Design Patterns

Vîlculescu Mihai-Bogdan

10 mai 2020

Contents

0 Examen

Pentru examen e necesar să cunoașteți doar câteva design patterns: **Singleton** (și particularizări ale lui). Restul sunt în plus pentru cei care vor să aprofundeze subiectul care e important pentru interviuri și, mai ales, pentru voi ca software developeri.

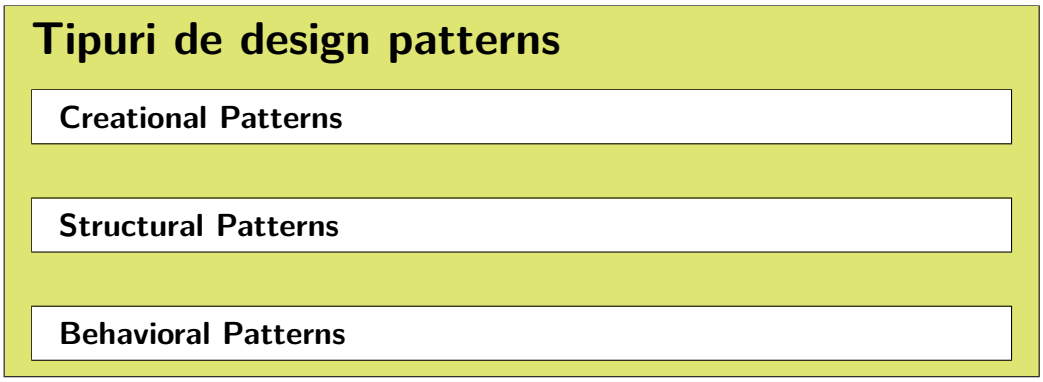
1 Introducere

1.1 Definiție

Sunt **soluții tipice pentru probleme comune** din dezvoltarea de software. Acestea sunt independente de limbaj, pentru că se referă mai mult la **proiectarea unor clase** decât la modul în care acestea sunt implementate.

Adică, un design pattern din C++ este același și în Java, cu diferențe minore de implementare.

1.2 Clasificare



2 Creational Patterns

2.1 Introducere

După cum spune și numele, pattern-urile creaționale oferă diverse **mecanisme de creare a obiectelor**. Astfel, este mai ușoară întreținerea codului care **poate fi refolosit și e mai modularizat**.

2.2 Singleton

2.2.1 Definiție

Un **design pattern creațional** care, prin diverse mecanisme, permite crearea unei clase care are în permanență **o singură instanță**. De asemenea, o clasă de tip Singleton trebuie să permită accesul global la acea instanță.

2.2.2 De ce?

Înainte să continuăm trebuie să răspundem la întrebarea pe care cei mai mulți o au în momentul studierii acestor design patterns: **de ce avem nevoie de ele?**

Probleme rezolvate de Singleton

În general, Singleton se ocupă de procese a căror creare este costisitoare din punct de vedere al timpului necesar.

Menține o singură conexiune activă la o bază de date.

Menține un singur flux deschis la un fișier comun mai multor procese.

Menținerea unei singure instanțe pentru meniul unui joc.

2.2.3 Exemplu din viața reală

Exemplul acesta este preluat de pe un site genial pentru a învăța despre design patterns, pe care o să îl menționez oricum și mai jos în resurse: Refactoring Guru - Singleton.

Exemplu pentru Singleton

O țară poate avea un singur guvern oficial indiferent de identitățile celor care formează acel guvern.

2.2.4 Metodă de rezolvare

Există mai multe implementări posibile pentru acest design pattern, fiecare cu propriile sale probleme.

Totuși, există 2 puncte comune în toate aceste implementări:

Pași pentru a implementa Singleton

Se creează un **constructor privat** pentru a împiedica crearea de noi instanțe prin cuvântul cheie **new**

Se creează un **câmp de date static** care va fi un **pointer către un obiect de tipul clasei** și va reprezenta **instanța unică** a acelei clase

Se creează o **metodă statică** care apelează **constructorul privat** dacă nu a fost creată deja o **instanță a clasei**

2.2.5 Implementare

```
class Singleton {
    private:
        static Singleton* instanta;

        Singleton() {
            cout << "Constructor called";
        }

    public:
        static Singleton* getInstanta() {
            if (instanta == NULL) {
                instanta = new Singleton;
            }

            return instanta;
        }
};

int main () {
    Singleton* s1;
    s1 = Singleton::getInstanta(); // Constructor called

    Singleton* s2;
    s2 = Singleton::getInstanta(); // nu intra in constructor

    // Singleton s3; // constructorul e privat => eroare
    // Singleton* s3 = new Singleton;  // eroare
}
```

Exemplu de implementare singleton

2.2.6 Particularizări

Deși situațiile practice sunt rare, acest design pattern mai poate fi particularizat pentru alt **număr de instanțe**. Se poate da la **examen** o astfel de particularizare.

Spre exemplu, se poate cere crearea unei clase care să nu permită existența simultană a mai mult de 3 instanțe.

2.3 Aprofundare

Nu pot aborda în acest tutoriat toate design pattern-urile creaționale importante (care oricum nu o să intre la examen), dar vă recomand să citiți mai mult de pe link-ul următor: Creational Patterns.

Recomandări: **Builder**, **Abstract Factory**.

3 Structural Patterns

3.1 Introducere

Ajută la asamblarea obiectelor și claselor în **structuri mai mari**, menținându-le, în același timp, **flexibile** și **eficiente**.

3.2 Proxy

3.2.1 Definiție

Permite oferirea unui **înlocuitor** (placeholder) pentru un alt obiect. Un proxy controlează accesul la obiectul original, permițând efectuarea unor operații fie înainte fie după ce **request-ul** ajunge la obiectul original.

3.2.2 De ce?

Probleme rezolvate de Proxy

În general, Proxy este folosit pentru a controla accesul la un anumit obiect care consumă **multe resurse**, dar de care nu e nevoie întotdeauna.

3.2.3 Exemplu din viața reală

Exemplu pentru Proxy

Un **card de credit** este un **proxy** pentru **contul bancar** care, la rândul său, este un proxy pentru **banii cash**. Și banii cash și cardul permit efectuarea de **plăți**.

Consumatorul e mai fericit că nu trebuie să umble cu bani cash după el. **Vânzătorul** e fericit că venitul este procesat automat electronic, fără a mai exista riscul de a fi jefuit în drumul spre bancă.

3.2.4 Metodă de rezolvare

Pași pentru a implementa Proxy

Se creează o **clasă abstracă** care va fi moștenită atât de clasa Proxy cât și de clasa seriviciu.

Se creează **clasa Proxy** care moștenește clasa de la punctul precedent.

Se implementează metodele **clasei Proxy**. De obicei, acestea apelează metode din **clasa serviciu**.

3.2.5 Implementare

```
class AbstractService {
    public:
        virtual void request() = 0;
};

class Service: public AbstractService {
    public:
        void request () {
```

```
            cout << "Serviciul se ocupa de request\n";
        }
};

class Proxy: public AbstractService {
    private:
        Service* service;

        // Un exemplu de functionalitate adaugata de proxy
        bool accesPermis () const {
            cout << "Proxy verifica daca accesul este permis
             ↪  pentru utilizator\n";
            return true;
        }

        void inregistreazaAcces () const {
            cout << "Proxy inregistreaza accesul
             ↪  utilizatorului la ora ...\n";
        }

    public:
        Proxy (Service* _service) {
            service = new Service(*_service);
        }
        ~Proxy () {
            delete service;
        }

        void request () {
            if (this->accesPermis()) {
                // cererea e procesata de serviciu
                this->service->request();
                this->inregistreazaAcces();
            }
        }
};
```

```
class Client {
    public:
        void interactWithService (const AbstractService&
        ↪  service) {
            service.request();
        }
};

int main () {
    Client client;
    Service* service = new Service;

    client.interactWithService(*service);
    ↪  // Serviciul se ocupa de request

    Proxy* proxy = new Proxy(service);
    client.interactWithService(*proxy);
    // Proxy verifica daca accesul este permis pentru
    // utilizator
    // Serviciul se ocupa de request
    // Proxy inregistreaza accesul utilizatorului la ora ...
}
```

Exemplu de implementare proxy

3.3 Aprofundare

Structural patterns
Recomandări: **Adapter**, **Facade**.

4 Behavioral Patterns

4.1 Introducere

Acest tip de design pattern se ocupă de **algoritmi** și de atribuirea de **responsabilități** între obiecte.

4.2 Observer

4.2.1 Definiție

Permite definirea unui mecanism care **notifică** diverse obiecte despre orice **eveniment** care se întâmplă obiectelor pe care acestea le **observă**.

4.2.2 De ce?

Probleme rezolvate de Observer

În general, Observer este folosit pentru a putea observa orice modificare adusă asupra unui obiect observat.

Afișarea unui mesaj la click-ul unui buton.

Modificarea valorii unei variabile în momentul unei schimbări aduse bazei de date.

4.2.3 Exemplu din viața reală

Exemplu pentru Observer

Dinamica dintre un Client și un Vânzător. **Clientul** își dorește un obiect, pe care **vânzătorul** încă nu îl are pe stoc, dar urmează să îl primească.

Clientul ar putea să verifice în fiecare zi dacă obiectul dorit a fost primit, ceea ce e destul de anevoios. Altfel, un obiect de tipul **Observer** îl **notifică** doar în momentul în care vânzătorul a primit obiectul.

4.2.4 Implementare

```
class AbstractObserver {
    public:
        virtual ~AbstractObserver () {}
        virtual void update(string) = 0;
};

class AbstractSubject {
    public:
        virtual ~AbstractSubject () {}
        virtual void attach (AbstractObserver*) = 0;
        virtual void detach (AbstractObserver*) = 0;
        virtual void notify () = 0;
};

class Subject: public AbstractSubject {
    // subiectul care va fi observat
    private:
        list<AbstractObserver*> observers;
        string message;

    public:
        virtual ~Subject() {
            std::cout << "Destructor Subject\n";
        }

        void attach (AbstractObserver* observer) {
            observers.push_back(observer);
        }

        void detach (AbstractObserver* observer) {
            observers.remove(observer);
        }

        void notify () {
            cout << observers.size() << " observatori \n";
```

```
            // Parcurgem lista de observatori
            // Actualizam fiecare observator
            list<AbstractObserver*>::iterator it =
             ↪  observers.begin();
            while (it != observers.end()) {
                (*it)->update(message);
                ++it;
            }
        }

        void createMessage (string message) {
            this->message = message;
            notify();
        }
};

class Observer: public AbstractObserver {
    private:
        string messageFromSubject;
        Subject& subject;
        static int staticNumber;
        int number;

    public:
        Observer (Subject& _subject): subject(_subject) {
            this->subject.attach(this);

            cout << "Observatorul nr. " << staticNumber + 1 <<
             ↪  "\n";
            staticNumber++;
            number = staticNumber;
        }
        virtual ~Observer () {
            cout << "Destructor observator nr. " << number <<
             ↪  "\n";
        }
```

```
        void update (string message) {
            messageFromSubject = message;
            cout << "Observatorul nr. " << number << " - un
             ↪  nou mesaj e disponibil ---> ";
            cout << messageFromSubject << "\n";
        }

        void removeFromList () {
            subject.detach(this);
            cout << "Observatorul nr. " << number << " este
             ↪  eliminat din lista\n";
        }

};

int Observer::staticNumber = 0;

int main () {
    Subject* subject = new Subject;

    Observer* o1 = new Observer(*subject);
    Observer* o2 = new Observer(*subject);
    Observer* o3 = new Observer(*subject);
    Observer* o4;

    subject->createMessage("Hello world");
    o3->removeFromList();

    subject->createMessage("Hello again");
    o4 = new Observer(*subject);
    // ...
}
```

Exemplu de implementare observer

Output-ul pentru codul de mai sus arată astfel:

```
Observatorul nr. 1
Observatorul nr. 2

3 observatori
Observatorul nr. 1-un nou mesaj disponibil ---> Hello World
Observatorul nr. 2-un nou mesaj disponibil ---> Hello World
Observatorul nr. 3-un nou mesaj disponibil ---> Hello World
Observatorul nr. 3 este eliminat din lista

2 observatori
Observatorul nr. 1-un nou mesaj disponibil ---> Hello again
Observatorul nr. 2-un nou mesaj disponibil ---> Hello again

Observatorul nr. 4
...
```

4.3 Aprofundare

Behavioral patterns
Recomandări: **Visitor**.

5 Resurse

- Refactoring Guru
- Source Making
- TutorialsPoint
- GeeksForGeeks