Templates (Șabloane) în C++

Anul Universitar 2023 – 2024, Semestrul II

Programare Orientată pe Obiecte

LABORATOR 9, GRUPA 133

Cuprins

Autor: Wagner Ștefan Daniel

I. Introducere în Design Patterns (Sabloane de Proiectare)

1. Definiție și importanță

Design Patterns (sabloane de proiectare) sunt soluții standardizate pentru probleme comune întâlnite în designul software. Ele reprezintă "șabloane" pentru scrierea codului eficient și reutilizabil, făcând astfel software-ul mai ușor de înțeles și de întreținut. Folosirea acestor pattern-uri ajută dezvoltatorii să evite capcanele comune prin oferirea unor practici bine stabilite.

2. Necesitatea Design Patterns

În dezvoltarea software-ului, inginerii se confruntă adesea cu probleme similare. Reutilizarea soluțiilor care au fost deja bine gândite și testate poate economisi timp și poate reduce numărul de bug-uri. Design patterns oferă o limbă comună între dezvoltatori, facilitând discuția despre soluții de design într-un mod abstract, fără a intra în detalii de implementare specifice.

3. Categorii de Design Patterns

Design patterns sunt de obicei împărțite în trei categorii principale:

- **Creational Patterns (Sabloane Creationale)**: Se concentrează pe mecanismele de creare a obiectelor într-un mod care să sporească flexibilitatea și reutilizarea codului existent. Exemple includ Singleton, Factory Method, Builder, și Prototype.
- **Structural Patterns (Sabloane Structurale)**: Se ocupă de compoziția obiectelor și claselor. Aceste pattern-uri ajută la formarea de structuri mari prin asamblarea obiectelor și claselor în structuri mai mari, simplificând în același timp complexitatea. Exemplele includ Adapter, Decorator, Facade, și Composite.
- **Behavioral Patterns (Sabloane Comportamentale)**: Se concentrează pe comunicația eficientă și distribuția responsabilității între obiecte. Acestea descriu modele de comunicare complexe, facilitând astfel fluxurile de lucru între obiecte. Exemple includ Observer, Strategy, Command, și Iterator.

II. Creational Patterns

1. Descriere generală

Pattern-urile creaționale sunt folosite pentru a controla procesul de creare a obiectelor în software, facilitând flexibilitatea și independența în instanțierea obiectelor. Acestea permit sistemului să fie independent de modul specific în care obiectele sunt create, compuse și reprezentate.

2. Singleton

Singleton este un design pattern creational care asigură că o clasă are o singură instanță, și oferă un punct de acces global la aceasta. Acesta este folosit frecvent pentru gestionarea resurselor în aplicații, cum ar fi conexiuni la baze de date sau fișiere de configurare.

Exemplu ilustrativ: Singleton Pattern

```cpp
#include <iostream>

class Singleton {
private:
    static Singleton* instance;
    Singleton() {} // Constructor privat pentru a preveni instantierea.

public:
    Singleton(const Singleton&) = delete; // Prevenim copierea.
    Singleton& operator=(const Singleton&) = delete; // Prevenim atribuirea.

    static Singleton* GetInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }

    void DoSomething() {
        std::cout << "Handling some singleton logic" << std::endl;
    }
};

Singleton* Singleton::instance = nullptr;

int main()
{
```

```
    Singleton* singletonInstance = Singleton::GetInstance();
    singletonInstance->DoSomething();
    return 0;
}
```

3. Factory Method

Factory Method definește o interfață pentru crearea unui obiect, dar lasă clasele care implementează interfața să decidă clasa către care să facă instanțiere. Aceasta permite unui sistem să fie mai flexibil în ceea ce privește tipurile de obiecte create.

Exemplu de Factory Method:

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

class Product {
public:
    virtual void Operate() = 0;
    virtual ~Product() {}
};

class ConcreteProductA : public Product {
public:
    void Operate() override {
        std::cout << "Operating ConcreteProductA" << std::endl;
    }
};

class ConcreteProductB : public Product {
public:
    void Operate() override {
        std::cout << "Operating ConcreteProductB" << std::endl;
    }
};

class Creator {
public:
    virtual std::unique_ptr<Product> FactoryMethod() = 0;
    virtual ~Creator() {}
};

class ConcreteCreatorA : public Creator {
public:
    std::unique_ptr<Product> FactoryMethod() override {
```

```
        return std::make_unique<ConcreteProductA>();
    }
};

class ConcreteCreatorB : public Creator {
public:
    std::unique_ptr<Product> FactoryMethod() override {
        return std::make_unique<ConcreteProductB>();
    }
};

int main()
{
    std::unique_ptr<Creator> creatorA = std::make_unique<ConcreteCreatorA>();
    std::unique_ptr<Product> productA = creatorA->FactoryMethod();
    productA->Operate();

    std::unique_ptr<Creator> creatorB = std::make_unique<ConcreteCreatorB>();
    std::unique_ptr<Product> productB = creatorB->FactoryMethod();
    productB->Operate();
    return 0;
}
```

4. Builder (Constructor)

4.1. Descriere Generală

Builder este un pattern creational care permite construirea pas cu pas a unui obiect complex, separând construcția unui obiect de reprezentarea sa. Acest lucru permite același proces de construcție să creeze diferite reprezentări.

4.2. Scopul Pattern-ului Builder

Scopul principal al pattern-ului Builder este de a izola complexitatea construcției unui obiect complex din clasa sa principală. Aceasta se face prin introducerea unui builder care încapsulează modul în care un obiect complex este construit și asamblat, permitând controlul fin al procesului de construcție.

4.3. Exemplu de Implementare a Builder

Să luăm exemplul unei mașini, unde mașina poate avea diverse configurații pentru motor, interior și opțiuni exterioare. Un builder pentru acest tip de obiect permite specificarea detaliilor pas cu pas, fără a complica constructorul clasei **Car**.

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

// Produsul final
class Car {
    std::string make;
    std::string model;
    int horsepower;
    std::string interior;

public:
    void SetMake(const std::string& make) { this->make = make; }
    void SetModel(const std::string& model) { this->model = model; }
    void SetHorsepower(int horsepower) { this->horsepower = horsepower; }
    void SetInterior(const std::string& interior) { this->interior = interior;
}

    void Display() const {
        std::cout << "Car: " << make << " " << model
                  << " with " << horsepower << " horsepower and "
                  << interior << " interior." << std::endl;
    }
};

// Abstract Builder
class CarBuilder {
protected:
    Car car;

public:
    virtual ~CarBuilder() {}
    Car GetCar() { return car; }

    virtual void BuildMake() = 0;
    virtual void BuildModel() = 0;
    virtual void BuildHorsepower() = 0;
    virtual void BuildInterior() = 0;
};

// Concrete Builder
class SportsCarBuilder : public CarBuilder {
public:
    void BuildMake() override { car.SetMake("Ferrari"); }
    void BuildModel() override { car.SetModel("488 Spider"); }
    void BuildHorsepower() override { car.SetHorsepower(670); }
```

```
    void BuildInterior() override { car.SetInterior("Leather"); }
};

// Director
class Director {
    CarBuilder* builder;

public:
    void SetBuilder(CarBuilder* b) { builder = b; }

    Car Construct() {
        builder->BuildMake();
        builder->BuildModel();
        builder->BuildHorsepower();
        builder->BuildInterior();
        return builder->GetCar();
    }
};

int main() {
    Director director;
    SportsCarBuilder builder;
    director.SetBuilder(&builder);

    Car myDreamCar = director.Construct();
    myDreamCar.Display();
}
```

4.4. Avantajele Utilizării Builder

- **Flexibilitate**: Posibilitatea de a schimba implementarea și detaliile produsului final fără a modifica codul client.
- **Control Mai Bun asupra Procesului de Construcție**: Permite construirea obiectelor pas cu pas și închiderea finală a obiectului pentru a preveni starea incompletă.
- **Separarea Codului pentru Obiect și Crearea**: Separă reprezentarea complexă a obiectelor de logica de construcție care este încapsulată în builder.

Prin urmare, Builder este un pattern valoros pentru scenariile unde avem de-a face cu obiecte complexe, al căror proces de construcție trebuie să fie mai flexibil sau configurabil decât ceea ce permite un simplu constructor.

Aceste exemple ilustrează cum pattern-urile creationale pot fi utilizate pentru a abstractiza detalii ale construcției instanțelor, permițând codului să fie mai modular și mai ușor de extins. Următoarele secțiuni vor continua cu pattern-uri structurale și comportamentale, oferind o viziune cuprinzătoare asupra aplicării pattern-urilor de design în proiectele software

III. Structural Patterns (Sabloane Structurale)

1. Descriere Generală

Pattern-urile structurale se concentrează pe modul de organizare a obiectelor și claselor în structuri mai mari, în timp ce mențin flexibilitatea și eficiența interfețelor. Aceste sabloane ajută la proiectarea arhitecturilor care facilitează comunicația și colaborarea între diferite componente ale sistemului.

2. Adapter

Adapter-ul permite obiectelor cu interfețe incompatibile să colaboreze. Acesta "adaptează" interfața unei clase existente într-o altă interfață, așteptată de client, fără a modifica codul sursă original al clasei.

Exemplu de Adapter:

```
##include <iostream>

// Interfata tinta
class Target {
public:
    virtual void Request() const = 0;
};

// Clasa existenta cu o interfata diferita
class Adaptee {
public:
    void SpecificRequest() const {
        std::cout << "Specific request." << std::endl;
    }
};

// Adapterul
class Adapter : public Target {
private:
    Adaptee* adaptee;
```

```
public:
    Adapter(Adaptee* a) : adaptee(a) {}
    void Request() const override {
        adaptee->SpecificRequest();
    }
};

int main() {
    Adaptee* adaptee = new Adaptee();
    Target* target = new Adapter(adaptee);
    target->Request();
    delete adaptee;
    delete target;
    return 0;
}
```

3. Decorator

Decoratorul adaugă responsabilități suplimentare obiectelor dinamic, oferind o alternativă flexibilă la subclasare pentru extinderea funcționalităților.

Exemplu de Decorator:

```
#include <iostream>

// Componenta abstracta
class Component {
public:
    virtual void Operation() = 0;
    virtual ~Component() {}
};

// Componenta concreta
class ConcreteComponent : public Component {
public:
    void Operation() override {
        std::cout << "ConcreteComponent operation." << std::endl;
    }
};

// Decoratorul abstract
class Decorator : public Component {
protected:
    Component* component;
```

```
public:
    Decorator(Component* c) : component(c) {}
    void Operation() override {
        component->Operation();
    }
};

// Decorator concret
class ConcreteDecorator : public Decorator {
public:
    ConcreteDecorator(Component* c) : Decorator(c) {}
    void Operation() override {
        Decorator::Operation();
        AddedBehavior();
    }
    void AddedBehavior() {
        std::cout << "ConcreteDecorator added behavior." << std::endl;
    }
};

int main() {
    Component* simple = new ConcreteComponent();
    Component* decorated = new ConcreteDecorator(simple);
    decorated->Operation();
    delete simple;
    delete decorated;
    return 0;
}
```

4. Facade

Design pattern-ul ”Façade” oferă o interfață simplificată la un sistem complex, reducând complexitatea generală a aplicației și ascunzând interacțiunile între sub-sisteme.

Exemplu de Facade:

```
#include <iostream>

class SubsystemA {
public:
    void OperationA() {
        std::cout << "Subsystem A operation." << std::endl;
    }
};

class SubsystemB {
```

```
public:
    void OperationB() {
        std::cout << "Subsystem B operation." << std::endl;
    }
};

class Facade {
private:
    SubsystemA* subsystemA;
    SubsystemB* subsystemB;

public:
    Facade() : subsystemA(new SubsystemA()), subsystemB(new SubsystemB()) {}
    ~Facade() {
        delete subsystemA;
        delete subsystemB;
    }
    void Operation() {
        subsystemA->OperationA();
        subsystemB->OperationB();
    }
};

int main() {
    Facade* facade = new Facade();
    facade->Operation();
    delete facade;
    return 0;
}
```

Aceste pattern-uri structurale ajută la proiectarea software-ului într-un mod care promovează reutilizarea și modularitatea, facilitând gestionarea complexității în sisteme mari.

Următorul capitol va explora pattern-uri comportamentale, care se concentrează pe gestionarea comunicării și interacțiunii între obiecte.

IV. Behavioral Patterns (Sabloane Comportamentale)

1. Descriere Generală

Pattern-urile comportamentale se concentrează pe eficientizarea comunicației între obiecte, gestionând relațiile complexe între acestea și distribuind responsabilitățile într-un mod care sporește flexibilitatea interacțiunilor. Aceste sabloane permit obiectelor să colaboreze în moduri bine definit și scalabil.

2. Observer

Observer permite unui număr de obiecte să observe și să răspundă la evenimentele emise de un alt obiect. Este util pentru crearea de sisteme în care starea unui obiect trebuie să fie monitorizată de unul sau mai multe obiecte.

Exemplu de Observer:

```
#include <iostream>
#include <vector>
#include <algorithm>

class Observer {
public:
    virtual void Update(const std::string& messageFromSubject) = 0;
};

class Subject {
    std::vector<Observer*> observerList;
public:
    void Attach(Observer* observer) {
        observerList.push_back(observer);
    }
    void Detach(Observer* observer) {
        observerList.erase(std::remove(observerList.begin(),
observerList.end(), observer), observerList.end());
    }
    void Notify(const std::string& message) {
        for (Observer* observer : observerList) {
            observer->Update(message);
        }
    }
};

class ConcreteObserver : public Observer {
public:
    void Update(const std::string& messageFromSubject) override {
        std::cout << "Observer received: " << messageFromSubject << std::endl;
    }
};

int main() {
    Subject subject;
    ConcreteObserver observer;
    subject.Attach(&observer);
```

```
    subject.Notify("Hello Observers!");
    subject.Detach(&observer);
    return 0;
}
```

3. Strategy

Strategy definește o familie de algoritmi, încapsulează fiecare dintre ele și le face interschimbabile. Strategy permite schimbarea algoritmului de execuție la runtime în funcție de context.

Exemplu de Strategy:

```
#include <iostream>
class Strategy {
public:
    virtual void AlgorithmInterface() = 0;
};
class ConcreteStrategyA : public Strategy {
public:
    void AlgorithmInterface() override {
        std::cout << "Implemented Algorithm A" << std::endl;
    }
};
class ConcreteStrategyB : public Strategy {
public:
    void AlgorithmInterface() override {
        std::cout << "Implemented Algorithm B" << std::endl;
    }
};
class Context {
    Strategy* strategy;
public:
    Context(Strategy* strategy) : strategy(strategy) {}
    void SetStrategy(Strategy* strategy) {
        this->strategy = strategy;
    }
    void ExecuteStrategy() {
        strategy->AlgorithmInterface();
    }
};
int main() {
    ConcreteStrategyA strategyA;
    ConcreteStrategyB strategyB;
    Context context(&strategyA);
    context.ExecuteStrategy();
```

```
    context.SetStrategy(&strategyB);
    context.ExecuteStrategy();
    return 0;
}
```

4. Command

Command încapsulează o comandă ca un obiect, permițând parametrizarea obiectelor cu diferite solicitări, coada sau logarea operațiilor și suportul pentru operațiuni reversibile.

Exemplu de Command:

```
#include <iostream>
#include <vector>

class Command {
public:
    virtual void Execute() = 0;
    virtual ~Command() {}
};

class Light {
public:
    void TurnOn() {
        std::cout << "Light turned on." << std::endl;
    }
    void TurnOff() {
        std::cout << "Light turned off." << std::endl;
    }
};

class TurnOnCommand : public Command {
    Light& light;
public:
    TurnOnCommand(Light& light) : light(light) {}
    void Execute() override {
        light.TurnOn();
    }
};

class TurnOffCommand : public Command {
    Light& light;
public:
    TurnOffCommand(Light& light) : light(light) {}
    void Execute() override {
        light.TurnOff();
```

```
    }
};

int main() {
    Light light;
    TurnOnCommand turnOn(light);
    TurnOffCommand turnOff(light);
    turnOn.Execute();
    turnOff.Execute();
    return 0;
}
```

Aceste exemple ilustrează cum sabloanele comportamentale pot facilita managementul complexităților în interacțiunile dintre obiecte, permițând un design modular și extensibil al aplicațiilor.

5. Notă Suplimentară: Diferența între Strategy și Factory Method și Similitudinile

Diferențe între Strategy și Factory Method

1. Scopul:

- **Strategy** este un pattern comportamental care se concentrează pe schimbarea algoritmului de comportament al unui obiect la runtime. Acesta permite definirea unei familii de algoritmi, încapsulează fiecare dintre ele, și le face interschimbabile fără a schimba clienții care le utilizează.
- **Factory Method** este un pattern creational care se concentrează pe crearea de obiecte fără a specifica clasa exactă a obiectului care va fi creat. Acesta permite o clasă să deleagă instanțierea obiectelor către subclase.

2. Implementare:

- **Strategy** implică definirea unui set de interfețe pentru algoritmii interschimbabili, unde fiecare strategie concretă implementează această interfață. Contextul va folosi o referință la interfața Strategy pentru a executa comportamentul dorit.
- **Factory Method** implică o interfață care definește metoda factory, iar subclasele implementează această metodă pentru a crea instanțe ale clasei corespunzătoare. Metoda

factory este folosită pentru a crea obiecte, ascunzând detalii despre clasele concrete utilizate de la utilizatori.

Similitudini între Strategy și Factory Method

1. Încapsulare:

- Ambele pattern-uri încapsulează comportamentul — Strategy încapsulează comportamentul algoritmic, în timp ce Factory Method încapsulează crearea de obiecte. Această încapsulare ajută la separarea responsabilităților în aplicație, ceea ce face codul mai ușor de gestionat și de extins.

2. Delegare:

- În ambele cazuri, responsabilitatea principală este delegată altor clase: Strategy deleghează modul în care se execută o anumită acțiune, în timp ce Factory Method deleghează crearea unei instanțe a unui obiect. Aceasta delegare permite flexibilitatea și extensibilitatea în gestionarea logicii de aplicație.

3. Principii SOLID:

- Ambele pattern-uri urmează principiul Open/Closed Principle din SOLID (despre care vom discuta în detaliu saptamana viitoare), care spune că software-ul ar trebui să fie deschis pentru extensie, dar închis pentru modificare. Strategy permite adăugarea de noi comportamente fără a modifica contextul, iar Factory Method permite adăugarea de noi clase fără a modifica codul care instantiază obiectele.

Prin urmare, deși Strategy și Factory Method sunt utilizate în scopuri diferite și manipulează aspecte diferite ale design-ului software (comportament vs. creare), ele împărtășesc similitudini în ceea ce privește încapsularea, delegarea și aderența la principiile de design solid. Această înțelegere a diferențelor și similitudinilor poate ajuta dezvoltatorii să aleagă pattern-ul potrivit în funcție de problema specifică cu care se confruntă.

6. Concluzii Finale și Bibliografie Suplimentară

Utilizarea corectă a Design Patterns necesită o viață de experiență practică în programare, și în mod cert timpul alocat laboratorului nu este destul decât pentru o introducere în cateva Design Patterns comune, categoriile din care fac parte prezentate, și motivația pentru cât de importante sunt în producție, indiferent de limbajul de programare în care veți lucra.

Pentru cei interesați să învețe mai mult, cea mai "clasică" resursă recomndata este cartea "Gang of Four" - **Design Patterns: Elements of Reusable Object-Oriented Software (1994).**

Deși este o carte relativ veche față de cum evoluează tehnologia, tot ce este scris acolo este înca de actualitate ca principiu, și toate resursele scrise despre Design Patterns au ca sursă de inspirație inițială cartea indicată.

Cartea este de încă actualitate, deoarece, evident, un lucru odată făcut bine, nu mai trebuie reinventat.

O altă resursa ar fi cartea **Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software**, care, deși explică conceptele în Java (nu contează limbajul de programare), este mult mai "beginner-friendly", și ofera multe explicații suplimentare.

O resursă TLDR ar fi site-ul **https://refactoring.guru/design-patterns**, unde găsiți ușor explicațiile alături de exemple de cod, pentru fiecare categorie de Design Pattern, pentru fiecare Design Pattern din acea categorie.

Evident, acestea nu sunt toate Design Pattern-urile (nici cele din laborator, nici cele din cărțile indicate), însa sunt cele fundamentale, și cele mai des utilizate, care nu vor disparea niciodată (afirmație puternica, dar mi-o asum) din producție, deoarece sunt deja soluții universal adaptate pentru a scrie cod cât mai curat, modular, corect, și într-un limbaj internațional cunoscut, cel al Design Patterns.