Informatique C++ : Le Patron Observer

Boris Alexandre Baude - ENS Rennes - Informatique

Compte rendu TP4

1 Introduction

L'objectif est de coder un exemple utilisant ce patron sur des valeurs simples. Cela implique de définir une interface Subject pour accéder et modifier une valeur de type String et gérer les abonnements d'observateurs. De plus, une interface Observer avec une méthode update est requise. Le TP inclut aussi la création de classes SubjectImpl et ObserverImpl, qui implémentent respectivement Subject et Observer. SubjectImpl utilise la classe list de la bibliothèque standard, tandis qu'ObserverImpl affiche un message sur la console lors des mises à jour. Le main.cpp sert de Mutator dans ce contexte.

2 Consignes

Il s'agit de coder un exemple de patrons Observer sur des valeurs simples.
Définissez :

— une interface **Subject**, qui permet d'accéder et de modifier une valeur de type String, et d'abonner des observateurs :

```
std::string getValue();
void setValue(std::string value);
boolean isRegistered(Observer* o);
void registerObserver(Observer* o);
void unregisterObserver(Observer* o);

```

— une interface **Observer** avec les opérations suivantes :

```
void update(Subject* s);

```

— une classe **SubjectImpl** mettant en œuvre Subject, utilisant la classe list de la bibliothèque standard.
— une classe **ObserverImpl** mettant en œuvre Observer et imprimant un message sur la console lors des mise à jour du sujet (cf le patron de conception Observer).
— le Mutator sera le **main.cpp**.

3 Telechargement de C++ sur VS code sur Windows et environnement Linux

Pour faire marcher C++ sur Vs code tous en étant sur Windows, on doit installer choco, make et un WSL (Subsystem Linux) Suite à cela, on le fait intéragir avec VS code avec Ubuntu via une extension WSL.

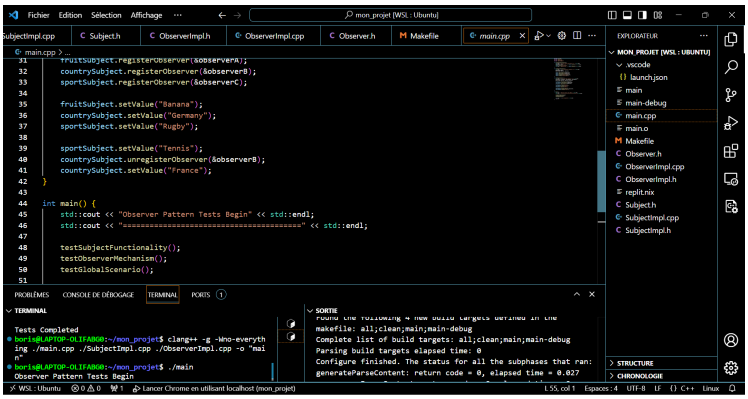

FIGURE 1 – Interface VS code sur Windows mais intérragissant sur Linux (Ubuntu)

4 Interface subject : Subject.h

Cette interface représente le "sujet" qui sera observé. Elle doit permettre : L'accès (getValue) et la modification (setValue) d'une valeur de type string. L'enregistrement (registerObserver) et la désinscription (unregisterObserver) d'observateurs (qui sont des instances d'une classe implémentant l'interface Observer). La vérification (isRegistered) si un observateur est déjà enregistré.Constructeur et Destructeur : Utilise les versions par défaut (= default) pour le constructeur et le destructeur.

```
#pragma once
#include <string>
#include <list>

class Observer;

class Subject {
public:
    // Constructeur par d faut
    Subject() {};

    // Destructeur virtuel
    virtual ~Subject() {};

    // R cup re la valeur actuelle du sujet
    virtual std::string getValue() = 0;

    // D finit la valeur du sujet
    virtual void setValue(const std::string &value) = 0;

    // Enregistre un observateur, s'il n'est pas d  j   enregistr
    virtual void registerObserver(Observer* observer) = 0;

    // D senregistre un observateur, s'il est enregistr
    virtual void unregisterObserver(Observer* observer) = 0;

    // V rifie si un observateur est enregistr
    virtual bool isRegisteredObserver(Observer* observer) = 0;

private:
    // Longueur ou autre propri t  (non utilis e dans cet exemple)
    int length;
};
```

5 Interface Observer

Cette interface doit être implémentée par les "observateurs". Elle contient une méthode update, qui sera appelée chaque fois que l'état du sujet change. Cette méthode prend en paramètre une référence au sujet (Subject*).

5.1 Observermlpl.h

Le code dans ObserverImpl.h définit la classe ObserverImpl, qui est une implémentation concrète de l'interface Observer. Cette classe hérite de Observer et doit donc implémenter la méthode virtuelle update. Le constructeur ObserverImpl() et le destructeur virtuel ObserverImpl() sont définis, ce dernier étant marqué avec le mot-clé override pour indiquer qu'il remplace une méthode virtuelle de la classe de base. La méthode update prend un pointeur vers un Subject et est également marquée avec override, soulignant qu'elle remplace la méthode virtuelle de la classe parente Observer.

```
#pragma once// Pr vient les inclusions multiples du m me fichier d'en-t te
#include <string>// Inclut la biblioth que standard des cha nes de caract res pour une utilisation
     ventuelle   dans la classe
#include "Observer.h"// Inclut l'en-t te de la classe de base 'Observer'

class ObserverImpl : public Observer {
// D clare la classe 'ObserverImpl' qui h rite de 'Observer'

public:
    // D claration des m thodes publiques

    ObserverImpl();
    // Constructeur par d faut de la classe ObserverImpl

    virtual ~ObserverImpl() override;
    // Destructeur virtuel de la classe ObserverImpl
    // Le mot-cl  'override' indique que cette m thode remplace une m thode virtuelle de la classe de base

```

```
    void update(Subject* s) override;
    // M thode pour mettre    jour l' tat  de l'observateur
    // 's' est un pointeur vers le sujet observ
    // 'override' indique que cette m thode remplace une m thode virtuelle de la classe de base
};
```

5.2 Observermlpl.cpp

ObserverImpl.cpp fournit l'implémentation concrète d'un observateur dans le patron de conception Observer. Lorsque le sujet (un objet de type Subject) change d'état, cette implémentation de l'observateur (ObserverImpl) est notifiée et réagit en affichant la nouvelle valeur du sujet. Ce comportement est typique du rôle d'un observateur dans ce modèle de conception, où il réagit aux changements d'état des sujets qu'il surveille.

```
#include "ObserverImpl.h"// Inclut l'en-t te de la classe ObserverImpl
#include <iostream> // Inclut la biblioth que iostream pour les op rations d'entr e/sortie (par exemple,
    std::cout et std::cerr)

ObserverImpl::ObserverImpl() {
  // Constructeur de la classe ObserverImpl
  std::cerr << "New Observer constructed\n";
  // Affiche un message dans le flux d'erreurs standard lors de la construction d'un nouvel observateur
};

ObserverImpl::~ObserverImpl() {
  // Destructeur de la classe ObserverImpl
  std::cerr << "Observer destructed\n";
  // Affiche un message dans le flux d'erreurs standard lors de la destruction de l'observateur
}

void ObserverImpl::update(Subject *s) {
  // M thode 'update' qui est appel e pour notifier l'observateur d'un changement dans le sujet
  std::cout << "Subject updated : " << s->getValue() << std::endl;
  // Affiche un message dans la sortie standard indiquant que le sujet a   t   mis    jour
  // 's->getValue()' appelle la m thode 'getValue' sur l'objet sujet pour obtenir sa valeur actuelle
};
```

5.3 Observer.h

Le fichier Observer.h définit une classe abstraite Observer qui joue un rôle central dans le patron de conception Observer. Ce fichier d'en-tête déclare un constructeur et un destructeur par défaut, ainsi qu'une méthode virtuelle pure update. La méthode update est destinée à être implémentée par des sous-classes concrètes d'Observer. Elle prend un pointeur vers un Subject en paramètre, permettant à l'observateur d'être informé et de réagir aux changements d'état du sujet observé. La déclaration anticipée de la classe Subject est nécessaire pour référencer Subject dans la définition d'Observer.

```
#pragma once  // Pr vient les inclusions multiples du m me fichier d'en-t te
#include "Subject.h" // Inclut l'en-t te de la classe 'Subject', n cessaire pour la r f rence    'Subject
    ' dans la classe 'Observer'
class Subject; // D claration anticip e de la classe 'Subject' pour indiquer son existence avant son
    utilisation compl te

class Observer {
public:
    // Section publique de la classe Observer

    Observer(){};
    // Constructeur par d faut de la classe Observer
    // Il ne fait rien de sp cial dans ce cas

    virtual ~Observer(){};
    // Destructeur virtuel de la classe Observer
    // Important pour une classe destin e     tre  h rit e , permettant une destruction propre des sous-
    classes

    virtual void update(Subject* s) = 0;
    // M thode virtuelle pure 'update', qui doit  tre  impl ment e par les sous-classes
    // Prend un pointeur vers un 'Subject' en tant que param tre
    // Cette m thode est destin e     tre  appel e pour notifier l'observateur d'un changement dans le
    sujet
};
```

6 Classe SubjectImpl

6.1 Classe SubjectImpl.h

Le code dans SubjectImpl.h définit la classe SubjectImpl, qui est une implémentation de l'interface Subject. Cette classe hérite de Subject et implémente ses méthodes virtuelles, y compris getValue, setValue, registerObserver, unregisterObserver, et isRegisteredObserver. Le constructeur SubjectImpl() initialise la classe, tandis que le destructeur virtuel SubjectImpl() assure une destruction appropriée pour la polymorphie. La classe gère une liste d'observateurs (Observerlist) et maintient l'état actuel du sujet (subj). Chaque méthode joue un rôle spécifique dans la gestion de l'état du sujet et la notification des observateurs.

```
#pragma once// Pr vient les inclusions multiples du m me fichier d'en-t te
#include <list>// Inclut la biblioth que standard de la liste pour la gestion des listes
#include <string>// Inclut la biblioth que standard des cha nes de caract res
#include "Subject.h"// Inclut l'en-t te de la classe de base 'Subject'
#include "Observer.h"// Inclut l'en-t te de la classe 'Observer'

class SubjectImpl : public Subject {
// D clare la classe 'SubjectImpl' qui h rite de 'Subject'
public:
    // D claration des m thodes publiques

    // Constructeur par d faut
    SubjectImpl();

    // Destructeur virtuel
    virtual ~SubjectImpl() override;
    // Le mot-cl  'virtual' assure que le destructeur est correct pour la polymorphie
    // 'override' signifie que cette m thode remplace une m thode virtuelle de la classe de base

    // R cup re l' tat  actuel du sujet
    std::string getValue() override;
    // 'override' indique que cette m thode remplace une m thode virtuelle de la classe de base

    // Met    jour l' tat  du sujet
    void setValue(const std::string &s) override;
    // 's' est la nouvelle valeur    attribuer    'subj'

    // Ajoute un observateur    la liste, si non pr sent
    void registerObserver(Observer* o) override;
    // 'o' est le pointeur vers l'observateur    enregistrer

    // Retire un observateur de la liste, si pr sent
    void unregisterObserver(Observer* o) override;
    // 'o' est le pointeur vers l'observateur    d senregistrer

    // V rifie si un observateur est enregistr
    bool isRegisteredObserver(Observer* o) override;
    // 'o' est le pointeur vers l'observateur dont on v rifie l'enregistrement
    // Retourne 'true' si l'observateur est enregistr , 'false' sinon

private:
    // D claration des membres priv s de la classe

    std::list<Observer*> Observer_list;
    // Liste contenant les pointeurs vers les observateurs enregistr s

    std::string subj = "subject initial";
    // Cha ne de caract res repr sentant l' tat  initial du sujet
};
```

6.2 Classe SubjectImpl.cpp

SubjectImpl.cpp met en œuvre la classe SubjectImpl, qui est une extension de la classe abstraite Subject. Le constructeur et le destructeur de cette classe affichent des messages lors de leur exécution. La méthode getValue renvoie l'état actuel du sujet, tandis que setValue met à jour cet état et notifie tous les observateurs de ce changement. Les méthodes registerObserver et unregisterObserver permettent d'ajouter ou de retirer des observateurs de la liste Observerlist. La méthode isRegisteredObserver vérifie si un observateur spécifique est déjà enregistré. Ce fichier illustre l'interaction entre le sujet et ses observateurs dans le cadre du patron de conception Observer.

```
#include "SubjectImpl.h"// Inclusion de l'en-t te de la classe SubjectImpl
#include "Observer.h"// Inclusion de l'en-t te de la classe Observer
#include <algorithm> // Pour std::find // Inclusion de la biblioth que algorithm pour utiliser la fonction
    std::find
#include <iostream>// Inclusion de la biblioth que iostream pour les op rations d'entr e/sortie
```

```cpp
#include <list> // Inclusion de la biblioth que list pour utiliser les listes
#include <string> // Inclusion de la biblioth que string pour utiliser les cha nes de caract res
#include <typeinfo> // Inclusion de la biblioth que typeinfo pour les informations de type

SubjectImpl::SubjectImpl() {
  std::cerr << "SubjectImpl: Nouveau sujet construit\n";
}; // Constructeur de la classe SubjectImpl qui affiche un message lors de la cr ation

SubjectImpl::~SubjectImpl() {
  std::cerr << "SubjectImpl: Sujet d truit\n";
}; // Destructeur de la classe SubjectImpl qui affiche un message lors de la destruction

std::string SubjectImpl::getValue() {
  // Retourne l' tat  actuel du sujet
  return subj;
}; // M thode pour obtenir la valeur actuelle du sujet

void SubjectImpl::setValue(const std::string &s) {
  // Met    jour l' tat  du sujet avec la nouvelle valeur
  subj = s;
  std::cerr << "SubjectImpl:  tat  modifi  en : " << subj << std::endl;
  // Notifie tous les observateurs enregistr s du changement
  for (auto observer : Observer_list) {
    observer->update(this);
  }
}; // M thode pour d finir la valeur du sujet et notifier les observateurs

void SubjectImpl::registerObserver(Observer *o) {
  // V rifie si l'observateur n'est pas d  j  enregistr
  if (!isRegisteredObserver(o)) {
    // Enregistre le nouvel observateur
    Observer_list.push_front(o);
    std::cout << "SubjectImpl: Nouvel observateur enregistr \n";
  } else {
    std::cout << "SubjectImpl: Observateur d  j  enregistr \n";
  }
}; // M thode pour enregistrer un nouvel observateur

void SubjectImpl::unregisterObserver(Observer *o) {
  // V rifie si l'observateur est enregistr
  if (isRegisteredObserver(o)) {
    // Retire l'observateur de la liste
    Observer_list.remove(o);
    std::cout << "SubjectImpl: Observateur d senregistr \n";
  } else {
    std::cout << "SubjectImpl: Tentative de d senregistrement d'un observateur non enregistr \n";
  }
}; // M thode pour d senregistrer un observateur

bool SubjectImpl::isRegisteredObserver(Observer *o) {
  // V rifie si l'observateur est dans la liste des observateurs
  return std::find(Observer_list.begin(), Observer_list.end(), o) != Observer_list.end();
}; // M thode pour v rifier si un observateur est enregistr
```

7 Le makefile

Le Makefile fourni est un script utilisé avec l'outil make pour automatiser la compilation de programmes en C++. La cible par défaut, all, déclenche la construction de l'exécutable main. Le compilateur spécifié est clang++, et les options de compilation incluent -g pour le débogage et -Wno-everything pour désactiver tous les avertissements. Les sources (fichiers .cpp) et les en-têtes (fichiers .h) sont automatiquement trouvés dans le répertoire courant et ses sous-répertoires, à l'exception des dossiers .ccls-cache. La cible main compile ces fichiers sources en un exécutable main, tandis que main-debug est une variante de compilation pour le débogage avec l'optimisation désactivée (-O0). Enfin, la cible clean permet de nettoyer le projet en supprimant les exécutables générés. Ce Makefile facilite la gestion de la compilation, offrant une méthode standardisée et automatisée pour construire des projets en C++.

```
all: main

CXX = clang++
override CXXFLAGS += -g -Wno-everything

SRCS = $(shell find . -name '.ccls-cache' -type d -prune -o -type f -name '*.cpp' -print | sed -e 's/ /\\ /g')
HEADERS = $(shell find . -name '.ccls-cache' -type d -prune -o -type f -name '*.h' -print)

```

```
main: $(SRCS) $(HEADERS)
  $(CXX) $(CXXFLAGS) $(SRCS) -o "$@"

main-debug: $(SRCS) $(HEADERS)
  $(CXX) $(CXXFLAGS) -O0 $(SRCS) -o "$@"

clean:
  rm -f main main-debug
```

8 Le mutator (main.cpp)

Le main.cpp du programme démontre l'utilisation du patron de conception Observer en C++. Il comprend trois fonctions de test : testSubjectFunctionality, testObserverMechanism, et testGlobalScenario.

testSubjectFunctionality vérifie la fonctionnalité de base de la classe SubjectImpl, en créant une instance de SubjectImpl et en testant la mise à jour de sa valeur.

testObserverMechanism teste le mécanisme d'observation en créant un SubjectImpl et deux ObserverImpl. Les observateurs sont enregistrés auprès du sujet, le sujet est mis à jour, et les observateurs sont notifiés. Ensuite, un observateur est désenregistré et une nouvelle mise à jour est effectuée pour montrer que seul l'observateur enregistré reçoit la notification.

testGlobalScenario crée un scénario plus complexe avec plusieurs sujets (SubjectImpl) et observateurs (ObserverImpl), où chaque observateur est enregistré auprès d'un sujet différent. Les sujets sont mis à jour et les réactions des observateurs sont observées.

```
#include <iostream> // Inclut la biblioth que iostream pour les op rations d'entr e/sortie
#include "Subject.h"// Inclut l'en-t te de la classe de base 'Subject'
#include "SubjectImpl.h" // Inclut l'en-t te de la classe 'SubjectImpl'
#include "ObserverImpl.h" // Inclut l'en-t te de la classe 'ObserverImpl'

// D finit une fonction pour tester la fonctionnalit  de 'SubjectImpl'
void testSubjectFunctionality() {
    std::cout << "=== Test Subject Functionality ===" << std::endl;
    // Affiche un titre pour cette section de test

    SubjectImpl subject;
    // Cr e une instance de 'SubjectImpl'

    std::cout << "Initial value: " << subject.getValue() << std::endl;
    // Affiche la valeur initiale du sujet

    subject.setValue("New Value");
    // Modifie la valeur du sujet

    std::cout << "Updated value: " << subject.getValue() << std::endl;
    // Affiche la valeur mise    jour du sujet
}

// D finit une fonction pour tester le m canisme d'observation
void testObserverMechanism() {
    std::cout << "\n=== Test Observer Mechanism ===" << std::endl;
    // Affiche un titre pour cette section de test

    SubjectImpl subject;
    // Cr e une instance de 'SubjectImpl'

    ObserverImpl observer1, observer2;
    // Cr e deux instances de 'ObserverImpl'

    subject.registerObserver(&observer1);
    subject.registerObserver(&observer2);
    // Enregistre les observateurs aupr s du sujet

    subject.setValue("Notify Observers");
    // Met    jour la valeur du sujet, ce qui d clenche la notification des observateurs

    subject.unregisterObserver(&observer2);
    // D senregistre le deuxi me observateur

    subject.setValue("Second Update");
    // Met    jour la valeur du sujet une deuxi me fois
}

// D finit une fonction pour tester un sc nario global avec plusieurs sujets et observateurs
void testGlobalScenario() {
    std::cout << "\n=== Test Global Scenario ===" << std::endl;
```

```
    // Affiche un titre pour cette section de test

    SubjectImpl fruitSubject, countrySubject, sportSubject;
    // Cr e trois sujets diff rents

    ObserverImpl observerA, observerB, observerC;
    // Cr e trois observateurs diff rents

    fruitSubject.registerObserver(&observerA);
    countrySubject.registerObserver(&observerB);
    sportSubject.registerObserver(&observerC);
    // Enregistre diff rents observateurs pour chaque sujet

    fruitSubject.setValue("Banana");
    countrySubject.setValue("Germany");
    sportSubject.setValue("Rugby");
    // Met    jour les valeurs de chaque sujet

    sportSubject.setValue("Tennis");
    countrySubject.unregisterObserver(&observerB);
    countrySubject.setValue("France");
    // Fait d'autres mises    jour et modifie les enregistrements des observateurs
}

// Point d'entr e du programme
int main() {
    std::cout << "Observer Pattern Tests Begin" << std::endl;
    std::cout << "=======================================" << std::endl;
    // Affiche un en-t te pour le d but des tests

    // Appelle les fonctions de test
    testSubjectFunctionality();
    testObserverMechanism();
    testGlobalScenario();

    std::cout << "\nTests Completed" << std::endl;
    // Affiche un message indiquant la fin des tests
    return 0;
    // Renvoie 0 pour indiquer que le programme s'est termin  avec succ s
}
```

9 Résultats

On observe donc en faisant la commande : clang++ -g -Wno-everything ./main.cpp ./SubjectImpl.cpp ./ObserverImpl.cpp -o "main" et ./main que le patron oberver marche bien :

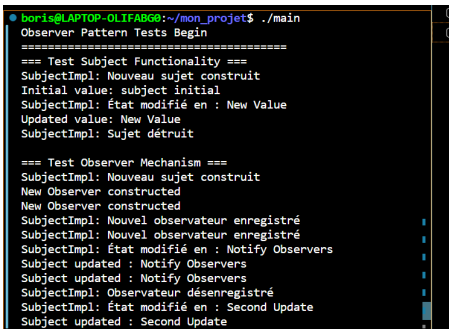

FIGURE 2 – interface VS code sur Windows mais intérragissant sur Linux (Ubuntu)

FIGURE 3 – Interface VS code sur Windows mais intérragissant sur Linux (Ubuntu)

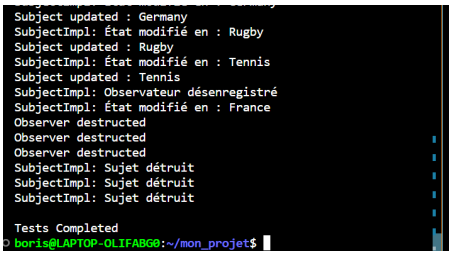

FIGURE 4 – interface VS code sur Windows mais intérragissant sur Linux (Ubuntu)

10 Conclusion

Le patron de conception Observer en C++ peut mettre en avant la réussite de l'implémentation du modèle Observer, comme illustré par les tests réussis. Elle peut souligner comment ce modèle permet une interaction efficace entre les sujets et les observateurs dans différents scénarios. La capacité à s'adapter dynamiquement aux changements de l'état du sujet et la réactivité des observateurs démontrent l'utilité du patron de conception Observer dans des applications réelles. Cette conclusion peut également évoquer les perspectives d'amélioration ou d'extension du projet, par exemple en intégrant des fonctionnalités supplémentaires ou en explorant d'autres aspects de la programmation orientée objet en C++.