisine);
    auto cmdSalonOff = std::make_shared<EteindreLampeCommand>(salon);

    // L'utilisateur (La télécommande)
    Telecommande remote;

    // Scénario
    remote.appuyerBouton(cmdSalonOn);   // Allume Salon
    remote.appuyerBouton(cmdCuisineOn); // Allume Cuisine
    remote.appuyerBouton(cmdSalonOff);  // Eteint Salon

    std::cout << "\n--- Oups, je voulais garder le salon allume ! ---" << std::endl;
    remote.boutonAnnuler(); // Annule "Eteindre Salon" -> Rallume Salon

    std::cout << "\n--- Et je veux eteindre la cuisine aussi finalement ---" << std::endl;
    remote.boutonAnnuler(); // Annule "Allumer Cuisine" -> Eteint Cuisine

    return 0;
}
```

```cpp
// --- 2. INTERFACE COMMAND ---
class ICommand {
public:
    virtual void execute() = 0;
    virtual void undo() = 0; // La clé du Ctrl+Z
    virtual ~ICommand() = default;
};
```




```cpp
// --- 3. CONCRETE COMMANDS ---

// Commande pour allumer
class AllumerLampeCommand : public ICommand {
private:
    // La commande doit savoir QUI elle contrôle
    Lampe& lampe;

public:
    AllumerLampeCommand(Lampe& l) : lampe(l) {}

    void execute() override {
        lampe.allumer();
    }

    void undo() override {
        // Pour annuler "Allumer", il faut faire "Eteindre"
        lampe.eteindre();
    }
};
```

```cpp
// --- 1. RECEIVER (Celui qui fait le vrai travail) ---
class Lampe {
private:
    std::string lieu;
public:
    Lampe(std::string l) : lieu(l) {}

    void allumer() {
        std::cout << "La lampe du " << lieu << " s'allume." << std::endl;
    }

    void eteindre() {
        std::cout << "La lampe du " << lieu << " s'eteint." << std::endl;
    }
};
```

Pourquoi c'est génial ?

1. **Découplage total :** La classe `Telecommande` ne connait pas la classe `Lampe`. Elle connait juste `ICommand`. Vous pouvez utiliser cette télécommande pour piloter un Garage, une TV ou un Drone, tant que vous créez la Commande associée.
2. **Files d'attente (Queue) :** Comme les commandes sont des objets, vous pouvez les mettre dans une liste "À faire plus tard" (Batch processing).
3. **Undo/Redo :** C'est le seul moyen propre d'implémenter un historique infini.

Exercice : Le "Instant Replay" de la Manette Pro

Contexte & Défi

Contexte : Vous développez le pilote d'une manette de jeu "Pro Gamer" avec une fonctionnalité tueuse : le bouton "Instant Replay". Quand le joueur appuie dessus, le jeu doit rejouer exactement les 10 dernières secondes d'actions, à l'endroit ou à l'envers.

Le défi : La manette ne sait pas ce que font les boutons. Elle sait juste que le bouton 'A' déclenche une commande, le bouton 'B' une autre, etc.

Scénario du Main

1. Créez le **Robot "Z-42"**.
2. Configurez la manette : **'A'** pour sauter, **'B'** pour tirer.
3. Le joueur fait une séquence : **Saut, Tir, Saut, Saut, Tir.**
4. Affichez "**--- REPLAY AVANT ---**" et lancez le replay forward.
   Affichez "**--- REPLAY AVANT ---**" et lancez le replay forward.
5. Affichez "**--- REPLAY ARRIERE (REWIND) ---**" et lancez le replay backward.

Cahier des charges technique

1. **Receiver (Le Héros) :** Créez une classe **Robot**.
   - Méthodes : **sauter()**, **atterrir()**, **tirerLaser()**, **absorberLaser()**.
2. **Interface Command :** **ICommand** avec **execute()** et **undo()**.
3. **Concrete Commands :**
   - **CommandeSaut** : **execute** appelle **sauter**, **undo** appelle **atterrir**.
   - **CommandeTir** : **execute** appelle **tirerLaser**, **undo** appelle **absorberLaser**.
   - Ces commandes doivent connaître le Robot qu'elles contrôlent.
4. **Invoker (La Manette) :** Créez une classe **ManettePro**.
   - Elle doit stocker un "mapping" des touches (**std::map<char, std::shared_ptr<ICommand>>**).
   - Elle doit stocker l'historique des commandes jouées (**std::vector<std::shared_ptr<ICommand>>**).
   - Méthode **assignerTouche(char touche, std::shared_ptr<ICommand> cmd)**.
   - Méthode **appuyerSur(char touche)** : Exécute et ajoute à l'historique.
   - Méthode **replayForward()** : Rejoue tout l'historique.
   - Méthode **replayBackward()** : Rejoue tout l'historique à l'envers.

Indice C++ : Pour parcourir un vecteur à l'envers, regardez du côté des itérateurs inversés **rbegin()** et **rend()**.

🏆 Projet Final : La Maison Domotique (Smart Home 2.0)

Votre Mission (Cahier des Charges)

Refondez cette architecture en utilisant au moins **3 Design Patterns** parmi ceux étudiés.

C'est l'heure de vérité!

Vous avez les outils. Vous avez la théorie...
Maintenant, nous allons affronter un monstre :
Le Code Hérité (Legacy Code).

Fichier : general/SmartHome-Legacy.cpp

1. Structure (Composite) :

- Créez une interface commune **IDevice**.
- Permettez de créer des objets simples (Lampe, Thermostat, DetecteurFumee) ET des conteneurs (**Piece**).
- On doit pouvoir appeler **allumer()** sur une **Piece**, et cela doit allumer tous les appareils de la pièce.

2. Création (Factory Method) :

- Supprimez les **new** dans le **main**.
- Créez une classe **DeviceFactory** qui prend une string ("hifi", "light", "heat") et retourne le bon objet encapsulé dans un **std::shared_ptr**.

3. Communication (Observer) :

- Le **DetecteurFumee** est un **Sujet**.
- Les **Volets** sont des **Observateurs**.
- Scénario : Quand le détecteur de fumée se déclenche (**alerteIncendie()**), tous les volets de la maison doivent s'ouvrir automatiquement pour l'évacuation.

Bonus (Si vous êtes rapides) :

- Utilisez un **Singleton** pour un **JournalDeBord** (Logger) qui écrit dans la console chaque action effectuée dans la maison.