Programmation Orientée Objets 2
Cours 8

Rémi Watrigant[1]

[1] basé sur le cours de Marianne Huchard

Introduction aux patrons de conception (design pattern)

Introduction

Définition

« Chaque patron décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution à ce problème, d'une façon telle que l'on puisse réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière » — Christopher Alexander, 1977.

$\Rightarrow$ solutions génériques à des problèmes de modélisation courants

Introduction

Pour présenter un patron de conception :

- présenter le problème
- présenter le schéma général du patron
- présenter un exemple

Introduction

Pour présenter un patron de conception :

- présenter le problème
- présenter le schéma général du patron
- présenter un exemple

Plusieurs catégories de patrons de conception :

- **pattern de comportement** :
  comment distribuer les reponsabilités/fonctions entre objets
  exemple : State, Observer, Visitor...
- **pattern de création** :
  pour créer, configurer les objets d'une application
  exemple : Factory, Singleton...
- **pattern de structure** définir l'organisation d'une partie de l'application au sein de l'application générale
  exemple : Composite, Adapter...

Introduction

Pour présenter un patron de conception :

- présenter le problème
- présenter le schéma général du patron
- présenter un exemple

Plusieurs catégories de patrons de conception :

- **pattern de comportement** :
  comment distribuer les reponsabilités/fonctions entre objets
  exemple : State, Observer, Visitor...
- **pattern de création** :
  pour créer, configurer les objets d'une application
  exemple : Factory, Singleton...
- **pattern de structure** définir l'organisation d'une partie de l'application au sein de l'application générale
  exemple : Composite, Adapter...

patrons de conception les plus connus : une vingtaine

Introduction

Pour présenter un patron de conception :

- présenter le problème
- présenter le schéma général du patron
- présenter un exemple

Plusieurs catégories de patrons de conception :

- **pattern de comportement** :
  comment distribuer les reponsabilités/fonctions entre objets
  exemple : State, Observer, Visitor...
- **pattern de création** :
  pour créer, configurer les objets d'une application
  exemple : Factory, Singleton...
- **pattern de structure** définir l'organisation d'une partie de l'application au sein de l'application générale
  exemple : Composite, Adapter...

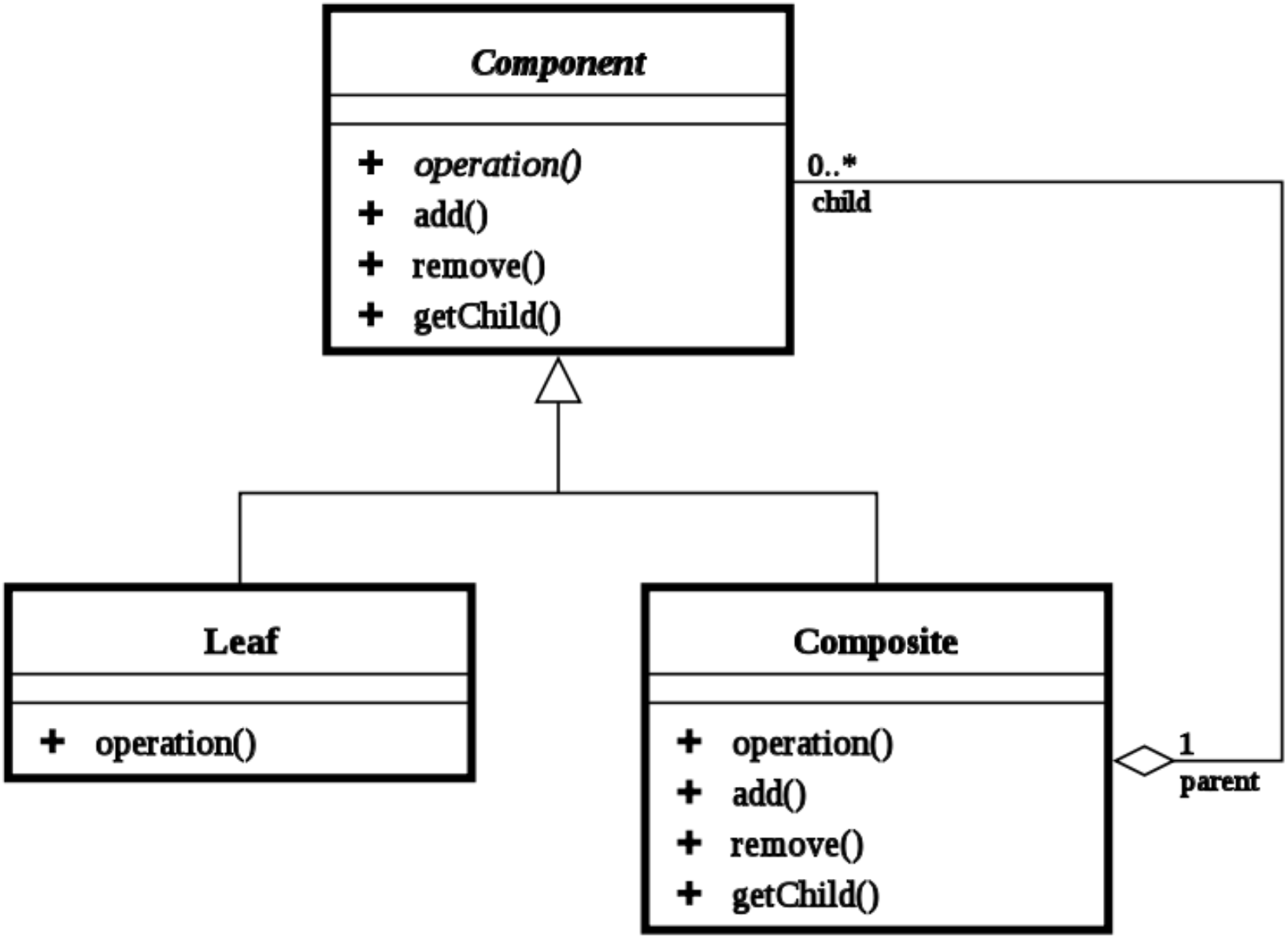
patrons de conception les plus connus : une vingtaine
Combinaison de patrons $\Rightarrow$ **patron d'architecture** (exemple : *Modèle-Vue-Controleur (MVC)*)

Le pattern Composite

Problème : nécessité d'une structure en arbre, et besoin dans certains cas de considérer les nœuds internes et les feuilles de la même manière. *Exemples :*

- un système de fichiers est composé de dossiers et de fichiers. Un dossier pouvant comprendre des fichiers et des dossiers. Certaines opérations, comme la copie ou l'affichage, nécessitent des traitements récursifs qui nécessite de traiter de manière identique les dossiers et fichiers.
- représentation d'objets graphiques : les objets graphiques peuvent être des points, rectangles, cercles...etc ou bien des agglomérats d'autres objets graphiques. Opérations communes : afficher, supprimer par exemple

Le pattern Composite



Le pattern Composite : exemple

```
class ObjectSystem {
public:
string name;
};

class File : public ObjectSystem {
public:
string extension;
};

class Dossier : public ObjectSystem {
public:
vector<ObjectSystem> children;
}

int main() {
Dossier root("root");
File fichier0("Fichier0");
Dossier dossier1("dossier1");
File fichier11("fichier11");
dossier1.add(fichier11);
root.add(dossier1);
}
```

Le pattern Singleton

Problème : dans une application, on veut qu'une classe *A* ne soit instanciée qu'une seule fois

Exemples :

- l'application gère un annuaire de personnes qui doit être unique
- l'application communique avec une autre machine via un unique socket
- les classes *TRUE* et *FALSE* de SmallTalk

Idée : pour empêcher d'instancier : mettre les constructeurs en protected !

Le pattern Singleton : exemple

```
class Annuaire {
public:
    static Annuaire* getInstance();
    void test();
protected:
    Annuaire();
    static Annuaire* instance;
};

Annuaire* Annuaire::instance = NULL;

Annuaire::Annuaire() {

}

Annuaire* Annuaire::getInstance() {
    if (Annuaire::instance == NULL) {
        Annuaire::instance = new Annuaire();
    }
    return Annuaire::instance;
}

void Annuaire::test() {
    cout << "hello world!" << endl;
}
```

Le pattern Singleton : exemple

Utilisation :

```
int main(int argc, char** argv) {

Annuaire::getInstance()->test();

return 0;
}
```

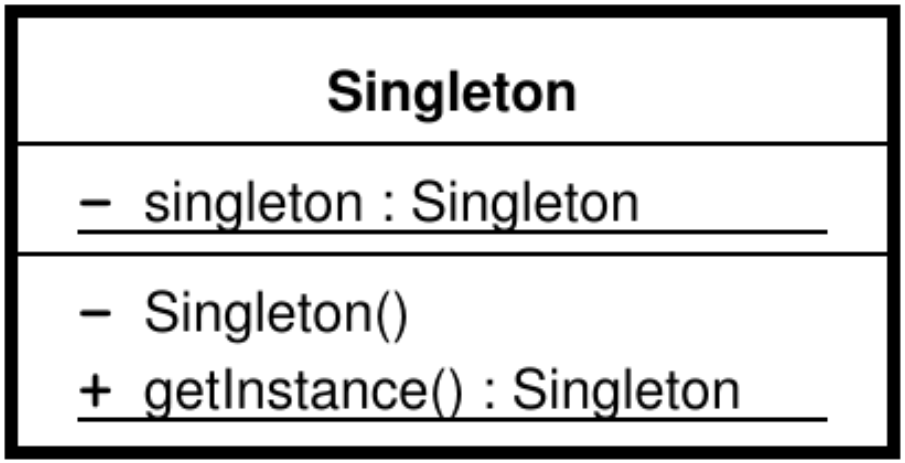
Le pattern Singleton : exemple

Utilisation :

```
int main(int argc, char** argv) {

Annuaire::getInstance()->test();

return 0;
}
```



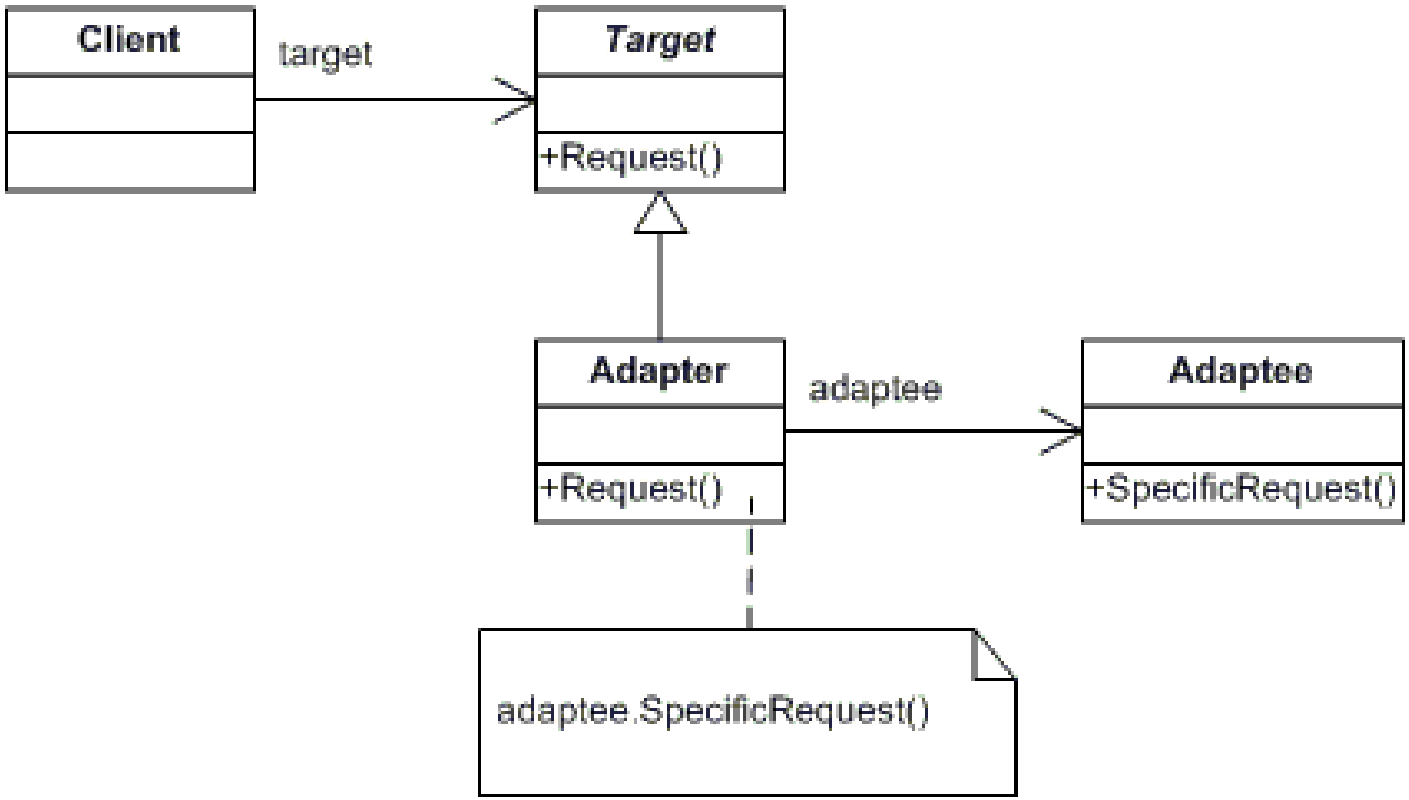
Le pattern Adapter

Problème : on désire utiliser une classe dont ne code n'est pas modifiable/disponible, mais la signature des méthodes ne nous conviennent pas.
Exemples :

- notre application manipule des dates, mais on ne sait pas encore si l'on va définir nous même la classe Date, ou bien utiliser une librairie existante.

Idée : on créée un classe *DateAdapter* qui va faire la liaison entre l'utilisation que l'on veut et la classe Date que l'on désire utiliser

Le pattern Adapter



Le pattern Adapter : exemple

```
class DateAdapter {
public:
virtual string getDate() = 0;
};

class FastDateAdapter : public DateAdapter {
public:
FastDate fastDate;
string getDate();
};

string FastDateAdapter::getDate() {

//ici on utilise les méthodes de l'objet fastDate
//afin de retourner l'objet voulu par notre application

}


int main() {

DateAdapter* date = new FastDateAdapter();
date->getDate()

}
```