Cours 06 - C++ pour les mathématiques appliquées

Patrons de Conception
Design Patterns en C++

Rappel : Conception Objet

Analyse (QUOI)

- Diagrammes cas d'utilisation
- Diagrammes de séquence
- Diagrammes états-transitions
- Diagrammes de classes d'analyse

Spécification des besoins

- Vue métier
- Concepts du domaine

Conception (COMMENT)

Architecture (structure globale)

- Couches
- MVC
- Paquetages

Conception détaillée

- Classes logicielles
- Interactions entre objets
- Comportement des objets

Affectation des responsabilités

Responsabilités de connaissance

- Données encapsulées
- Objets connexes
- Données dérivées (calculables)

Exemple

```
class Emprunt {
    Utilisateur* utilisateur;
    Exemplaire* exemplaire;
    Date dateRetourPrevue;
    // Connaissance des données
};
```

Responsabilités de comportement

- Action (calcul, création)
- Délégation (action sur autre objet)
- Coordination entre objets

Exemple

```
class Emprunt {
    double calculerPenalite() {
        // Action : calcul
    }

    void retourner() {
        exemplaire->liberer();
        utilisateur->libererQuota();
        // Coordination
    }
};
```

Rappel : Principes de conception

Couplage faible

- Minimiser les dépendances
- Classes indépendantes
- Facile à réutiliser

Cohésion forte

- Services liés au sein d'une classe
- Objectif clair
- Facile à comprendre

Principe de l'Expert

- Tâche T réalisée par l'objet qui a l'information nécessaire

Principe du Créateur

- Classe B crée A si B contient A ou a l'info pour créer A

Principe du Contrôleur

- Événements système → classes de contrôle

Cohérence entre diagrammes

Vigilance requise

- L'ensemble des diagrammes forme un tout
- Correspond à échéance proche à la structure du code

Vérifications

- Cohérence entre tous les diagrammes
- Pas de contradiction
- Complétude

Exemples de cohérence

Les objets des diagrammes :

- Sont instances de classes du diagramme de classes
- Ont les attributs déclarés
- Ont les méthodes nécessaires

Les relations :

- Associations cohérentes
- Multiplicités respectées
- Navigation possible

Patrons de Conception

Solutions éprouvées aux problèmes récurrents

Pourquoi les Design Patterns ?

Constat

- La conception de logiciels est difficile
- Conception objet (extensibilité, réutilisabilité)
- Nécessite des développeurs expérimentés

Problème

- Comment transmettre l'expérience ?
- Comment capitaliser le savoir-faire ?
- Comment éviter de réinventer la roue ?

Solution : Design Patterns

Objectifs

- Recueillir l'expérience et l'expertise
- Transmettre pour réutilisation
- Créer un vocabulaire commun

Bénéfices

- Solutions testées et éprouvées
- Communication facilitée entre développeurs
- Maintenance et évolution simplifiées

Historique des Design Patterns

Origines

1977 - Christopher Alexander

- Architecte en bâtiment
- "A Pattern Language"
- Patterns pour l'architecture

1987 - Beck et Cunningham

- Introduction en conception objet
- Conférence OOPSLA

Développement

1991 - Erich Gamma

- Thèse de doctorat
- Création de JUnit, Eclipse JDT
- Aujourd'hui chez Microsoft

1995 - "Gang of Four" (GoF)

- Gamma, Helm, Johnson, Vlissides
- "Design Patterns: Elements of Reusable OO Software"
- 23 patterns fondamentaux

Bibliographie

Livres de référence

- **Design Patterns** - Gamma, Helm, Johnson, Vlissides (GoF), 1995
  - Ouvrage fondateur, 23 patterns classiques
- **Head First Design Patterns** - Freeman, Freeman, Sierra, Bates, 2011
  - Approche pédagogique et visuelle
- **La programmation orientée objet** - Hugues Bersini, 2013
  - Perspective francophone
- **Object-Oriented Modeling and Design with UML** - Blaha, Rumbaugh, 2007
  - Lien entre UML et patterns

Classification des Design Patterns

Créationnels

Création d'objets

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Structurels

Composition d'objets

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Proxy

Comportementaux

Interactions entre objets

- Observer
- Strategy
- Command
- State
- Iterator
- Template Method

Patrons Créationnels

Contrôler la création d’objets

Singleton - Principe

Objectif

- Garantir qu'une classe n'a qu'une seule instance
- Fournir un point d'accès global à cette instance

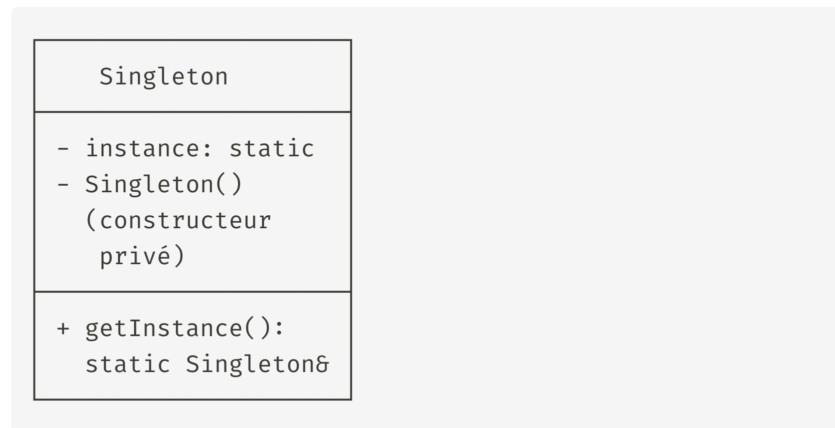
Cas d'usage

- Gestionnaire de configuration
- Pool de connexions
- Logger
- Gestionnaire de ressources partagées

Contraintes

- Une seule instance
- Création paresseuse (lazy)
- Thread-safe en C++11+

Diagramme



Caractéristiques

- Constructeur privé
- Copie interdite
- Déplacement interdit
- Méthode statique getInstance()

Singleton - Implémentation C++

Version moderne (C++11+)

```
class Singleton {
private:
    // Constructeur privé
    Singleton() {
        // Initialisation
    }

    // Interdire copie
    Singleton(const Singleton&) = delete;

    // Interdire déplacement
    Singleton(Singleton&&) = delete;

    // Interdire affectation
    Singleton& operator=(const Singleton&)
        = delete;

public:
    static Singleton& getInstance() {
        static Singleton instance;
        return instance;
    }
};
```

Utilisation

```
// Accès à l'instance unique
Singleton& s1 = Singleton::getInstance();
Singleton& s2 = Singleton::getInstance();

// s1 et s2 sont la même instance
assert(&s1 == &s2);
```

Avantages C++11

- Thread-safe automatique
- Construction paresseuse
- Destruction automatique

Attention

- Ordre de destruction non garanti
- Difficile à tester (état global)
- Dépendance cachée

Factory Method - Principe

Objectif

- Créer des objets sans spécifier leur classe exacte
- Déléguer la décision de création aux sous-classes

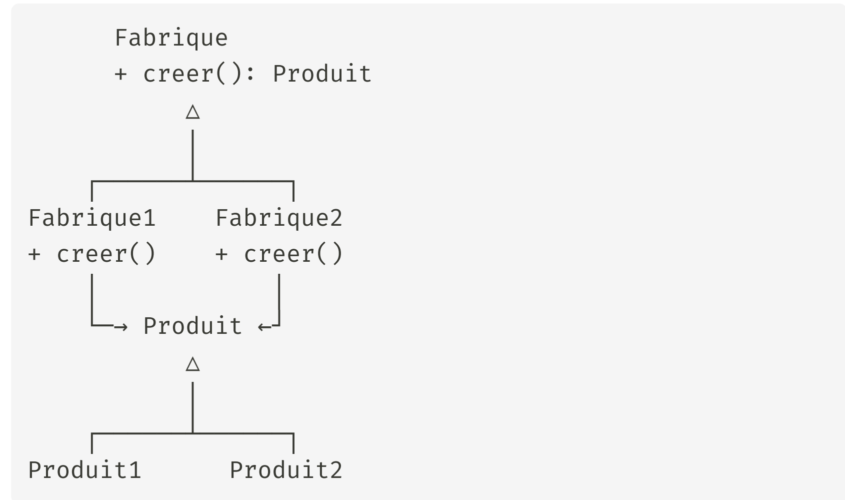
Motivation

- Client ne connaît que l'interface
- Classe concrète choisie à l'exécution
- Pas d'appel direct au constructeur

Structure

- Interface `Produit`
- Classes `ProduitConcret`
- Interface `Fabrique` avec méthode fabrique
- Classes `FabriqueConcrète`

Diagramme



Code relatif à la création

- Dans `ProduitConcret`
- Dans `FabriqueConcrète`
- Client utilise interfaces

Factory Method - Exemple Point

Problème

Comment créer un Point en coordonnées :

- Cartésiennes (x, y)
- Polaires (r, θ)

Sans surcharge ambiguë ?

```
// ❌ Ambigu - même signature
Point(float x, float y);  // Cartésien
Point(float r, float theta); // Polaire
```

Solution : Factory Methods

```
enum class PointType {
    cartesian, polar
};

class Point {
    float m_x, m_y;
    PointType m_type;

    // Constructeur privé
    Point(float x, float y, PointType t)
        : m_x{x}, m_y{y}, m_type{t} {}

public:
    // Factory methods statiques
    static Point NewCartesian(
        float x, float y) {
        return {x, y,
                PointType::cartesian};
    }

    static Point NewPolar(
        float r, float theta) {
        return {r * cos(theta),
```

Factory Method - Utilisation

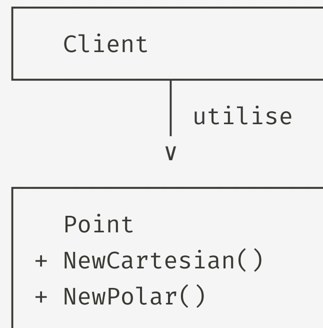
Utilisation claire

```cpp
// Création en cartésien
Point p1 = Point::NewCartesian(3.0, 4.0);

// Création en polaire
Point p2 = Point::NewPolar(5.0, 0.927);

// Pas d'ambiguïté !
```

Avantages

- Noms explicites (NewCartesian vs NewPolar)
- Pas de confusion de paramètres
- Extensible (ajouter NewCylindrical, etc.)

Séparation des responsabilités



Alternative : classe Fabrique séparée

```cpp
class PointFactory {
public:
    static Point cartesian(float x, float y);
    static Point polar(float r, float t);
};
```

Abstract Factory - Principe

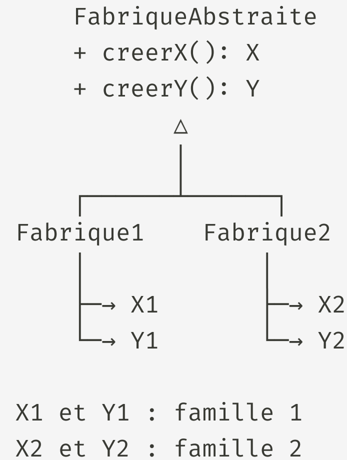
Objectif

- Créer des familles d'objets interdépendants
- Sans spécifier leurs classes concrètes

Contexte

- Client veut produire objets X et Y (interfaces)
- Plusieurs implémentations : familles
- X1 fonctionne avec Y1, X2 avec Y2
- Client reçoit une Fabrique (interface)
- Fabrique produit X et Y d'une même famille

Diagramme



Abstract Factory - Exemple

Interface Point polymorphe

```
struct Point {
    virtual ~Point() = default;

    // Factory methods virtuelles
    virtual unique_ptr<Point>
        create() = 0;

    virtual unique_ptr<Point>
        clone() = 0;
};
```

Implémentations concrètes

```
struct Point2D : Point {
    unique_ptr<Point> create() {
        return make_unique<Point2D>();
    }

    unique_ptr<Point> clone() {
        return make_unique<Point2D>(*this);
    }
};

struct Point3D : Point {
    unique_ptr<Point> create() {
        return make_unique<Point3D>();
    }

    unique_ptr<Point> clone() {
        return make_unique<Point3D>(*this);
    }
};
```

Abstract Factory - Utilisation

Code client générique

```
void process(Point* p) {
    // Créer nouveau de même type
    auto nouveau = p->create();

    // Dupliquer
    auto copie = p->clone();

    // Client ne connaît pas
    // le type concret !
}

int main() {
    Point* p2d = new Point2D();
    Point* p3d = new Point3D();

    process(p2d); // Crée Point2D
    process(p3d); // Crée Point3D
}
```

Évaluation

Avantages

- Changement de famille facile
- Ajout de famille facile
- Utilisation cohérente des produits
- Client découplé des classes concrètes

Inconvénients

- Ajouter un produit = modifier toutes les fabriques
- Complexité accrue
- Plus de classes

Quand utiliser ?

- Système indépendant de la création
- Plusieurs familles de produits
- Produits d’une famille utilisés ensemble

Patrons Structurels

Composer des objets en structures

Composite - Motivation

Problème

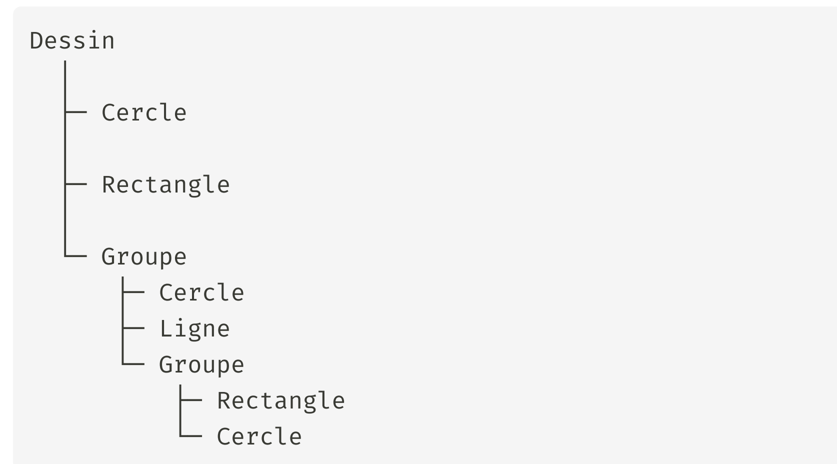
Représenter des figures géométriques :

- Figures simples (Cercle, Rectangle)
- Figures composées (Groupe de figures)

Besoins

- Traitement uniforme
- Manipulation récursive
- Calculs sur compositions

Exemple : Dessin



Comment calculer l'aire totale ? Comment déplacer tout le dessin ?

Composite - Principe

Objectif

- Composer objets en arborescences
- Traiter uniformément objets simples et composés

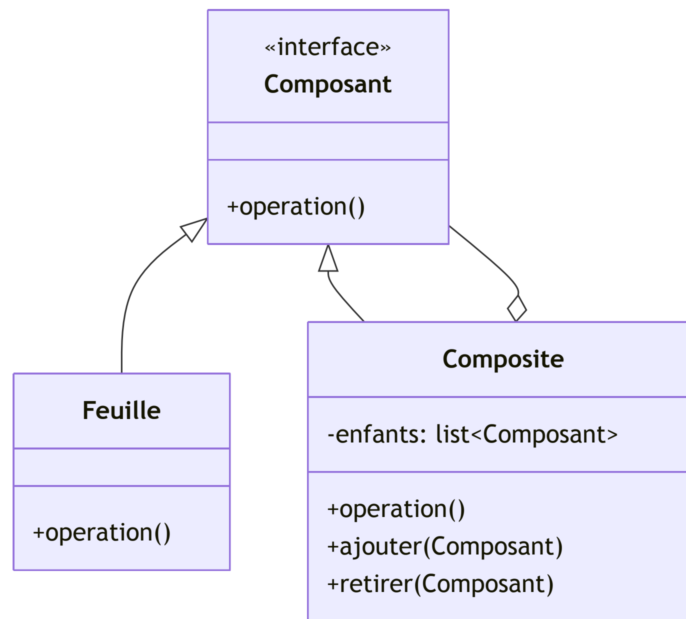
Solution

- Interface `Composant`
- Classe `Feuille` (élément simple)
- Classe `Composite` (contient des Composants)
- Client manipule via interface

Clé

- `Composite` contient des `Composant`
- Récursivité naturelle

Diagramme



Implémentation

Composite - Exemple Géométrique

Interface Forme

```cpp
class Forme {
public:
    virtual ~Forme() = default;
    virtual double aire() const = 0;
    virtual void deplacer(int dx, int dy) = 0;
};
```

Feuilles

```cpp
class Cercle : public Forme {
    double rayon;
    int x, y;
public:
    double aire() const override {
        return M_PI * rayon * rayon;
    }

    void deplacer(int dx, int dy) override {
        x += dx;
        y += dy;
    }
};
```

Composite

```cpp
class Groupe : public Forme {
    vector<shared_ptr<Forme>> formes;

public:
    void ajouter(shared_ptr<Forme> f) {
        formes.push_back(f);
    }

    double aire() const override {
        double total = 0;
        for(const auto& f : formes) {
            total += f->aire();
        }
        return total;
    }

    void deplacer(int dx, int dy) override {
        for(auto& f : formes) {
            f->deplacer(dx, dy);
        }
    }
};
```

Composite - Évaluation

Avantages

Simplicité client

- Traitement uniforme simple/composé
- Pas de test de type

Extensibilité

- Ajout facile de nouveaux types
- Structure récursive naturelle

Flexibilité

- Compositions arbitraires
- Réutilisation

Inconvénients

Trop général

- Difficile de restreindre composants
- Vérifications à l'exécution

Conception

- Où mettre opérations de gestion ?
- Feuille ou Composite ?

Quand utiliser ?

- Structure arborescente
- Traitement uniforme nécessaire
- Exemples : GUI, système fichiers, expressions mathématiques

Décorateur - Principe

Objectif

- Ajouter dynamiquement des responsabilités à un objet
- Alternative flexible à l'héritage

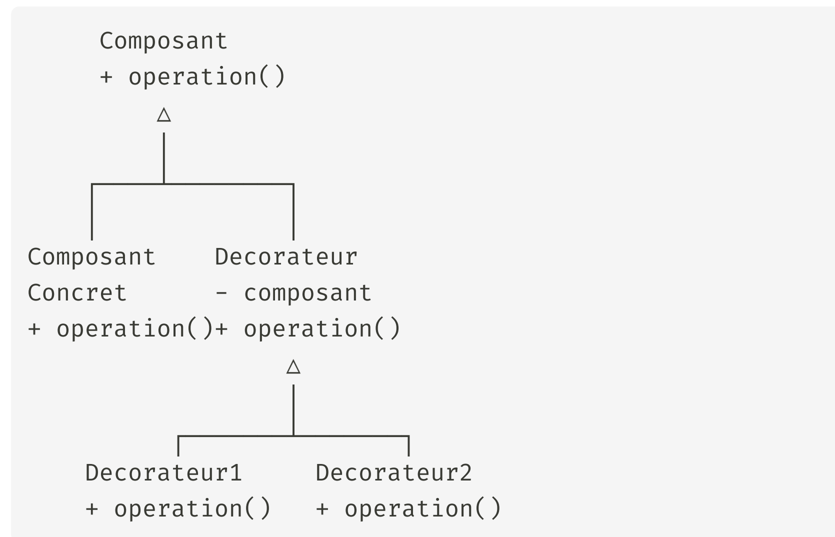
Problème héritage

- Explosion combinatoire de classes
- Exemple : Fenêtre + Bordure + Scrollbar
- 2^n classes pour n options

Solution Décorateur

- Enveloppe un objet
- Même interface
- Ajoute comportement

Diagramme



Délégation + Ajout

```
void Decorateur::operation() {
    composant->operation();
    // + comportement additionnel
}
```

Décorateur - Exemple Boisson

Interface et base

```cpp
class Boisson {
public:
    virtual ~Boisson() = default;
    virtual string description() = 0;
    virtual double prix() = 0;
};

class Cafe : public Boisson {
public:
    string description() override {
        return "Café";
    }
    double prix() override {
        return 2.0;
    }
};
```

Décorateurs

```cpp
class Decorateur : public Boisson {
protected:
    shared_ptr<Boisson> boisson;
public:
    Decorateur(shared_ptr<Boisson> b)
        : boisson(b) {}
};

class Lait : public Decorateur {
public:
    using Decorateur::Decorateur;

    string description() override {
        return boisson->description()
               + " + Lait";
    }
    double prix() override {
        return boisson->prix() + 0.5;
    }
};

class Sucre : public Decorateur {
public:
```

Décorateur - Utilisation

Composition dynamique

```cpp
// Café simple
auto b1 = make_shared<Cafe>();
cout << b1->description()
     << " : " << b1->prix() << "€\n";
// Café : 2.0€

// Café + Lait
auto b2 = make_shared<Lait>(
    make_shared<Cafe>()
);
cout << b2->description()
     << " : " << b2->prix() << "€\n";
// Café + Lait : 2.5€

// Café + Lait + Sucre
auto b3 = make_shared<Sucre>(
    make_shared<Lait>(
        make_shared<Cafe>()
    )
);
cout << b3->description()
     << " : " << b3->prix() << "€\n";
// Café + Lait + Sucre : 2.7€
```

Avantages

Flexibilité

- Combinaisons à l'exécution
- Pas d'explosion de classes

Extensibilité

- Nouveaux décorateurs indépendants
- Respect Open/Closed

vs Héritage

- Héritage : CafeLait, CafeSucre, CafeLaitSucre…
- Décorateur : 2 classes (Lait, Sucre)

Inconvénients

- Beaucoup de petits objets
- Identité d'objet complexe
- Ordre des décorateurs important

Patrons Comportementaux

Interactions et responsabilités entre objets

Observer - Principe

Objectif

- Notifier automatiquement des objets des changements d'état
- Dépendances 1-n entre objets

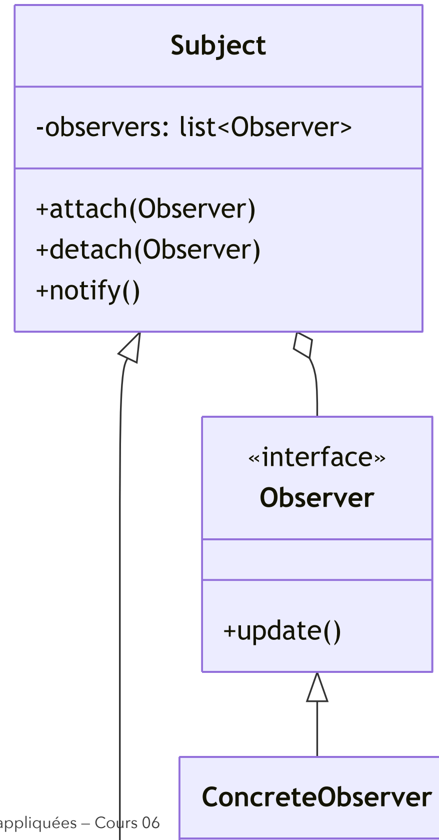
Motivation

- Maintenir cohérence entre objets liés
- Sans couplage fort
- Notifications automatiques

Participants

- **Subject** (Sujet) : objet observé
- **Observer** (Observateur) : notifié des changements

Diagramme



Observer - Exemple Bibliothèque (1/2)

Subject : Catalogue

```cpp
class Observer {
public:
    virtual void update() = 0;
};

class Catalogue {
    vector<Document*> documents;
    vector<Observer*> observers;

public:
    void attach(Observer* obs) {
        observers.push_back(obs);
    }

    void detach(Observer* obs) {
        auto it = find(
            observers.begin(),
            observers.end(), obs);
        if (it != observers.end())
            observers.erase(it);
    }

    void ajouterDocument(Document* doc) {
```

Observers : Vues

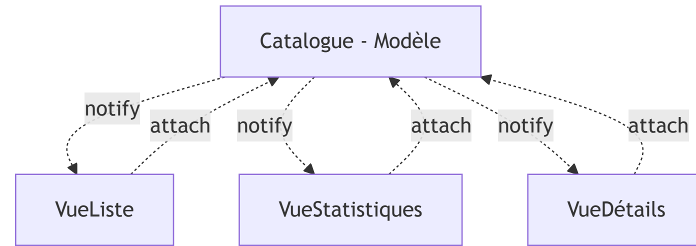
```cpp
class VueListe : public Observer {
    Catalogue* catalogue;

public:
    VueListe(Catalogue* c)
        : catalogue(c) {
        catalogue->attach(this);
    }

    ~VueListe() {
        catalogue->detach(this);
    }

    void update() override {
        cout << "VueListe: Mise à jour\n";
        auto docs = catalogue->getDocuments();
        // Rafraîchir affichage liste
        afficherListe(docs);
    }
};

class VueStatistiques : public Observer {
    Catalogue* catalogue;
```

Observer - Exemple Bibliothèque (2/2)

Utilisation

```
Catalogue catalogue;

// Créer les vues
VueListe vueListe(&catalogue);
VueStatistiques vueStats(&catalogue);

// Ajouter un document
// → Notifie automatiquement
//   les deux vues
catalogue.ajouterDocument(
    new Livre("Design Patterns")
);

// Console :
// VueListe: Mise à jour
// VueStats: Recalcul

// Retirer un document
// → Notifie à nouveau
catalogue.retirerDocument(livre);

// Console :
// VueListe: Mise à jour
```

Architecture MVC



Avantages

- Catalogue découplé des vues
- Ajout facile de nouvelles vues
- Synchronisation automatique
- Pattern Observer = cœur du MVC

Observer - Évaluation

Avantages

Couplage abstrait

- Subject connaît interface Observer
- Pas les observateurs concrets

Support broadcast

- Notification à tous les intéressés
- Nombre variable d'observateurs

Réutilisabilité

- Subject et Observer indépendants

Inconvénients

Mises à jour en cascade

- Peut être coûteux
- Ordre non garanti

Gestion mémoire

- Fuites si oubli unsubscribe
- Utiliser weak_ptr en C++

Quand utiliser ?

- MVC (Modèle notifie Vues)
- Système d'événements
- Binding de données
- Exemple : bibliothèque (Catalogue notifie Vues)

Stratégie, Commande, État - Vue d'ensemble

Points communs

Ces trois patterns partagent :

- Associer objets à des traitements
- Hiérarchie de classes de traitements
- Accès uniforme aux traitements
- Découplage données/traitement
- Séparation hiérarchie traitement et classes utilisatrices

Différences

Stratégie

- Implémenter une opération de plusieurs façons
- Choisir à l'exécution laquelle utiliser

Commande

- Associer même commande à plusieurs objets
- Gérer séquences et historiques

État

- Plusieurs opérations associées à un état
- État qui évolue

Stratégie - Principe

Objectif

- Encapsuler des algorithmes dans objets
- Les rendre dynamiquement interchangeables

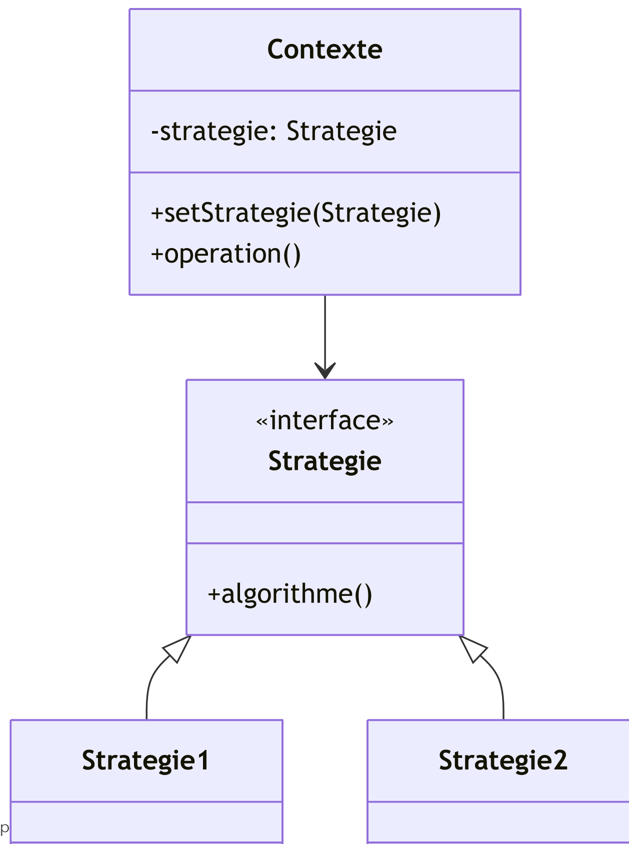
Motivation

- Plusieurs algorithmes pour un problème
- Choisir à l'exécution lequel utiliser
- Éviter if/switch pour sélection

Participants

- **Contexte** : utilise une Stratégie
- **Stratégie** : interface commune
- **StratégieConcrète** : implémente algorithme

Diagramme



Stratégie - Exemple Bibliothèque

Stratégies de recherche

```cpp
class SearchStrategy {
public:
    virtual ~SearchStrategy() = default;
    virtual vector<Document*> search(
        const vector<Document*>& docs,
        const string& critere) = 0;
};

class SearchByTitle : public SearchStrategy {
public:
    vector<Document*> search(
        const vector<Document*>& docs,
        const string& titre) override {

        vector<Document*> resultats;
        for(auto doc : docs) {
            if (doc->getTitre()
                  .find(titre) != string::npos) {
                resultats.push_back(doc);
            }
        }
        return resultats;
    }
```

Contexte et utilisation

```cpp
class Catalogue {
    vector<Document*> documents;
    unique_ptr<SearchStrategy> strategy;

public:
    void setSearchStrategy(
        unique_ptr<SearchStrategy> s) {
        strategy = move(s);
    }

    vector<Document*> rechercher(
        const string& critere) {
        return strategy->search(
            documents, critere);
    }
};

// Utilisation
Catalogue cat;

// Recherche par titre
cat.setSearchStrategy(
    make_unique<SearchByTitle>());
```

Stratégie - Exemple Format Liste

Interface et stratégies

```
enum class Format { Markdown, Html };

struct ListStrategy {
    virtual ~ListStrategy() = default;
    virtual void start(ostringstream&) {}
    virtual void end(ostringstream&) {}
    virtual void add_item(
        ostringstream&, string&) = 0;
};

struct MarkdownListStrategy
    : ListStrategy {
    void add_item(ostringstream& oss,
                  string& item) override {
        oss << " - " << item << endl;
    }
};

struct HtmlListStrategy : ListStrategy {
    void start(ostringstream& oss) override {
        oss << "<ul>" << endl;
    }
    void end(ostringstream& oss) override {
```

Contexte

```
struct TextProcessor {
    ostringstream m_oss;
    unique_ptr<ListStrategy> m_strategy;

    void setFormat(Format f) {
        switch (f) {
        case Format::Markdown:
            m_strategy =
                make_unique<
                    MarkdownListStrategy>();
            break;
        case Format::Html:
            m_strategy =
                make_unique<
                    HtmlListStrategy>();
            break;
        }
    }

    void appendList(vector<string>& items) {
        m_strategy->start(m_oss);
        for (auto& item: items)
            m_strategy->add_item(m_oss, item);
```

Stratégie - Utilisation et Évaluation

Utilisation

```
TextProcessor tp;
vector<string> items =
    {"foo", "bar", "baz"};

// Markdown
tp.setFormat(Format::Markdown);
tp.appendList(items);
cout << tp.str();
// - foo
// - bar
// - baz

// HTML
tp.setFormat(Format::Html);
tp.appendList(items);
cout << tp.str();
// <ul>
//   <li>foo</li>
//   <li>bar</li>
//   <li>baz</li>
// </ul>
```

Avantages

- Élimination conditionnels
- Changement dynamique
- Réutilisation algorithmes
- Alternatives claires

Inconvénients

- Plus d'objets
- Client doit connaître stratégies
- Overhead indirection

Quand utiliser ?

- Variantes d'algorithme
- Beaucoup de conditionnels
- Clients ne doivent pas connaître détails

Exemples

Commande - Principe

Objectif

- Encapsuler une requête comme objet
- Paramétrer clients avec différentes requêtes
- Supporter annulation (undo)

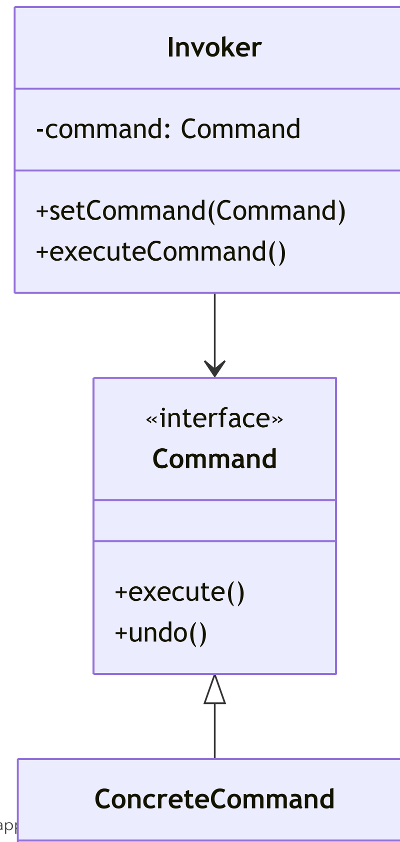
Motivation

- Partager actions (menu + bouton)
- Stocker historique
- Logger opérations
- Transactions

Participants

- **Command** : interface execute()
- **ConcreteCommand** : implémente
- **Receiver** : sait effectuer action
- **Invoker** : demande exécution

Diagramme



Commande - Exemple Lumière

Interface et Receiver

```cpp
class Command {
public:
    virtual ~Command() = default;
    virtual void execute() = 0;
    virtual void undo() = 0;
};

// Receiver
class Light {
public:
    void on() {
        cout << "Lumière allumée\n";
    }
    void off() {
        cout << "Lumière éteinte\n";
    }
};
```

Commandes concrètes

```cpp
class LightOnCommand : public Command {
    Light* light;
public:
    LightOnCommand(Light* l) : light(l) {}

    void execute() override {
        light->on();
    }
    void undo() override {
        light->off();
    }
};

class LightOffCommand : public Command {
    Light* light;
public:
    LightOffCommand(Light* l) : light(l) {}

    void execute() override {
        light->off();
    }
    void undo() override {
        light->on();
```

Commande - Invoker avec Undo

Télécommande avec historique

```cpp
class RemoteControl {
    vector<unique_ptr<Command>> history;

public:
    void executeCommand(
        unique_ptr<Command> cmd) {

        cmd->execute();
        history.push_back(move(cmd));
    }

    void undo() {
        if (!history.empty()) {
            history.back()->undo();
            history.pop_back();
        }
    }
};
```

Utilisation

```cpp
Light light;
RemoteControl remote;

// Allumer
remote.executeCommand(
    make_unique<LightOnCommand>(&light)
);
// Lumière allumée

// Éteindre
remote.executeCommand(
    make_unique<LightOffCommand>(&light)
);
// Lumière éteinte

// Annuler (rallume)
remote.undo();
// Lumière allumée

// Annuler (éteint)
remote.undo();
// Lumière éteinte
```

Commande - Macro-Commande

Composition de commandes

```cpp
class MacroCommand : public Command {
    vector<unique_ptr<Command>> commands;

public:
    void add(unique_ptr<Command> cmd) {
        commands.push_back(move(cmd));
    }

    void execute() override {
        for (auto& cmd : commands) {
            cmd->execute();
        }
    }

    void undo() override {
        // Annuler dans l'ordre inverse
        for (auto it = commands.rbegin();
             it != commands.rend(); ++it) {
            (*it)->undo();
        }
    }
};
```

Utilisation

```cpp
// Macro : tout éteindre
auto macroOff = make_unique<MacroCommand>();

macroOff->add(
    make_unique<LightOffCommand>(&light1)
);
macroOff->add(
    make_unique<LightOffCommand>(&light2)
);
macroOff->add(
    make_unique<LightOffCommand>(&light3)
);

remote.executeCommand(move(macroOff));
// Éteint toutes les lumières

remote.undo();
// Rallume toutes les lumières
```

Avantages

- Séquences complexes

Commande - Évaluation

Avantages

Découplage

- Invoker indépendant de Receiver
- Ajout facile de nouvelles commandes

Flexibilité

- Commandes en objets
- Stockage, transmission
- Composition (macro-commandes)

Fonctionnalités

- Undo/Redo facile
- Logging
- Transactions

Inconvénients

Complexité

- Beaucoup de classes
- Une par commande

Mémoire

- Historique peut être lourd

Quand utiliser ?

- Paramétrer actions
- File d'attente d'opérations
- Undo/Redo nécessaire
- Transactions

Exemples

- Éditeurs (undo/redo)
- Interfaces graphiques

Combiner les Design Patterns

Patterns qui se complètent

Singleton + Observer

```cpp
class Catalogue {  // Singleton + Subject
    static Catalogue& getInstance() {
        static Catalogue instance;
        return instance;
    }

    void notify() {
        for(auto obs : observers)
            obs->update();
    }
};
```

Composite + Decorator

```cpp
// Composite de formes
class Groupe : public Forme {
    vector<shared_ptr<Forme>> formes;
};

// Décoré avec bordure
```

Factory + Strategy

```cpp
class SearchStrategyFactory {
    static unique_ptr<SearchStrategy>
    create(const string& type) {
        if (type == "titre")
            return make_unique<SearchByTitle>();
        if (type == "auteur")
            return make_unique<SearchByAuthor>();
        // ...
    }
};
```

Command + Composite (Macro)

```cpp
class MacroCommand : public Command {
    vector<unique_ptr<Command>> commands;

    void execute() override {
        for(auto& cmd : commands)
            cmd->execute();
    }
};
```

Résumé des Design Patterns

Créationnels

Singleton

- Une seule instance
- Accès global

Factory Method

- Création sans classe exacte
- Nommage explicite

Abstract Factory

- Familles d'objets
- Cohérence garantie

Structurels

Composite

- Arborescence d'objets
- Traitement uniforme

Decorator

- Ajout responsabilités
- Alternative héritage

Comportementaux

Observer

- Notifications automatiques
- Dépendances 1-n

Strategy

- Algorithmes interchangeables
- Choix à l'exécution

Command

- Requêtes objets
- Undo/Redo

Conclusion

Ce que nous avons vu

Patterns créationnels

- Contrôle de création
- Flexibilité d'instanciation

Patterns structurels

- Composition d'objets
- Relations entre classes

Patterns comportementaux

- Communication entre objets
- Répartition responsabilités

Principes clés

Conception pour l'interface

- Pas pour l'implémentation

Favoriser composition

- Sur héritage

Programmer pour extension

- Principe Open/Closed

Points importants

- Vocabulaire commun
- Solutions éprouvées
- Pas des recettes magiques
- À adapter au contexte

La prochaine fois

Templates et Métaprogrammation en C++

Templates de fonctions

- Généricité
- Spécialisation

Templates de classes

- Conteneurs génériques
- Paramètres multiples

Métaprogrammation

- Calculs à la compilation
- SFINAE
- Concepts C++20

STL en profondeur

- Algorithmes génériques
- Itérateurs

Questions ?

N'hésitez pas à poser vos questions sur les Design Patterns

Prochaine session : Templates et Métaprogrammation C++