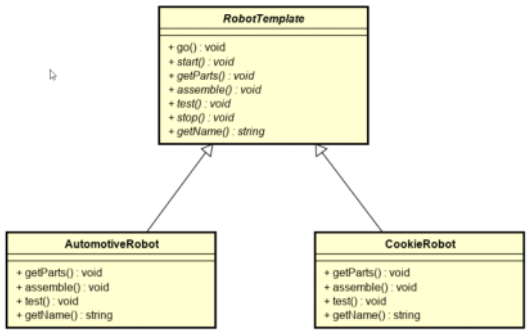
Behavioral Pattern

Wednesday, June 30, 2021 5:39 PM

Template Pattern:

- Template Method Pattern được áp dụng để định nghĩa ra cấu trúc chung, bộ khung (hay còn gọi skeleton – xương sống) chung cho một xử lý nào đó ở class cha (base class) và cho phép các lớp con (subclass) định nghĩa lại một số step trong bộ khung mà không làm thay đổi cấu trúc chung của xử lý. Design Pattern này có lẽ là pattern dễ hiểu, dễ áp dụng và quen thuộc nhất trong tất cả các patterns.



- Nếu phân tích kỹ chúng ta sẽ nhận thấy rằng Cookie Robot có một số điểm chung với Automotive Robot, nó cũng start, stop giống như Automotive Robot nhưng cần làm công việc khác. Ví dụ: hàm *getParts* bây giờ không phải là "Getting a carburetor …" nữa mà phải là " Getting flour and sugar".
- Sau đó, chúng ta biến nó thành template, bằng cách cho phép các class con (subclass) định nghĩa lại (hay override) công việc cụ thể của các step trong template đó nếu cần thiết. Đối với Cookie Robot thì chúng ta sẽ cần override các hàm *getParts*, *assemble*, và *test*.

Example:

// Create Class Template:

```
class RobotTemplate
{
Public:
   virtual void start()
   {
      cout << "Starting ..." << endl;
   }
   virtual void getParts()
   {
      cout << "Getting parts ..." << endl;
   }
   virtual void assemble()
   {
      cout << "Assembling ..." << endl;
   }
   virtual void test()
   {
      cout << "Testing ..." << endl;
   }
   virtual void stop()
   {
      cout << "Stopping ..." << endl;
   }
   virtual std::string getName() = 0;

   void go()
   {
      start();
      getParts();
      assemble();
      test();
      stop();
   }
};
```

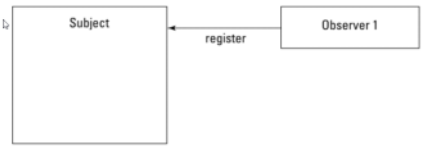
// 1 số đối tượng sử dụng chung nhưng 1 số hành động cần định nghĩa lại sẽ overide lại
//Create **Class *AutomotiveRobot***

```
class AutomotiveRobot : public RobotTemplate
{
public:
   AutomotiveRobot() {}

   void getParts()
   {
      cout << "Getting a carburetor ..." << endl;
   }

   void assemble()
   {
      cout << "Installing the carburetor ..." << endl;
   }

   void test()
   {
      cout << "Revving the engine ..." << endl;
   }

   std::string getName()
   {
      return "Automotive Robot";
   }
};
```

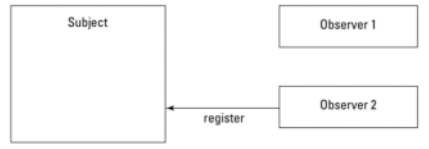
Observer Pattern:

- Observer Pattern còn có thể gọi là Publish-Subscribe Pattern, là design pattern dùng để tạo ra mối quan hệ phụ thuộc one-to-many giữa các đối tượng, khi một đối tượng thay đổi trạng thái, tất cả các phụ thuộc của nó sẽ được thông báo và cập nhật tự động.

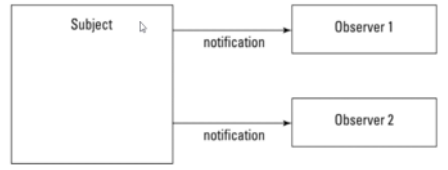
Cơ chế hoạt động:

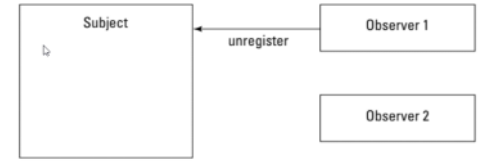


Một Subject có thể có nhiều Observer đăng ký nên Observer 2 cũng có thể đăng ký giống như Observer 1 →

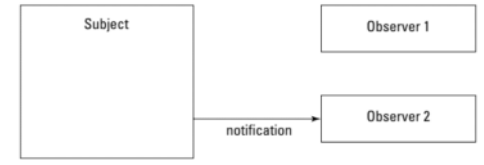


Subject sẽ lưu thông tin về các tất cả các observer đã đăng ký. Khi có sự kiện xảy ra, subject sẽ thông báo (notify) đến tất cả các observer đã đăng ký →

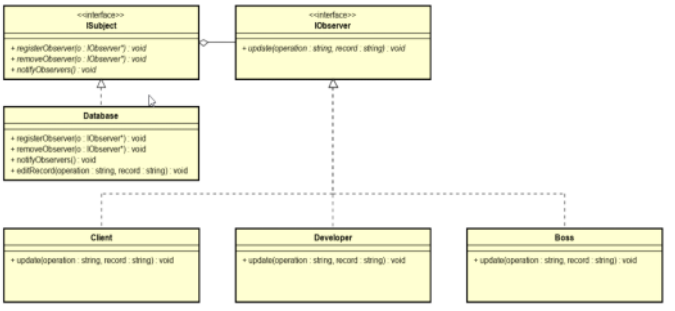




Khi Observer 1 đã unregister thì nó sẽ không còn nhận được notification từ subject nữa. Lúc này chỉ c
Observer 2 nhận được notification →



Class diagram:



Example:

//Create Isubject interface:

```
class ISubject
{
public:
   virtual void registerObserver(IObserver* o) = 0;
   virtual void removeObserver(IObserver* o) = 0;
   virtual void notifyObservers() = 0;
};
```

```
    {
        return "Automotive Robot";
    }
};
```

//Tương tự tạo **Class *CookieRobot***

```
class CookieRobot : public RobotTemplate
{
public:
    CookieRobot() {}

    void getParts()
    {
        cout << "Getting flour and sugar ..." << endl;
    }

    void assemble()
    {
        cout << "Baking a cookie ..." << endl;
    }

    void test()
    {
        cout << "Crunching a cookie ..." << endl;
    }

    std::string getName()
    {
        return "Cookie Robot";
    }
};
```

//Main: sử dụng khi cần đến loại robot nào

```
int main()
{
    AutomotiveRobot automotiveRobot;
    CookieRobot cookieRobot;
    cout << automotiveRobot.getName() << ":" << endl;
    automotiveRobot.go();
    cout << "----------" << endl;
    cout << cookieRobot.getName() << ":" << endl;
    cookieRobot.go();
    return 0;
}
```

State Pattern

- State Pattern cho phép một đối tượng có thể thay đổi hành vi (behavior) của nó dựa trên trạng thái bên trong (internal state). Do đó State Pattern phù hợp để áp dụng trong trường hợp hành vi một đối tượng phụ thuộc vào trạng thái của nó và nó phải thay đổi hành vi lúc runtime tùy thuộc vào từng trạng thái. Trong các hệ thống lớn và phức tạp thì áp dụng State Pattern sẽ giúp code của bạn sáng sủa, độc lập về logic, không phải check quá nhiều điều kiện if...else rồi switch...case loằng ngoằng, dễ maintain, dễ mở rộng hơn.
- Có 1 class xử lý các state, trong class này sẽ lưu current state bằng hàm setState và 1 loạt action của các state đó. Các state sẽ có chung các action nên nó sẽ implement 1 interface có sẵn, trong các state override lại các action này

Ví dụ đơn giản:

//Create 1 interface cho các state:

```
interface MobileAlertState
{
    public void alert(AlertStateContext ctx);
}

// Create 2 state cơ bản:
class Vibration implements MobileAlertState
{
    @Override
    public void alert(AlertStateContext ctx)
    {
         System.out.println("vibration...");
    }

}

class Silent implements MobileAlertState
{
    @Override
    public void alert(AlertStateContext ctx)
    {
        System.out.println("silent...");
    }

}

// Create class quản lý các state

class AlertStateContext
{
    private MobileAlertState currentState;

     // Gán current là 1 state bất kỳ
    public AlertStateContext()
    {
        currentState = new Vibration();
    }

    public void setState(MobileAlertState state)
    {
        currentState = state;
    }
```

```
public:
    virtual void registerObserver(IObserver* o) = 0;
    virtual void removeObserver(IObserver* o) = 0;
    virtual void notifyObservers() = 0;
};

//Create Subject cho phép các Observer register

class Database : public ISubject
{
private:
    vector<IObserver*> mObservers;
    string mOperation;
    string mRecord;

public:
    Database() {}

    void registerObserver(IObserver* o) {
        mObservers.push_back(o);
    }

    void removeObserver(IObserver* o) {
        auto observer = find(mObservers.begin(), mObservers.end(), o);
        if (observer != mObservers.end()) {
            mObservers.erase(observer, observer + 1); // remove observer from mObservers
        }
    }

  // Thông báo đến các observer đang register khi có sự thay đổi
    void notifyObservers() {
        for (auto& o : mObservers) {
            o->update(mOperation, mRecord);
        }
    }

  // Các action của Subject
    void editRecord(string operation, string record) {
        mOperation = operation;
        mRecord = record;
        notifyObservers();
    }
};
```

// Tạo class base của các observer

```
class IObserver
{
public:
    virtual void update(string operation, string record) = 0;
};
```

//Tạo các Observer kế thừa từ interface base:

```
class Client : public IObserver
{
public:
    Client() {}

    void update(string operation, string record) {
        cout << "Client: " << operation << " opeation was performed on " << record << endl;
    }
};

class Developer : public IObserver
{
public:
    Developer() {}

    void update(string operation, string record) {
        cout << "Developer: " << operation << " opeation was performed on " << record << endl;
    }
};

class Boss : public IObserver
{
public:
    Boss() {}

    void update(string operation, string record) {
        cout << "Boss: " << operation << " opeation was performed on " << record << endl;
    }
};
```

//Ở hàm main() tạo các đối tượng observer sau đó register vào Subject

```
int main()
{
    Database database;
    Developer dev;
    Client client;
    Boss boss;
    database.registerObserver(&dev);
    database.registerObserver(&client);
    database.registerObserver(&boss);
    database.editRecord("delete", "record1");
    return 0;
}
```

```
Output:
Developer: delete opeation was performed on record1
Client: delete opeation was performed on record1
Boss: delete opeation was performed on record1
```

Memento Pattern

```
    }

    public void setState(MobileAlertState state)
    {
        currentState = state;
    }

    public void alert()
    {
        currentState.alert(this);
    }
}
```

//Ở hàm main chỉ cần gán state và thực hiện các action:

```
class StatePattern
{
    public static void main(String[] args)
    {
        AlertStateContext stateContext = new AlertStateContext();
        stateContext.alert();
        stateContext.alert();
        stateContext.setState(new Silent());
        stateContext.alert();
        stateContext.alert();
        stateContext.alert();
    }
}
```

Output:

```
vibration...
vibration...
silent...
silent...
silent…
```

Vì sao cần phải sử dụng State Pattern, nó được sử dụng trong các trường hợp như thế nào? Lợi ích???

Ví dụ: có 1 đối tượng là điện thoại, điện thoại này có các chế độ như silent, vibration, …
Điện thoại thực hiện 1 số action như cuộc gọi đến, tin nhắn đến, âm báo thức

Bình thường nếu k dùng state pattern, ta sẽ tạo 1 class là điện thoại trong đó các phương thức là cuộc gọi đến, tin nhắn đến, âm báo thức sẽ có các if/else xem đt đang ở trạng thái nào. Nếu làm theo các cơ bản sẽ bị lặp lại code, mỗi một phương thức sẽ phải check xem đth đang ở chế độ nào để thực hiện action đúng nhất

State pattern xử lý đc sự cồng kềnh này và khi cần thêm state thì chỉ cần tạo thêm class cho state đó và thêm vào class chung xử lý các state thay vì thêm vào các phương thức các if else như các thông thường trên=> dễ maintain mở rộng hơn rất nhiều

Startegy Pattern

- In computer programming, the **strategy pattern** (also known as the **policy pattern**) is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use

Advantages:

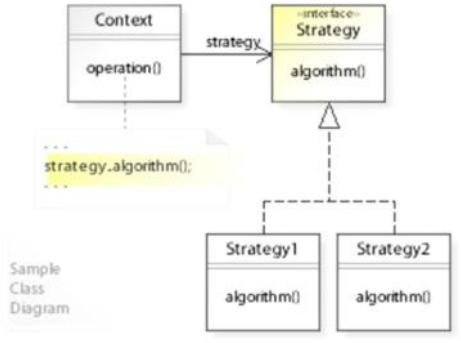
- A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.
- By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
- The application can switch strategies at run-time.
- Strategy enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.
- Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.

Disadvantages

- The application must be aware of all the strategies to select the right one for the right situation.
- Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.

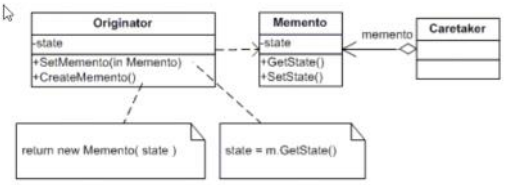
=> có nhiều điểm giống template, tùy vào thuật toán mình truyền vào sẽ xử lý khác nhau, cùng 1 họ là so sánh có các thuật toán các kiểu khác nhau(so sánh 2 số nguyên, so sánh 2 object,…)

Class diagram



Memento Pattern

- Memento pattern là một pattern được sử dụng với mục đích lưu trữ trạng thái của một đối tượng nhất định. Nhờ đó, chúng ta có thể đưa đối tượng trở về trạng thái đã lưu. Việc này đặc biệt hiệu quả cho việc xây dựng các chức năng liên quan đến trạng thái của hệ thông như undo hay redo.



Memento pattern

- Class Originator thể hiện đối tượng ứng dụng cần quản lý như hệ thống, nhân viên hay trò chơi
- Class Memento được dùng để lưu trữ trạng thái của Originator
- Class Caretaker sẽ lưu trữ nhiều đối tượng Memento để hỗ trợ chương trình quản lý và sử dụng chúng. Thông thường, các Memento được lưu trữ bằng list hoặc là stack sẽ hiệu quả hơn.

Tóm tắt theo ý hiểu: Có 1 thằng là caretaker lưu giữ các memento khác nhau gộp lại thành 1 list kiểu stack, Originator là 1 đối tượng duy nhất(ví dụ là 1 nhân vật game), nhân vật này ở các thời điểm có các state khác nhau (tên khác level khác tuổi khác) vì vậy mỗi state sẽ được lưu lại nếu cần undo. Và các state đó được lưu qua thằng Memento, mỗi lần lưu sẽ tạo 1 đối tượng Memento mới và truyền *this chính thằng originator đấy vào, trong class mementor sẽ có 1 biến local _originator để khi cần get ra lúc restore.
Main: tạo 1 thằng caretaker duy nhất, tạo 1 originator duy nhất, mỗi lần set 1 state cho originator thì dùng caretaker-> setMemento(đối tượng truyền vào là originator->createMemetor) => lưu memetor vào caretake.
Lúc cần lấy ra thì dùng originator->restoreToMemento(đối tượng truyền vào là caretaker-> getMemento() để lấy thằng ngoài cùng nhất ra đầu tiên).

// Create class Caretaker

```
#include <iostream>
#include "Caretaker.h"
#include "Memento.h"
#include <list>

=> tạo std::list _listMemento for kiểu class Memento *
private:
  std::list<Memento *> _listMemento;
  Memento *_lastElementInList;

=> khởi tạo _lastElementInList = NULL
Caretaker::Caretaker() : _lastElementInList(NULL) {
}

=> hàm hủy => hủy các object tồn tại
Caretaker::~Caretaker() {
 for (Memento *element : _listMemento) {
  delete element;
 }
 _listMemento.clear();

 if (NULL != _lastElementInList) {
  delete _lastElementInList;
 }
}

=> thêm vào list
void Caretaker::setMemento(Memento *memento) {
 _listMemento.push_back(memento);
}

=> lấy ra từ list từ ngoài vào trong (stack)
Memento *Caretaker::getMemento() {
 _lastElementInList = _listMemento.back();
 _listMemento.pop_back();
 return _lastElementInList;
}
```

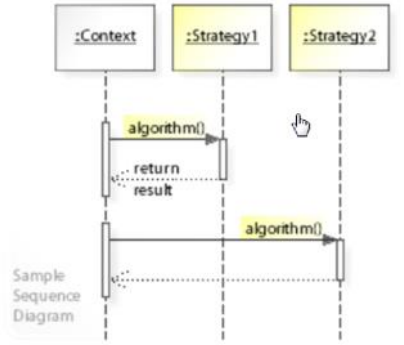
// Create class Mementor: chuyên lưu trữ trạng thái Originator

```
#include <iostream>
#include "Memento.h"
//khởi tạo thì truyền originator vào và lưu ở biến local _... Để sau này get
Memento::Memento(Originator originator) : _originator(originator) {
}

Memento::~Memento() {
}

void Memento::setOriginator(Originator originator) {
 _originator = originator;
}

Originator Memento::getOriginator() {
 return _originator;
}
```

//Create class Originator: đây là các đối tượng cần lưu trữ nó có name, age

```
#include <iostream>
#include "Originator.h"
#include "Memento.h"
Originator::Originator(const std::string &name, const int age) :
 _name(name), _age(age) {
}
```

Sequence diagram:



//Ví dụ với thuật toán so sánh

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

// tạo 1 thằng base của các kiểu so sánh
class IComparator
{
public:
   virtual bool compare(int a, int b) = 0;
   virtual ~IComparator(){}
};

// so sánh bé hơn sẽ có cách xử lý khác với lớn hơn
class LesserCompratoor : public IComparator
{
public:
   bool compare(int a, int b)
   {
      if(a > b)
         return true;
      else
         return false;
   }
};

// so sánh lớn hơn với cách xử lý khác
class GreaterCompratoor : public IComparator
{
public:
   bool compare(int a, int b)
   {
      if(a < b)
         return true;
      else
         return false;
   }
};
```

// thuật toán sắp xếp

```
class SortingAlgo
{

   IComparator * m_pComparator; // thằng base
   void swap(int &x, int &y)
   {
      int tmp = x;
      x = y;
      y = tmp;
   }
public:
   SortingAlgo()
   {
      m_pComparator = new LesserCompratoor(); // mặc định là kiểu so sánh bé hơn
   }
 // hàm sort sẽ truyền tham chiếu vào mảng cần sắp xếp và kiểu sắp xếp(là class con của class base)
 // thuật toán bubbe sort

   void sort(std::vector<int> & arr, IComparator * pComparator = nullptr)
   {
       if(pComparator == nullptr)
         pComparator = m_pComparator;

       bool isSwapped = true;
       int x = 0;
       while (isSwapped)
       {
           isSwapped = false;
           x++;
           for (int i = 0; i < arr.size() - x; i++)
           {
               if (pComparator->compare(arr[i] , arr[i + 1]) )
               {
                   swap(arr[i], arr[i + 1]);
                   isSwapped = true;
               }
           }
       }
```

```
#include <iostream>
#include "Originator.h"
#include "Memento.h"
Originator::Originator(const std::string &name, const int age) :
  _name(name), _age(age) {
}

Originator::~Originator() {
}
void Originator::setName(const std::string &name) {
  _name = name;
}
const std::string Originator::getName() const{
  return _name;
}

void Originator::setAge(const int age) {
  _age = age;
}

const int Originator::getAge() const{
  return _age;
}

//Create Memento và truyền đối tượng Originator đấy vào

Memento *Originator::createMemento() {
  return new Memento(*this);
}

//lấy đối tượng Originator ra từ mementor
void Originator::restoreToMemento(Memento *memento) {
    *this = memento->getOriginator();
}
```

// Hàm Main:

```
#include <memory>
#include <iostream>
#include "Originator.h"
#include "Caretaker.h"

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);


    // A
   => tạo đối tượng Originator có trạng thái name = A, age = 1 và Caretaker()
    Originator *originator = new Originator("A", 1);
    Caretaker *caretaker = new Caretaker();

    std::cout << "Name = " << originator->getName() << std::endl;
    std::cout << "Age = " << originator->getAge() << std::endl;
    => originator->createMemento() => khởi tạo đối tượng Memento để lưu trữ đối tượng
Originator  này
    => cho memento này vào list trong đối tượng caretaker
    caretaker->setMemento(originator->createMemento());

    // B
    originator->setName("B");
    originator->setAge(2);
    std::cout << "Name = " << originator->getName() << std::endl;
    std::cout << "Age = " << originator->getAge() << std::endl;
    caretaker->setMemento(originator->createMemento());

    // Go back
    originator->restoreToMemento(caretaker->getMemento());
    std::cout << "Name = " << originator->getName() << std::endl;
    std::cout << "Age = " << originator->getAge() << std::endl;
    originator->restoreToMemento(caretaker->getMemento());
    std::cout << "Name = " << originator->getName() << std::endl;
    std::cout << "Age = " << originator->getAge() << std::endl;

    delete caretaker;
    delete originator;

}
```

```
                {
                    swap(arr[i], arr[i + 1]);
                    isSwapped = true;
                }
            }
        }
    }

};
```

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    std::vector<int> arr = {1,5,2,4,3};
    SortingAlgo obj;
    IComparator * pComp = new LesserCompratаor();
    obj.sort(arr, pComp);
    for (int var = 0; var < 5; ++var) {
        std::cout<<arr[var]<<" ";
    }
    std::cout<<std::endl;
    delete pComp;
    pComp = nullptr;

    pComp = new GreaterCompratаor();
    obj.sort(arr, pComp);
    for (int var = 0; var < 5; ++var) {
      std::cout<<arr[var]<<" ";
    }
    std::cout<<std::endl;
    delete pComp;
    pComp = nullptr;

    delete pComp;
    pComp = nullptr;

    obj.sort(arr);
    for (int var = 0; var < 5; ++var) {
        std::cout<<arr[var]<<" ";
    }
    std::cout<<std::endl;
    delete pComp;
    pComp = nullptr;

    return a.exec();
}
```