Design pattern Visiteur en C++

Hyacinthe MENIET

3 août 2019

Le design pattern Visiteur (**Visitor**) permet d'agir sur la structure de données d'un objet sans avoir à modifier la classe dont l'objet est une instance. En d'autres termes, au lieu d'ajouter des méthodes ou des algorithmes à une classe pour réaliser certaines fonctions, grâce au design pattern Visiteur, le développeur à l'opportunité de passer des références à d'autres classes qui elles implémentent la fonctionnalité désirée. Cela permet aux méthodes et algorithmes de la classe passée en référence d'intervenir sur le comportement de l'objet que le développeur souhaite modifier.
Note : J'utilise sciemment l'appellation anglaise *Design pattern* au lieu des françaises *Patron de conception* et *Motif de conception*. Ceci car c'est généralement sous ce nom que ces techniques de programmation apparaissent dans la littérature technique, y compris francophone.

1. Pré-requis

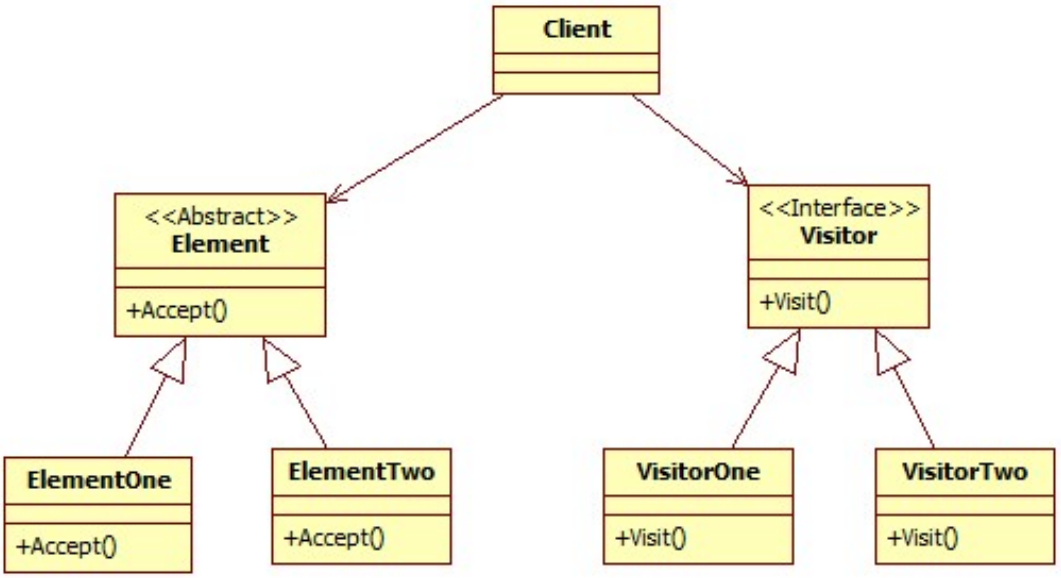
- Vous êtes familier de la programmation orientée objet
- Vous savez ce qu'est un design pattern
- Vous maîtrisez la syntaxe C++
- Vous êtes familier de la compilation séparée
- Vous êtes familier des bibliothèques C++ (ex : STL)
- Vous disposez d'un environnement de développement pour C++, à minima 1 éditeur de texte (ex : vim, emacs, notepad ...) et un compilateur (ex : g++, cygwin, mingw).

Note : Pour limiter les adhérences techniques, j'ai choisis de développer le code ci-dessous à l'aide de vim et g++ sous Linux. Cependant, si vous êtes plus à l'aise avec, sachez que Microsoft Visual C++ Express (Windows) ou Code : :Blocks (Windows, Linux et OSX) feront également l'affaire.

2. Vue d'ensemble

2.1 Vue d'ensemble du design pattern Visiteur

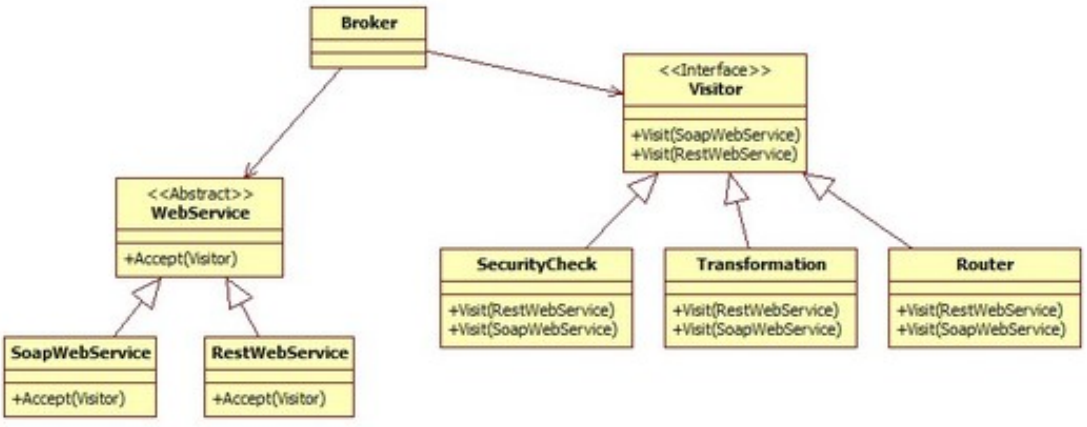
Comme vous l'avez compris, ce design pattern est particulièrement indiqué dans le cas où vous souhaitez appliquer un algorithme à des objets (i.e. des structures de données) mais que cet algorithme est déjà implémenté dans d'autres classes. De plus pour des raisons évidentes vous ne souhaitez pas réécrire cet algorithme et pour diverses autres raisons - dont la clarté - il n'est pas acceptable d'introduire une relation d'héritage entre les objets que vous souhaitez modifier et les classes qui implémentent l'algorithme.
Le design pattern visiteur est construit à partir de plusieurs composants :



— **Les structures de données :** Element (l'abstraction) et ElementOne, ElementTwo (les implémentations).
— **Les classes qui contiennent les algorithmes :** Visitor (l'abstraction) et VisitorOne, VisitorTwo (les implémentations).
— **Le programme principal :** Client qui contient une collection d'Elements que le développeur expose aux Visitors.

2.2 Vue d'ensemble de l'article

Pour illustrer le design pattern Visiteur je vais montrer comment résoudre simplement ce problème un peu simpliste mais qui vous servira de base pour résoudre des problèmes plus pointus :
Nous devons réaliser un courtier de Web services, c'est-à-dire un routeur de web services capables d'appliquer des médiations (ex : filtrage de sécurité, transformation de contenu) à des web services SOAP et REST. Les algorithmes de médiation et de routage existants déjà, la difficulté ici est de l'appliquer à nos structures de données : les web services SOAP et REST.
La solution proposée correspond au schéma UML suivant :



Vous pouvez également télécharger le source C++ du tutoriel Visiteur.

3. Les structures de données

3.1 L'abstraction

Créez la classe abstraite WebService dont la méthode virtuelle pure *Accept* prend en paramètre un *Visitor*. Dans mon cas cela correspond aux en-têtes WebService.h :

```
/*
* WebService.h
*
*       Author: Hyacinthe MENIET
*/
#ifndef WEBSERVICE_H_
#define WEBSERVICE_H_
#include <string>
using namespace std;
namespace dotmyself {
/*
* Abstract base for the class that actually contains the state
* we wish to modify.
*/
class WebService {
public:
enum wsType {Rest,Soap};
protected:
string name;
wsType type;
bool processed;
WebService();
public:
virtual ~WebService();
virtual void Accept(class Visitor *) = 0;
string GetName() const;
void SetName(string);
bool IsProcessed() const;
void SetProcessed (bool);
wsType GetType() const;
void  SetType(wsType);
};
}
#endif /* WEBSERVICE_H_ */
```

Et au source WebService.cpp :

```
/*
* WebService.cpp
*
```

```
*       Author: Hyacinthe MENIET
*/
#include <string>
#include "WebService.h"
using namespace std;
namespace dotmyself {
WebService::~WebService() {}
WebService::WebService() {}
string WebService::GetName() const {return this->name;}
void WebService::SetName( string n ) { this->name = n;}
bool WebService::IsProcessed() const {return this->processed;}
void WebService::SetProcessed(bool p) {this->processed = p;}
WebService::wsType WebService::GetType() const {return this->type
    ;};
void  WebService::SetType(WebService::wsType t) {this->type = t
    ;};
}
```

3.2 Les implémentations

Spécialisez la classe WebService pour REST. Dans mon cas cela correspond aux en-têtes RestWebService.h :

```
/*
* RestWebService.h
*
*       Author: Hyacinthe MENIET
*/
#ifndef RESTWEBSERVICE_H_
#define RESTWEBSERVICE_H_
#include <string>
#include "WebService.h"
using namespace std;
namespace dotmyself {
/**
* Rest implementation of WebService abstraction.
*/
class RestWebService: public dotmyself::WebService {
public:
RestWebService(string n);
virtual ~RestWebService();
void Accept(Visitor *);
};
}
#endif /* RESTWEBSERVICE_H_ */
```

Et au source RestWebService.cpp :

```
/*
* RestWebService.cpp
*
*       Author: Hyacinthe MENIET
*/
#include <string>
#include "RestWebService.h"
#include "Visitor.h"
using namespace std;
namespace dotmyself {
RestWebService::RestWebService(string n) {
this->name = n;
this->type = Rest;
this->processed = false;
}
RestWebService::~RestWebService() {}
void RestWebService::Accept(Visitor *v) {
v->visit(this);
}
}
```

Spécialisez la classe WebService pour SOAP. Dans mon cas cela correspond aux en-têtes SoapWebService.h :

```
/*
* SoapWebService.h
*
*       Author: Hyacinthe MENIET
*/
#ifndef SOAPWEBSERVICE_H_
#define SOAPWEBSERVICE_H_
#include <string>
#include "WebService.h"
using namespace std;
namespace dotmyself {
/*
* Soap implementation of WebService abstraction.
*/
class SoapWebService: public dotmyself::WebService {
public:
SoapWebService(string n);
virtual ~SoapWebService();
void Accept(Visitor *);
};
}
#endif /* SOAPWEBSERVICE_H_ */
```

Et au source SoapWebService.cpp :

```
/*
* SoapWebService.cpp
*
*       Author: Hyacinthe MENIET
*/
#include <string>
#include "SoapWebService.h"
#include "Visitor.h"
using namespace std;
namespace dotmyself {
SoapWebService::SoapWebService(string n) {
this->name = n;
this->type = Soap;
this->processed = false;
}
SoapWebService::~SoapWebService() {}
void SoapWebService::Accept(Visitor *v) {
v->visit(this);
}
}
```

4. Les algorithmes

Créez l'interface *Visitor* qui possède 2 méthodes virtuelles pures *Visit*, une pour chaque type de web service. Dans mon cas cela correspond aux en-têtes Visitor.h :

```
/*
* Visitor.h
*
*       Author: Hyacinthe MENIET
*/
#ifndef VISITOR_H_
#define VISITOR_H_
#include "RestWebService.h"
#include "SoapWebService.h"
using namespace dotmyself;
namespace dotmyself {
/*
* Interface for the implementation that contains the functional
    algorithms.
*/
class Visitor {
public:
virtual ~Visitor();
virtual void visit(SoapWebService*) =0;
virtual void visit(RestWebService*) =0;
protected:
Visitor();
```

```
};
}
#endif /* VISITOR_H_ */
```

Et au source Visitor.cpp :

```
/*
* Visitor.cpp
*
*       Author: Hyacinthe MENIET
*/
#include "Visitor.h"
namespace dotmyself {
Visitor::~Visitor() {}
Visitor::Visitor() {}
}
```

4.1 L'abstraction

4.2 Les implémentations

Implémentez l'interface Visitor pour les vérifications de sécurité. Dans mon cas cela correspond aux en-têtes SecurityCheck.h :

```
/*
* SecurityCheck.h
*
*       Author: Hyacinthe MENIET
*/
#ifndef SECURITYCHECK_H_
#define SECURITYCHECK_H_
#include "Visitor.h"
namespace dotmyself {
/*
* Visitor implementation containing security check algorithms.
*/
class SecurityCheck: public dotmyself::Visitor {
public:
SecurityCheck();
virtual ~SecurityCheck();
void visit(SoapWebService*);
void visit(RestWebService*);
};
}
#endif /* SECURITYCHECK_H_ */
```

Et au source SecurityCheck.cpp :

```
/*
* SecurityCheck.cpp
*
*       Author: Hyacinthe MENIET
*/
#include <iostream>
#include "SecurityCheck.h"
using namespace std;
namespace dotmyself {
SecurityCheck::SecurityCheck() {}
SecurityCheck::~SecurityCheck() {}
void SecurityCheck::visit(RestWebService* ws) {
cout << "[REST] Validating against XML schema" << endl;
}
void SecurityCheck::visit(SoapWebService* ws) {
cout << "[SOAP] Validating against WSDL" << endl;
}
}
```

Implémentez l'interface Visitor pour la transformation de contenu. Dans mon cas cela correspond aux en-têtes Transformation.h :

```
/*
* Transformation.h
*
*       Author: Hyacinthe MENIET
*/
#ifndef TRANSFORMATION_H_
#define TRANSFORMATION_H_
#include "Visitor.h"
namespace dotmyself {
/*
* Visitor implementation containing transformation algorithms.
*/
class Transformation: public dotmyself::Visitor {
public:
Transformation();
virtual ~Transformation();
void visit(SoapWebService*);
void visit(RestWebService*);
};
}
#endif /* TRANSFORMATION_H_ */
```

Et au source Transformation.cpp :

```
/*
* Transformation.cpp
```

```
*
*       Author: Hyacinthe MENIET
*/
#include <iostream>
#include "Transformation.h"
using namespace std;
namespace dotmyself {
Transformation::Transformation() {}
Transformation::~Transformation() {}
void Transformation::visit(RestWebService* ws) {
cout << "[REST] Converting to JSON ..." << endl;
}
void Transformation::visit(SoapWebService* ws) {
cout << "[SOAP] Removing MTOM attachment ..." << endl;
}
}
```

Implémentez l'interface Visitor pour le routage. Dans mon cas cela correspond aux en-têtes Router.h :

```
/*
* Router.h
*
*       Author: Hyacinthe MENIET
*/
#ifndef ROUTER_H_
#define ROUTER_H_
#include "Visitor.h"
namespace dotmyself {
/*
* Visitor implementation containing routing algorithms.
*/
class Router: public dotmyself::Visitor {
public:
Router();
virtual ~Router();
void visit(SoapWebService*);
void visit(RestWebService*);
};
}
#endif /* ROUTER_H_ */
```

Et au source Router.cpp :

```
/*
* Router.cpp
*
*       Author: Hyacinthe MENIET
*/
```

```
#include <iostream>
#include "Router.h"
using namespace std;
namespace dotmyself {
Router::Router() {}
Router::~Router() {}
void Router::visit(RestWebService* ws) {
ws->SetProcessed(true);
cout << "[REST] Updating URI" << endl;
}
void Router::visit(SoapWebService* ws) {
ws->SetProcessed(true);
cout << "[SOAP] Updating port and operation" << endl;
}
}
```

5. Le courtier

Terminez par le courtier qui est le programme principal. Dans mon cas cela correspond au source Broker.cpp :

```
//============================================================================

// Name        : Broker.cpp
// Author      : Hyacinthe MENIET
// Description : A simple program that mediates and routes client
    requests to web services providers.
//============================================================================

#include <iostream>
#include "dotmyself/RestWebService.h"
#include "dotmyself/SoapWebService.h"
#include "dotmyself/SecurityCheck.h"
#include "dotmyself/Transformation.h"
#include "dotmyself/Router.h"
using namespace std;
using namespace dotmyself;
int main() {
WebService *wss[] =
{
new RestWebService("RestZero"), new RestWebService("RestOne"),
new SoapWebService("SoapTwo"), new RestWebService("RestThree"),
new SoapWebService("SoapFour")
};
SecurityCheck sc;
Transformation tf;
Router rt;
int i=0;
```

```
for (i = 0; i<5; i++)
{
cout << endl << "Processing " << wss[i]->GetName() << ":" << endl
    ;
wss[i]->Accept(&sc);
wss[i]->Accept(&tf);
wss[i]->Accept(&rt);
}
return 0;
}
```

6. Compilation et exécution

Pour ceux qui ne disposent pas d'environnement de développement intégré et ont téléchargé le source ci-dessus, les lignes suivantes vous permettront de compiler le projet. Ces commandes sont à entrer dans un terminal Linux et supposent que vous ayez le compilateur g++ installé sur votre distribution.
Compilation des classes :

```
$ mkdir -p target/classes/dotmyself target/executable
$ g++ -Wall -c -o"target/classes/dotmyself/RestWebService.o" "src
    /main/cpp/dotmyself/RestWebService.cpp"
$ g++ -Wall -c -o"target/classes/dotmyself/Router.o" "src/main/
    cpp/dotmyself/Router.cpp"
$ g++ -Wall -c -o"target/classes/dotmyself/SecurityCheck.o" "src/
    main/cpp/dotmyself/SecurityCheck.cpp"
$ g++ -Wall -c -o"target/classes/dotmyself/SoapWebService.o" "src
    /main/cpp/dotmyself/SoapWebService.cpp"
$ g++ -Wall -c -o"target/classes/dotmyself/Transformation.o" "src
    /main/cpp/dotmyself/Transformation.cpp"
$ g++ -Wall -c -o"target/classes/dotmyself/Visitor.o" "src/main/
    cpp/dotmyself/Visitor.cpp"
$ g++ -Wall -c -o"target/classes/dotmyself/WebService.o" "src/
    main/cpp/dotmyself/WebService.cpp"
$ g++ -Wall -c -o"target/classes/Broker.o" "src/main/cpp/Broker.
    cpp"
```

Quelques explications :

- `src/main/cpp` est le dossier qui contient les sources
- `target/classes` est le dossier qui contient les résultats des compilations
- `target/executable` est le dossier qui contiendra l'exécutable

Génération de l'exécutable :

```
$ g++  -o"target/executable/Broker" ./target/classes/dotmyself/
    RestWebService.o ./target/classes/dotmyself/Router.o ./target/
    classes/dotmyself/SecurityCheck.o ./target/classes/dotmyself/
    SoapWebService.o ./target/classes/dotmyself/Transformation.o
```

```
./target/classes/dotmyself/Visitor.o ./target/classes/
dotmyself/WebService.o ./target/classes/Broker.o
```