Contents

Documents de réflexion pour le TP4 de C++

Vocabulaire

Classe abstraite :

Une classe abstraite est une classe qui contient au moins une méthode pure virtuelle.

Comportement :

Le comportement fait référence à l'ensemble des actions ou des fonctionnalités qu'un objet peut réaliser en réponse à une méthode ou à une demande particulière. Le comportement est souvent défini par les méthodes exposées par un composant, et il peut être modifié ou étendu dynamiquement par les décorateurs sans changer la structure du code de base.

Composition :

La composition est un principe de conception en programmation orientée objet qui consiste à créer des objets en combinant d'autres objets, permettant ainsi de construire des comportements complexes à partir de composants simples. Contrairement à l'héritage, où une classe dérive d'une autre, la composition favorise une relation de type "a un", où les objets contiennent d'autres objets pour réutiliser des fonctionnalités. En d'autres termes, pour la composition, une classe contient une ou plusieurs instances d'autres classes.

Composant :

Un composant fait généralement référence à l'interface ou à la classe de base abstraite qui définit les fonctionnalités communes que toutes les classes décorées ou décoratrices doivent implémenter. Le composant joue un rôle central, car c'est lui qui sert de point d'entrée pour la manipulation des objets dans une hiérarchie de décorateurs.

- Le composant est l'interface commune à la fois pour les classes de base et les décorateurs. Il définit les méthodes que tous les objets devront implémenter.
- Le composant décoré (classe concrète) est une implémentation de cette interface qui fournit des fonctionnalités de base.
- Les décorateurs sont des classes qui héritent de l'interface composant et qui enveloppent un autre composant pour ajouter dynamiquement de nouvelles fonctionnalités tout en conservant la même interface.

Implémenter :

Implémenter signifie fournir le code réel pour les méthodes définies dans une interface ou une classe abstraite.

Interface :

Une interface est une sorte de contrat qui définit un ensemble de méthodes que les classes doivent implémenter.
Elle ne contient pas de code d'implémentation, mais seulement les signatures des méthodes.
En d'autres termes, une interface dit "ce que l'on doit faire" sans dire "comment le faire".
En C++, une interface est souvent représentée par une classe abstraite.

Méthode pure virtuelle

Une méthode pure virtuelle est une méthode qui n'a pas d'implémentation dans la classe de base et doit être implémentée par les classes dérivées.

Polymorphisme:

- Le polymorphisme *statique* se base sur la surcharge et les templates, est *résolu à la compilation* (polymorphisme de compilation).
- Le polymorphisme *dynamique*, basé sur les méthodes virtuelles, est *résolu à l'exécution* (polymorphisme d'exécution), permettant à des objets dérivés de redéfinir des comportements spécifiques .

Dans la pratique, ce concept permet de traiter des objets de types différents de manière uniforme tout en permettant à chaque type de définir son propre comportement spécifique.

Responsabilité :

Dans le contexte du `design pattern Decorator`, le terme "responsabilité" fait référence aux fonctionnalités ou aux comportements spécifiques que l'objet doit accomplir.
En d'autres termes, une responsabilité est une tâche ou une action que l'objet est censé réaliser.

Design pattern "Decorator"

Le design pattern "Decorator" est un modèle de conception qui permet d'ajouter des responsabilités à un objet de manière dynamique. En d'autres termes, il permet d'ajouter des fonctionnalités supplémentaires à un objet sans modifier sa structure.

Pour utiliser le design pattern Decorator, on doit suivre quelques étapes clés.
Guide simple pour comprendre comment l'implémenter :

- **Définir l'interface de base** : Crée une interface ou une classe abstraite qui définit les méthodes que tous les objets (de base et décorateurs) doivent implémenter.
- **Créer l'objet de base** : Implémente une classe concrète qui représente l'objet de base. Cette classe doit implémenter l'interface de base.
- **Créer les décorateurs** : Crée une classe abstraite de décorateur qui implémente l'interface de base et contient une référence à un objet de base ou à un autre décorateur. Implémente des classes concrètes de décorateurs qui ajoutent des fonctionnalités supplémentaires. Chaque décorateur doit appeler la méthode de l'objet enveloppé et ajouter sa propre fonctionnalité.

Classes décoratrices et décorées

Dans le `design pattern Decorator` en C++, les classes décoratrices et décorées jouent des rôles spécifiques mais étroitement liés. Voici les principales différences entre ces deux types de classes :

1. **Rôle dans la structure**.
   - *Classe décorée* (Composant ou Composant de base) : C'est la classe de base (abstraite ou concrète) qui définit l'interface ou les fonctionnalités de base. Elle représente l'objet de base que l'on veut étendre.
   - *Classe décoratrice (ou Décorateur)* : C'est une classe qui "enrobe" (wrap) la classe décorée et ajoute dynamiquement de nouvelles fonctionnalités sans modifier la classe d'origine.
2. **Fonctionnalités**.
   - *Classe décorée* : Fournit une implémentation de base des fonctionnalités. Elle ne sait rien de ses décorateurs. C'est généralement une implémentation simple et directe de l'interface.
   - *Classe décoratrice* : Étend ou modifie dynamiquement le comportement de la classe décorée en ajoutant des fonctionnalités supplémentaires avant ou après avoir délégué l'appel à la classe décorée. Elle conserve une référence à un objet de type composant (souvent via un pointeur ou une référence à la classe de base) qu'elle enveloppe.
3. **Héritage**.
   - *Classe décorée* : Elle peut être une classe concrète ou abstraite. Si elle est abstraite, elle définit simplement l'interface que les décorateurs et les autres composants doivent implémenter.
   - *Classe décoratrice* : Elle hérite de la classe composant (la classe décorée) pour garantir qu'elle possède la même interface et peut être utilisée de manière interchangeable avec la classe décorée.
4. **Composition**
   - *Classe décorée* : Représente l'objet que l'on veut étendre, sans nécessairement connaître l'existence des décorateurs.
   - *Classe décoratrice* : Elle contient généralement un membre privé ou protégé qui est un pointeur ou une référence à un objet de type composant, c'est-à-dire l'objet qu'elle décore. Elle délègue certaines tâches à cet objet tout en ajoutant ses propres comportements.

Des exemples

Exemple simple

```cpp
#include <iostream>
#include <string>

// Interface
class Animal {
public:
    virtual std::string make_sound() const = 0; // Méthode pure virtuelle
    virtual ~Animal() {} // Destructeur virtuel
};

// Classe dérivée qui implémente l'interface
class Dog : public Animal {
public:
    std::string make_sound() const override {
        return "Woof!";
    }
};

// Classe dérivée qui implémente l'interface
class Cat : public Animal {
public:
    std::string make_sound() const override {
        return "Meow!";
    }
};
```

Dans cet exemple :

- Dog et Cat sont des classes dérivées qui implémentent l'interface Animal.
- Chaque classe fournit sa propre implémentation de la méthode make_sound.

Maintenant, On peut utiliser ces classes comme suit :

```cpp
int main() {
    Dog dog;
    Cat cat;

    std::cout << dog.make_sound() << std::endl;  // Affiche : Woof!
    std::cout << cat.make_sound() << std::endl;  // Affiche : Meow!

    return 0;
}
```

Exemple plus complet

```cpp
#include <iostream>
#include <memory>

// Interface composant de base
class Component {
public:
    virtual ~Component() = default;
    virtual void operation() const = 0;
};

// Composant concret
class ConcreteComponent : public Component {
```

```
public:
    void operation() const override {
        std::cout << "ConcreteComponent: base behavior" << std::endl;
    }
};

// Classe décoratrice abstraite
class Decorator : public Component {
protected:
    std::unique_ptr<Component> component;
public:
    Decorator(std::unique_ptr<Component> comp) : component(std::move(comp)) {}
    void operation() const override {
        if (component) {
            component->operation();  // Délégation au composant décoré
        }
    }
};

// Décorateur concret ajoutant un comportement avant
class ConcreteDecoratorA : public Decorator {
public:
    ConcreteDecoratorA(std::unique_ptr<Component> comp) : Decorator(std::move(comp)) {}

    void operation() const override {
        std::cout << "ConcreteDecoratorA: additional behavior before" << std::endl;
        Decorator::operation();  // Appelle le comportement de base
    }
};

// Décorateur concret ajoutant un comportement après
class ConcreteDecoratorB : public Decorator {
public:
    ConcreteDecoratorB(std::unique_ptr<Component> comp) : Decorator(std::move(comp)) {}

    void operation() const override {
        Decorator::operation();  // Appelle le comportement de base
        std::cout << "ConcreteDecoratorB: additional behavior after" << std::endl;
    }
};

int main() {
    // Composant de base
    std::unique_ptr<Component> component = std::make_unique<ConcreteComponent>();

    // Ajout de décorateurs
    std::unique_ptr<Component> decoratedComponent = std::make_unique<ConcreteDecoratorA>(std::move(component));
    decoratedComponent = std::make_unique<ConcreteDecoratorB>(std::move(decoratedComponent));

    // Exécution du comportement
    decoratedComponent->operation();  // Comportement de base avec ajouts avant et après

    return 0;
}
```

Polymorphisme

Le polymorphisme est un concept fondamental de la programmation orientée objet qui permet à des objets de types différents de réagir de manière différente à un même appel de méthode ou d'opération, en fonction de leur type réel. En d'autres termes, le polymorphisme permet à une même interface ou à un même nom de méthode d'avoir plusieurs comportements en fonction de l'objet sur lequel il est invoqué.

Polymorphisme statique (Surcharge et Templates)

Le polymorphisme statique est déterminé à la compilation. Cela signifie que le compilateur détermine, au moment de la compilation, quelle méthode ou quel opérateur utiliser. Deux mécanismes principaux en C++ permettent d'implémenter ce type de polymorphisme : la surcharge (overloading) et les templates.

a) Surcharge (Overloading)

La surcharge consiste à utiliser plusieurs fonctions ou opérateurs avec le même nom, mais des signatures différentes. Le compilateur choisit la fonction ou l'opérateur à appeler en fonction du type et du nombre de paramètres.

Exemple de surcharge de méthode :

```
class Math {
public:
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
};
```

Ici, la fonction `add` est surchargée : la version à utiliser sera déterminée par les types d'arguments passés.

Précautions d'utilisation des surcharge :

- Clarté :
  Évitez de surcharger des opérateurs d'une manière qui pourrait prêter à confusion. Le comportement des opérateurs doit rester intuitif pour les utilisateurs de la classe.
- Consistance :
  Assurez-vous que la surcharge d'opérateurs respecte la logique du langage. Par exemple, si vous surchargez ==, vous devez également penser à surcharger != pour que les comparaisons soient cohérentes.
- Performance :
  Certaines surcharges, en particulier celles des opérateurs comme « (pour les flux) ou les opérateurs arithmétiques, peuvent introduire des copies d'objets ou des calculs coûteux. Utilisez des références et des optimisations comme le move semantics (avec &&) si nécessaire.

b) Templates

(Que nous verrons prochainnement)

Les templates (gabarits) permettent d'écrire des fonctions ou des classes qui fonctionnent avec n'importe quel type sans avoir à les surcharger pour chaque type spécifique.

Exemple de fonction template :

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

Ici, la fonction `add` peut être utilisée avec des `int`, des `double`, ou n'importe quel autre type pour lequel l'opérateur + est défini.

Polymorphisme dynamique

(via des fonctions virtuelles)

Le polymorphisme dynamique repose sur l'héritage et les méthodes virtuelles. Il est déterminé à l'exécution et permet de changer le comportement d'une méthode selon le type réel de l'objet, même si on ne connaît que le type de base au moment de l'appel.

a) Méthodes virtuelles

Pour implémenter le polymorphisme dynamique en C++, on utilise le mot-clé virtual dans les classes de base. Cela permet à une méthode d'être redéfinie dans les classes dérivées et d'être appelée en fonction du type réel de l'objet, et non de son type déclaré.

Exemple :

```
class Animal {
public:
    virtual void makeSound() const {  // Méthode virtuelle
        std::cout << "Animal sound" << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() const override {  // Redéfinition
        std::cout << "Bark" << std::endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() const override {  // Redéfinition
        std::cout << "Meow" << std::endl;
    }
};

int main() {
    Animal* animal1 = new Dog();
    Animal* animal2 = new Cat();

    animal1->makeSound();  // Appelle la version de Dog : "Bark"
    animal2->makeSound();  // Appelle la version de Cat : "Meow"

    delete animal1;
    delete animal2;
}
```

La méthode `makeSound()` est déclarée virtuelle dans la classe de base `Animal`, ce qui permet à la méthode redéfinie dans `Dog` et `Cat` d'être appelée en fonction du type réel des objets (Dog ou Cat), *même si on les manipule à travers des pointeurs ou références de type* `Animal`.
C'est un exemple typique de polymorphisme dynamique : l'appel à la méthode dépend du type réel de l'objet (qui est déterminé à l'exécution).

Polymorphisme dans la pratique

Le polymorphisme est particulièrement utile lorsqu'on veut créer des interfaces communes pour différents types d'objets, tout en permettant à chaque type d'avoir son propre comportement.

Par exemple, dans une application de gestion de formes géométriques, on pourrait avoir une classe de base `Shape` avec une méthode virtuelle `draw()`. Ensuite, des classes dérivées comme `Circle`, `Square`, etc., redéfinissent cette méthode pour dessiner leurs propres formes spécifiques.
Le polymorphisme permet de manipuler toutes les formes via des références ou des pointeurs de type `Shape`, tout en exécutant les bonnes méthodes pour chaque forme à l'exécution.

Précautions et avantages

Avantages :

Flexibilité : Le polymorphisme rend le code plus flexible et extensible. Vous pouvez ajouter de nouveaux types sans modifier le code existant.

Réutilisabilité : En définissant des comportements communs dans une interface ou classe de base, vous pouvez réutiliser le code à travers de nombreuses sous-classes.
Simplicité : Vous manipulez des objets de types dérivés différents avec une seule interface, sans avoir besoin de savoir quel est leur type spécifique à la compilation.

Précautions :

Surcharge de performance : Le polymorphisme dynamique nécessite une résolution à l'exécution des appels de méthodes (via une table des méthodes virtuelles, ou vtable), ce qui peut entraîner un léger coût en termes de performance par rapport aux appels directs.

Complexité : Si le polymorphisme est utilisé de manière excessive ou mal géré, il peut rendre le code difficile à lire et à maintenir, surtout avec des hiérarchies de classes complexes.

Virtualité non nécessaire : Ne rendez pas toutes vos méthodes virtuelles par défaut. Cela peut introduire une surcharge inutile si vous n'avez pas réellement besoin de polymorphisme.

Héritage

L'héritage est un concept fondamental de la programmation orientée objet (POO) qui permet à une classe (appelée classe dérivée ou sous-classe) d'hériter des propriétés (attributs et méthodes) d'une autre classe (appelée classe de base ou superclasse). Cela favorise la réutilisation du code et la création de relations hiérarchiques entre les classes. Concepts clés de l'héritage

- Classe de base :
  La classe dont les propriétés et méthodes sont héritées. Elle peut être abstraite ou concrète.
- Classe dérivée :
  La classe qui hérite des propriétés et méthodes de la classe de base. Elle peut ajouter ses propres propriétés et méthodes ou redéfinir celles de la classe de base.
- Surcharge de méthodes :
  Une classe dérivée peut redéfinir une méthode de la classe de base pour fournir une implémentation spécifique.
- Accès aux membres :
  Les membres (attributs et méthodes) de la classe de base peuvent être accessibles dans la classe dérivée selon leur niveau d'accès (public, protected, private).

Types d'héritage

- Héritage simple : Une classe dérivée hérite d'une seule classe de base.
- Héritage multiple : Une classe dérivée hérite de plusieurs classes de base. (C++ le permet, mais cela peut rendre la conception plus complexe).
- Héritage hiérarchique :
  Une classe de base est héritée par plusieurs classes dérivées.
- Héritage multi-niveau :
  Une classe dérivée hérite d'une classe de base, et à son tour, elle est la classe de base d'une autre classe dérivée.

Voici un exemple simple illustrant l'héritage :

```
#include <iostream>

// Classe de base
class Animal {
```

```
public:
    void eat() {
        std::cout << "L'animal mange." << std::endl;
    }
};

// Classe dérivée
class Dog : public Animal {
public:
    void bark() {
        std::cout << "Le chien aboie." << std::endl;
    }
};

int main() {
    Dog myDog;
    myDog.eat(); // Appel de la méthode héritée de la classe Animal
    myDog.bark(); // Appel de la méthode de la classe Dog

    return 0;
}
```

Avantages de l'héritage

- Réutilisation du code :
  Permet de réutiliser des méthodes et des attributs existants, ce qui réduit la duplication de code.
- Organisation :
  Favorise une structure logique et hiérarchique des classes, facilitant ainsi la gestion et la compréhension du code.
- Polymorphisme :
  Permet d'utiliser des références de la classe de base pour traiter des objets de classes dérivées, ce qui permet d'écrire du code plus flexible et extensible.

Inconvénients de l'héritage

- Couplage :
  Une forte dépendance entre les classes peut rendre le code plus difficile à maintenir et à modifier.
- Complexité :
  L'héritage multiple peut conduire à des ambiguïtés et des complications, notamment avec le problème du "Diamond Problem".
- Difficulté de test :
  Les relations hiérarchiques peuvent rendre les tests unitaires plus complexes, car les modifications apportées à la classe de base peuvent avoir des effets en cascade sur les classes dérivées.

`override`

Ce terme est directement associé à la gestion de comportements dynamiques, d'héritage dans les classes.

Le mot-clé `override` en C++ est utilisé pour indiquer explicitement qu'une méthode dans une classe dérivée remplace ou redéfinit une méthode virtuelle de la classe de base. Il assure que la signature de la méthode dans la classe dérivée correspond à celle de la classe de base, et que la méthode redéfinie est effectivement virtuelle dans la classe de base.

Usage :
Redéfinition d'une méthode virtuelle dans une classe dérivée. Assure que la méthode correspond bien à la méthode virtuelle de la classe de base (protection contre les erreurs de signature).

Exemple :

```
class Base {
```

```
public:
    virtual void function() const {
        std::cout << "Base function" << std::endl;
    }
};

class Derived : public Base {
public:
    void function() const override {  // Indique que cette fonction redéfinit la méthode virtuelle de Base
        std::cout << "Derived function" << std::endl;
    }
};
```

Précautions d'utilisation `override` :

- Assurez-vous que la méthode dans la classe de base est virtuelle. Si vous essayez d'utiliser override sur une méthode qui n'est pas virtuelle dans la classe de base, cela entraînera une erreur de compilation.
- Signature exacte requise :
  Si la signature de la méthode dans la classe dérivée ne correspond pas exactement à celle de la méthode virtuelle de la classe de base, l'utilisation de override déclenchera une erreur. Cela permet de détecter des erreurs subtiles comme une signature différente (par exemple, un paramètre différent).

Le principe de composition versus l'héritage

Le principe de composition plutôt que de recourir à l'héritage est un principe fondamental dans la conception logicielle, souvent associé au paradigme de la programmation orientée objet (POO).

Ce principe recommande d'utiliser la composition (associer des objets entre eux) plutôt que l'héritage (étendre des classes) pour créer des systèmes flexibles et extensibles.

Pourquoi préférer la composition à l'héritage ?

L'héritage peut rendre un système rigide et difficile à maintenir, car les classes dérivées dépendent fortement des classes de base. Les classes dérivées héritent automatiquement de toutes les fonctionnalités de leurs classes parents, ce qui peut créer une forte liaison entre les deux et rendre le code moins flexible.

La composition, en revanche, permet de combiner des objets en fonction des comportements spécifiques que vous souhaitez leur attribuer. Cela favorise une approche plus modulaire où chaque composant peut être changé ou réutilisé indépendamment des autres. Héritage vs Composition

Héritage

L'héritage représente une relation hiérarchique où une classe dérivée hérite des propriétés et méthodes d'une classe de base. Par exemple, un Car est un type de Vehicle, donc Car peut hériter de Vehicle.

```
class Vehicle {
public:
    void startEngine() {
        std::cout << "Starting engine" << std::endl;
    }
};

class Car : public Vehicle {
public:
    void openTrunk() {
        std::cout << "Opening trunk" << std::endl;
    }
};
```

Composition

La composition, en revanche, permet de composer des objets avec d'autres objets, plutôt que d'hériter de leurs propriétés.

Par exemple, une Car a un Engine, plutôt que d'être un Vehicle.

Cela permet de mieux séparer les responsabilités et de rendre les composants interchangeables.

```
class Engine {
public:
    void start() {
        std::cout << "Engine starting..." << std::endl;
    }
};

class Car {
private:
    Engine engine;  // Composition : Car a un moteur
public:
    void startCar() {
        engine.start();  // Appelle le moteur pour démarrer la voiture
    }
};
```

Avantages de la Composition

- Flexibilité : Contrairement à l'héritage, la composition permet de changer dynamiquement les comportements en modifiant les composants utilisés. Vous pouvez facilement remplacer un composant sans affecter les autres parties du système.
- Faible couplage : Avec la composition, les classes sont moins dépendantes les unes des autres. Le système est donc plus modulaire et plus facile à maintenir.
- Réutilisabilité : Chaque composant peut être réutilisé dans différentes classes sans être lié à une hiérarchie spécifique.
- Extensibilité : En ajoutant ou en remplaçant des composants, il est plus facile d'ajouter de nouvelles fonctionnalités sans modifier le code existant.

Exemple concret : Decorator Pattern

Le Design Pattern Decorator est un excellent exemple d'utilisation de la composition pour étendre dynamiquement les fonctionnalités d'un objet sans utiliser l'héritage rigide.

Exemple sans décorateur (avec héritage) :

Si on utilise l'héritage pour ajouter des fonctionnalités supplémentaires à une classe, on se retrouve rapidement avec des sous-classes multiples, chacune représentant une combinaison possible de fonctionnalités. Cela devient difficile à maintenir et à étendre.

```
class Beverage {
public:
    virtual double cost() const = 0;
    virtual ~Beverage() {}
};

class Coffee : public Beverage {
public:
    double cost() const override {
        return 2.0;  // Prix d'un café
    }
};

class MilkCoffee : public Coffee {  // Héritage pour ajouter du lait
public:
```

```
    double cost() const override {
        return Coffee::cost() + 0.5;  // Prix avec supplément lait
    }
};
```

Ici, si vous souhaitez ajouter d'autres ingrédients comme du sucre, vous devez créer des sous-classes comme SugarMilkCoffee, WhippedCreamMilkCoffee, etc., ce qui rend l'héritage complexe et peu flexible.

Exemple avec composition :

Avec la composition et le Decorator, vous pouvez décorer dynamiquement un objet de base avec des fonctionnalités supplémentaires sans créer une explosion de sous-classes.

```
#include <iostream>

class Beverage {
public:
    virtual double cost() const = 0;
    virtual ~Beverage() {}
};

class Coffee : public Beverage {
public:
    double cost() const override {
        return 2.0;
    }
};

// Classe Decorator de base
class AddOnDecorator : public Beverage {
protected:
    Beverage* beverage; // Utilisation de pointeur brut
public:
    AddOnDecorator(Beverage* bev) : beverage(bev) {}
    virtual ~AddOnDecorator() {
        delete beverage; // Libération de la mémoire du composant décoré
    }
};

// Un décorateur pour ajouter du lait
class MilkDecorator : public AddOnDecorator {
public:
    MilkDecorator(Beverage* bev) : AddOnDecorator(bev) {}

    double cost() const override {
        return beverage->cost() + 0.5;  // Supplément lait
    }
};

// Un décorateur pour ajouter du sucre
class SugarDecorator : public AddOnDecorator {
public:
    SugarDecorator(Beverage* bev) : AddOnDecorator(bev) {}

    double cost() const override {
        return beverage->cost() + 0.2;  // Supplément sucre
    }
};

int main() {
```

```
    Beverage* coffee = new Coffee();  // Un café simple
    coffee = new MilkDecorator(coffee);  // Ajoute du lait
    coffee = new SugarDecorator(coffee);  // Ajoute du sucre

    std::cout << "Prix du café avec lait et sucre : " << coffee->cost() << std::endl;

    delete coffee; // Libération de la mémoire
}
```

Avec une approche par composition :

- Vous pouvez facilement ajouter ou enlever des fonctionnalités à un objet sans créer de nouvelles sous-classes.
- Vous pouvez changer les décorations à l'exécution, ce qui est impossible avec l'héritage classique.

Ici, vous créez dynamiquement un objet `Coffee` avec des décorations pour ajouter du lait et du sucre. Chaque décoration est une composition, ajoutant des fonctionnalités sans créer de nouvelles sous-classes. Conclusion

Le principe de composition plutôt que d'héritage est une bonne pratique de conception car :

- Il rend le code plus flexible et facile à maintenir.
- Il réduit le couplage entre les classes et permet de changer les comportements dynamiquement.
- Il favorise la réutilisabilité des composants.

L'héritage peut parfois être nécessaire, mais il doit être utilisé avec précaution et dans les bons contextes, là où une relation hiérarchique forte est réellement justifiée. Pour des comportements dynamiques et évolutifs, la composition est souvent une meilleure solution.

Quelques remarques

- Gestion de la mémoire :
  Lorsque vous utilisez des pointeurs bruts, vous devez gérer vous-même la mémoire. Cela signifie que vous devez appeler delete sur les objets lorsque vous avez terminé de les utiliser. Dans le code ci-dessus, le destructeur de AddOnDecorator libère également la mémoire du beverage décoré.
- Risques de fuites de mémoire :
  Si vous oubliez de libérer la mémoire ou si un chemin d'exécution ne parvient pas à atteindre la ligne de delete, vous pouvez avoir des fuites de mémoire.
- Pas de partage d'objet :
  Contrairement aux `std::shared_ptr` (voir plus loin) les pointeurs bruts ne gèrent pas automatiquement le partage de la propriété d'un objet. Vous devez vous assurer qu'il n'y a pas de tentatives de libérer le même pointeur plusieurs fois, ce qui provoquerait une erreur d'exécution.

Conseils et pièges à éviter

Voici les principaux pièges à éviter et des conseils pour chacun des aspects aborder précédemment.

Design Pattern Decorator

Le Decorator est un patron de conception qui permet d'ajouter dynamiquement des comportements à un objet, tout en respectant le principe de composition plutôt que de recourir à l'héritage. Il repose fortement sur le polymorphisme et l'héritage.

Écueils à éviter :

S'assurer que les décorateurs respectent :

- l'interface du composant : Le but du pattern Decorator est d'envelopper un objet et d'ajouter un comportement supplémentaire. Tous les décorateurs doivent donc implémenter la même interface que le composant de base. Sinon, il sera difficile de les utiliser de manière interchangeable.
  - Exemple :

```
class Component {
public:
    virtual void operation() = 0;
    virtual ~Component() = default;
```

```
};

class ConcreteComponent : public Component {
public:
    void operation() override {
        std::cout << "Base operation" << std::endl;
    }
};

class Decorator : public Component {  // Doit dériver de Component
protected:
    Component* component;
public:
    Decorator(Component* c) : component(c) {}
};
```

- Problèmes de destruction des objets :
  Si vous utilisez des pointeurs pour stocker des composants, assurez-vous que la destruction des objets est bien gérée. Le patron de conception `Decorator` crée souvent une hiérarchie de classes contenant des objets imbriqués. Si les destructeurs ne sont pas correctement définis (via `virtual`), il pourrait y avoir des fuites de mémoire ou une destruction incomplète des objets.
  - Solution :
    Déclarez des destructeurs virtuels pour vous assurer que la destruction est propre à travers toute la hiérarchie des décorateurs.
  - Exemple :

```
class Decorator : public Component {
public:
    virtual ~Decorator() {
        delete component;
    }
};
```

- Empilement de décorateurs :
  Lorsque vous composez plusieurs décorateurs, faites attention à l'ordre dans lequel les décorateurs sont appliqués. Un mauvais ordre pourrait conduire à des comportements inattendus.
  - Astuce (pour plus tard voir fin du document):
    Utilisez des smart pointers (pointeurs intelligents) pour gérer les objets décorés et éviter les fuites de mémoire, en particulier lorsque les décorateurs sont créés et détruits dynamiquement :
  - Exemple

```
std::shared_ptr<Component> component = std::make_shared<ConcreteComponent>();
std::shared_ptr<Component> decoratedComponent = std::make_shared<ConcreteDecorator>(component);
```

Polymorphisme et Héritage

Le polymorphisme et l'héritage sont étroitement liés en C++. Cependant, il y a des pièges potentiels lorsqu'ils sont mal utilisés.

Écueils à éviter :

- Oublier de déclarer les destructeurs comme virtual dans les classes de base :
  Cela est crucial si vous utilisez l'héritage polymorphique. Si vous détruisez un objet d'une classe dérivée via un pointeur de classe de base sans destructeur virtuel, cela peut entraîner un comportement indéfini et des fuites de mémoire.
  - Solution :
    Toujours définir le destructeur de la classe de base comme virtual, même s'il est vide :
  - Exemple

```
class Base {
public:
  virtual ~Base() {}  // Destructeur virtuel
```

```
};
```

- Problèmes de slicing :
  Le slicing se produit lorsque vous affectez un objet dérivé à une variable de type classe de base par valeur, coupant ainsi les membres spécifiques à la classe dérivée. Cela est généralement indésirable en C++.
  - Solution :
    Manipulez les objets via des références ou des pointeurs à la classe de base pour éviter ce problème.
  - Exemple

```
Base* obj = new Derived();  // Polymorphisme dynamique
```

- Utilisation excessive de l'héritage :
  Un mauvais usage de l'héritage peut rendre le code rigide et difficile à maintenir. Par exemple, étendre une hiérarchie de classes trop profondément peut devenir complexe à gérer.
  - Solution : Préférez parfois la composition à l'héritage lorsque cela a du sens. Le pattern Decorator est un bon exemple où la composition est préférable à l'héritage.
  - Astuce :
    Utilisez le mot-clé `override` pour clarifier et sécuriser la redéfinition des méthodes dans les classes dérivées. Cela permet d'éviter les erreurs de signature non détectées.
  - Exemple

```
class Derived : public Base {
public:
    void function() override {  // Vérification par le compilateur
        // Redéfinition de la fonction
    }
};
```

Constructeurs et Destructeurs

La gestion correcte des constructeurs et destructeurs est cruciale pour éviter les fuites de mémoire et les comportements imprévisibles dans les hiérarchies de classes.

Écueils à éviter :

- Problème de copie profonde :
  Si vos classes manipulent des ressources dynamiques (comme des pointeurs), un constructeur par défaut ou un opérateur d'affectation généré automatiquement par le compilateur pourrait entraîner une copie superficielle, ce qui pourrait poser problème avec la gestion de la mémoire.
  - Solution : Implémentez un constructeur de copie et un opérateur d'affectation personnalisés pour gérer correctement la copie de ressources dynamiques (méthode de la copie profonde).
  - Exemple

```
class MyClass {
    int* data;
public:
    MyClass(const MyClass& other) {  // Constructeur de copie
        data = new int(*other.data);
    }

    MyClass& operator=(const MyClass& other) {  // Opérateur d'affectation
        if (this != &other) {
            delete data;
            data = new int(*other.data);
        }
        return *this;
    }

    ~MyClass() {  // Destructeur
        delete data;
    }
};
```

- Double libération de mémoire (double free) :

Si vous utilisez des pointeurs bruts et gérez manuellement la mémoire, une mauvaise gestion peut entraîner une double libération de mémoire (libérer deux fois la même mémoire). Cela conduit à un comportement indéfini.
  - Solution : Utilisez des smart pointers (`std::unique_ptr`, `std::shared_ptr`) pour la gestion automatique de la mémoire.
- Problèmes liés aux constructeurs par défaut et surchargés :
  Lors de la création de classes avec plusieurs constructeurs surchargés, assurez-vous que chaque constructeur initialise correctement tous les membres de la classe, ou déléguez l'initialisation à un constructeur principal.
  - Astuce : Constructeurs de délégation : Si vous avez plusieurs constructeurs surchargés, utilisez la délégation pour centraliser la logique d'initialisation, réduisant ainsi le risque d'incohérence.
  - Exemple

```
class MyClass {
    int x, y;
public:
    MyClass(int a) : MyClass(a, 0) {}  // Délégation
    MyClass(int a, int b) : x(a), y(b) {}
};
```

Opérateurs d'affectation

En plus de gérer les constructeurs et destructeurs, vous devez prendre soin des opérateurs d'affectation pour assurer une copie et une gestion correcte des ressources dynamiques.

Écueils à éviter :

- Non-respect de la règle des trois ou cinq :
  Si vous devez gérer des ressources dans une classe (pointeurs, fichiers, etc.), vous devez généralement implémenter :
  1- Un destructeur.
  2- Un constructeur de copie.
  3- Un opérateur d'affectation.

  Et, en C++11 et plus, si vous travaillez avec des ressources dynamiques, vous devez également ajouter : 4- Un constructeur de déplacement (move constructor) 5- Un opérateur d'affectation par déplacement (move assignment operator)

  Si vous omettez l'un de ces éléments, cela peut entraîner des fuites de mémoire, des doublons de libération de mémoire ou des comportements imprévisibles.

  - Exemple de la règle des trois :

```
class MyClass {
    int* data;
public:
    MyClass(int val) : data(new int(val)) {}  // Constructeur
    ~MyClass() { delete data; }  // Destructeur

    MyClass(const MyClass& other) {  // Constructeur de copie
        data = new int(*other.data);
    }

    MyClass& operator=(const MyClass& other) {  // Opérateur d'affectation
        if (this != &other) {
            delete data;
            data = new int(*other.data);
        }
        return *this;
    }
```

Liaisons statique & dynamique, `vtable`

En C++, les concepts de liaison statique et dynamique sont cruciaux pour comprendre comment les appels de fonctions sont résolus. Voici une explication détaillée de ces concepts et de la table de méthodes virtuelles (vtable).

Liaison Statique

La liaison statique, également appelée *liaison à la compilation* se produit lorsque le compilateur peut déterminer exactement quelle fonction appeler à la compilation. Cela signifie que le compilateur sait quel code de fonction insérer à l'endroit de l'appel.

Exemple de liaison statique

```
class Circle {
public:
    void show() const {
        std::cout << "Circle show" << std::endl;
    }
};

int main() {
    Circle c;
    c.show(); // Liaison statique
    return 0;
}
```

Dans cet exemple, le compilateur sait que `c` est de type `Circle` et peut donc appeler directement `Circle::show()`. Cela permet des optimisations comme l'inlining, où le code de la fonction `show()` est inséré directement à l'endroit de l'appel.

Liaison Dynamique

La liaison dynamique, également appelée *liaison retardée ou liaison à l'exécution*, se produit lorsque le compilateur ne peut pas déterminer exactement quelle fonction appeler à la compilation. Cela se produit généralement avec les fonctions virtuelles.

Exemple de liaison dynamique

```
class Shape {
public:
    virtual void show() const {
        std::cout << "Shape show" << std::endl;
    }
};

class Circle : public Shape {
public:
    void show() const override {
        std::cout << "Circle show" << std::endl;
    }
};

int main() {
    Shape * pshape = new Circle();
    pshape->show(); // Liaison dynamique
    delete pshape;
    return 0;
}
```

Dans cet exemple, `pshape` est un pointeur vers une instance de `Shape` ou d'une classe dérivée de `Shape`. Le compilateur ne sait pas à la compilation quel type d'objet `pshape` pointe réellement. Par conséquent, il utilise une table de méthodes virtuelles (`vtable`) pour déterminer à l'exécution quel code de fonction exécuter.

Table de Méthodes Virtuelles (`vtable`)

La table de méthodes virtuelles (`vtable`) est une structure utilisée par le compilateur pour implémenter la liaison dynamique. Chaque classe avec des fonctions virtuelles a une vtable associée. Cette table contient des pointeurs vers les fonctions virtuelles de la classe.

Exemple de `vtable`

```cpp
class Shape {
public:
    virtual void show() const {
        std::cout << "Shape show" << std::endl;
    }
};

class Circle : public Shape {
public:
    void show() const override {
        std::cout << "Circle show" << std::endl;
    }
};
```

Shape a une vtable avec un pointeur vers `Shape::show()`. Circle a une vtable avec un pointeur vers `Circle::show()`.

Lorsque vous appelez `pshape->show()`, le compilateur utilise la `vtable` de l'objet pointé par `pshape` pour déterminer quelle fonction `show()` appeler. Si `pshape` pointe vers un objet de type `Circle`, la `vtable` de Circle sera utilisée, et `Circle::show()` sera appelé.

Conclusion

- Liaison statique : Le compilateur sait exactement quelle fonction appeler à la compilation.
- Liaison dynamique : Le compilateur utilise une vtable pour déterminer quelle fonction appeler à l'exécution.

La liaison dynamique est essentielle pour le polymorphisme en C++, permettant à un objet de se comporter différemment en fonction de son type réel à l'exécution. Cependant, elle introduit un coût supplémentaire par rapport à la liaison statique en raison de l'utilisation de la `vtable`.

Pour aller plus loin: Pointeurs intelligent

A ne pas utiliser dans ce TP!

Les pointeurs intelligents en C++ sont des classes qui facilitent la gestion de la mémoire dynamique en remplaçant les pointeurs bruts.

Voici un aperçu des trois types les plus courants : `std::shared_ptr`, `std::make_shared`, et `std::unique_ptr`.

`std::shared_ptr`

- Description :
  `std::shared_ptr` est un pointeur intelligent qui permet de partager la propriété d'un objet entre plusieurs pointeurs. Il utilise un compteur de références pour suivre le nombre de `shared_ptr` qui pointent vers le même objet. Lorsque le dernier `shared_ptr` qui pointe vers l'objet est détruit, l'objet est automatiquement supprimé.
- Usage :
  Utile lorsque plusieurs parties d'un programme doivent partager l'accès à un même objet.
  Exemples typiques : les objets utilisés dans des structures de données comme les graphes ou les arbres.
- Avantages :
  Gestion automatique de la mémoire : l'objet est libéré automatiquement lorsque le dernier pointeur est détruit.
  Simple à utiliser, avec un comportement similaire aux pointeurs bruts.
- Inconvénients : Surcharge de performance dû à la gestion du compteur de références. Risque de cycles de références, où deux ou plusieurs `shared_ptr` se référencent mutuellement, empêchant la libération de la mémoire.

`std::make_shared`

- Description :
  `std::make_shared` est une fonction utilitaire qui crée un `std::shared_ptr` à partir d'un objet, allouant la mémoire pour l'objet et le pointeur en une seule opération. Cela optimise la mémoire et améliore les performances par rapport à l'utilisation de new pour créer l'objet.

- Usage :
  Recommandé lors de la création d'un shared_ptr pour améliorer l'efficacité mémoire et performance. Il crée et renvoie un `std::shared_ptr` à un nouvel objet en mémoire.
- Avantages :
  Réduction des frais généraux d'allocation mémoire (l'objet et le compteur de références sont alloués en une seule opération). Simplification de la syntaxe, car vous n'avez pas besoin de gérer l'allocation de mémoire manuellement.
- Inconvénients :
  Moins flexible que new, car vous ne pouvez pas contrôler la manière dont l'objet est construit ou fournir des arguments à son constructeur (à moins d'utiliser des overloads).

`std::unique_ptr`

- Description :
  `std::unique_ptr` est un pointeur intelligent qui représente la possession exclusive d'un objet. Il ne peut pas être copié, mais peut être transféré d'un `unique_ptr` à un autre à l'aide de `std::move()`. Cela garantit qu'il n'y a qu'un seul propriétaire d'un objet à un moment donné.
- Usage :
  Idéal pour la gestion de ressources qui doivent être possédées par un seul pointeur.
  Utilisé lorsque la durée de vie d'un objet doit être clairement définie et contrôlée.
- Avantages :
  Aucune surcharge de compteur de références, ce qui le rend plus performant que `shared_ptr`. Prévention des fuites de mémoire, car l'objet est libéré lorsque le `unique_ptr` est détruit.
- Inconvénients :
  Ne peut pas être copié (uniquement déplacé), ce qui limite sa flexibilité par rapport à `shared_ptr`. Pas adapté pour les situations où vous avez besoin de partager la propriété de l'objet.

Comparaison

Caractéristique	`std::shared_ptr`	`std::make_shared`	`std::unique_ptr`
Propriété	Partagée	Partagée	Exclusive
Compteur de références	Oui	Oui	Non
Copie	Oui	Non	Non
Déplacement	Non (via `std::move()`)	Non (via `std::move()`)	Oui
Allocation mémoire	Deux allocations (pointeur et objet)	Une seule allocation (optimisée)	Une seule allocation
Performances	Légèrement plus lent	Optimisé (plus rapide)	Plus rapide

Conclusion

- `std::shared_ptr` est utilisé pour le partage d'objets entre plusieurs parties d'un programme, avec une gestion automatique de la mémoire.
- `std::make_shared` est recommandé pour la création de shared_ptr pour améliorer l'efficacité.
- `std::unique_ptr` est utilisé pour la gestion d'objets avec une propriété exclusive, garantissant qu'il n'y a qu'un seul propriétaire à la fois.

Chaque type de pointeur intelligent a ses propres avantages et inconvénients, et le choix dépend du contexte et des besoins spécifiques de votre programme.