Composite,Observer,Undo/Redo 的C++实现

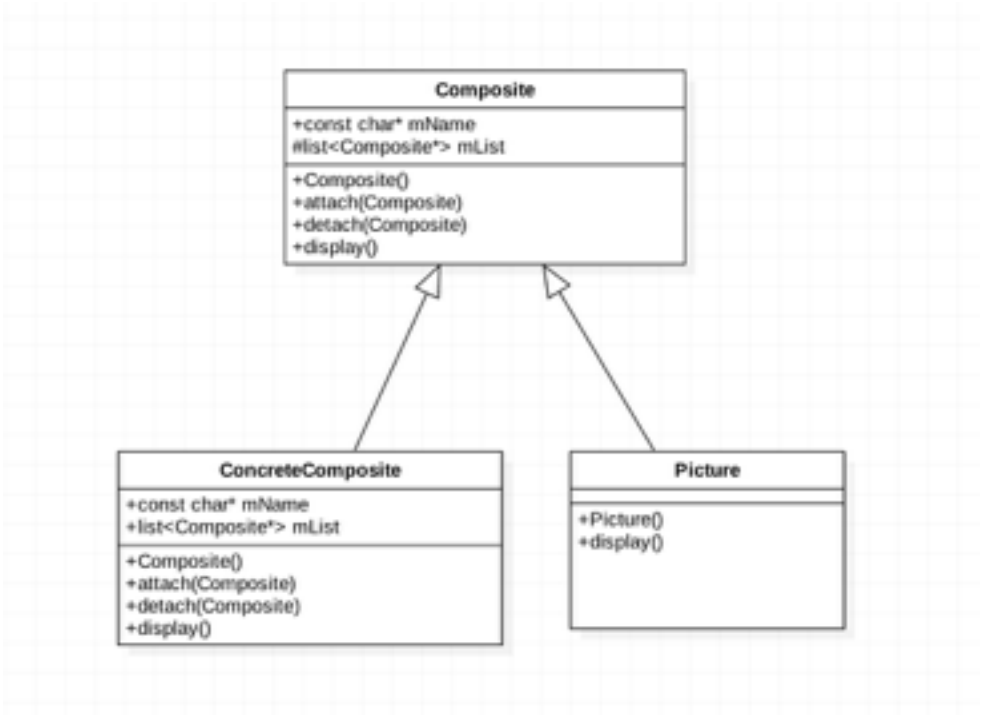
——1452794 孙晓彤

1. Composite

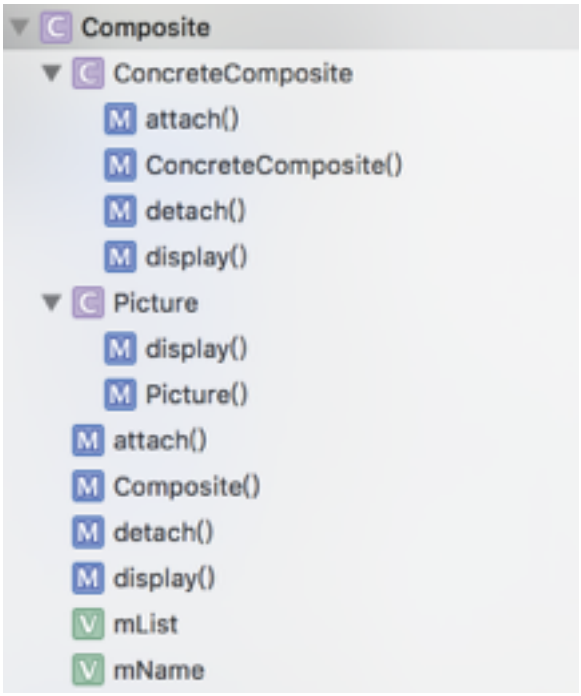
1.0. 概述

组合模式将对象组合成树形结构以表示“部分-整体”的层次结构。Composite使得用户对单个对象和组合对象的使用具有一致性。

1.1. 类图



1.2. Hierarchical图



1.3. 代码及运行结果

main.cpp

```
#include "Composite.hpp"
#include <iostream>

using namespace std;

int main()
{
    //创建文件夹1并添加文件夹1_1和1_2到文件夹1下
    ConcreteComposite folder1("2016照片集");
    ConcreteComposite folder1_1("2016.5");
    ConcreteComposite folder1_2("2016.7");
    folder1.attach(&folder1_1);
    folder1.attach(&folder1_2);

   //创建文件夹1_3并添加文件1_3_1和1_3_2到文件夹1_3下
    ConcreteComposite folder1_3("2016.11");
    Picture file1_3_1("2016.11.1");
    Picture file1_3_2("2016.11.2");
    folder1_3.attach(&file1_3_1);
    folder1_3.attach(&file1_3_2);

    //将文件夹1_3添加到文件夹1下
    folder1.attach(&folder1_3);

    string str("+ ");
    folder1.display(str);
    cout<<endl;
    folder1_3.detach(&file1_3_1);
    folder1.display(str);


    cout<<endl;
    folder1.detach(&folder1_3);
    folder1.display(str);
    return 0;
}
```

Composite.cpp

```
#include "Composite.hpp"
#include <iostream>
using namespace std;

void ConcreteComposite::attach(Composite* cmp)
{
    if(nullptr != cmp)
    {
        mList.push_back(cmp);
    }
}

void ConcreteComposite::detach(Composite* cmp)
{
```

```
    if(nullptr != cmp)
    {
        mList.remove(cmp);
    }
    cout << "File " << cmp->mName << " is deleted" << endl;
}

void ConcreteComposite::display(string str)
{
    list<Composite* >::iterator beg = mList.begin();
    cout<<str<<mName<<endl;
    str = str + "+ ";
    for ( ; beg != mList.end(); beg++)
    {
        (*beg)->display(str);
    }
}

void Picture::display(string str)
{
    cout<<str<<mName<<endl;
}
```

Composite.hpp

```
#ifndef Composite_hpp
#define Composite_hpp


#include <list>
#include <string>
#include <iostream>
using namespace std;

class Composite
{
public:
    Composite(const char* name):mName(name){}
    virtual void attach(Composite* file_name){}
    virtual void detach(Composite* file_name){}
    virtual void display(string str){}
    const char* mName;

protected:
    list<Composite* > mList;
};

class ConcreteComposite:public Composite{
public:
    ConcreteComposite(const char* name):Composite(name){}
    virtual void attach(Composite* file_name);
    virtual void detach(Composite* file_name);
    virtual void display(string str);
};

class Picture:public Composite{
public:
```

```
    Picture(const char* name):Composite(name){}
    virtual void display(string str);
};


#endif
```

运行结果：

```
+ 2016照片集
+ + 2016.5
+ + 2016.7
+ + 2016.11
+ + + 2016.11.1
+ + + 2016.11.2

File 2016.11.1 is deleted
+ 2016照片集
+ + 2016.5
+ + 2016.7
+ + 2016.11
+ + + 2016.11.2

File 2016.11 is deleted
+ 2016照片集
+ + 2016.5
+ + 2016.7
Program ended with exit code: 0
```

1.4. 设计说明

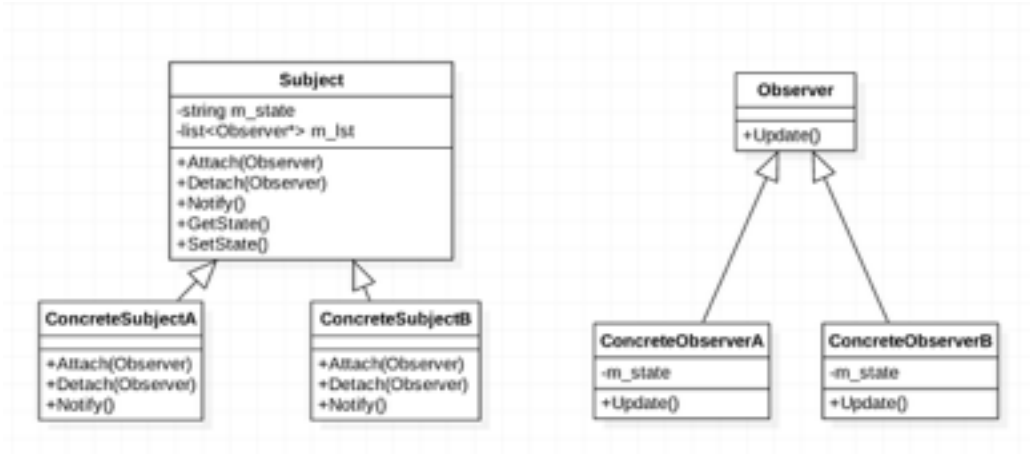
加号用来表示文件的层级结构，代码中的Picture类相当于是叶子节点。

2.Observer

2.0. 概述

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己

2.1. 类图



2.2. Hierarchical图

2.3. 代码及运行结果

main.cpp

```
#include "Observer.hpp"
#include <iostream>

using namespace std;

int main()
{
    Observer* p1 = new ConcreteObserverA();
    Observer* p2 = new ConcreteObserverB();
    Observer* p3 = new ConcreteObserverA();

    Subject* pSubject = new ConcreteSubjectA();
    pSubject->Attach(p1);
    pSubject->Attach(p2);
    pSubject->Attach(p3);
```

```
    pSubject->SetState("old");
    cout << "SET STATE OLD" << endl;

    pSubject->Notify();

    cout << "-------------------------------------" << endl;
    pSubject->SetState("new");
    cout << "SET STATE NEW" << endl;

    pSubject->Detach(p3);
    pSubject->Notify();

    return 0;
}
```

observer.cpp

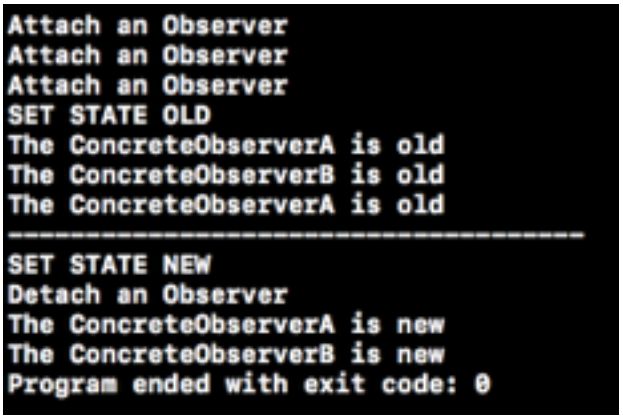
```
#include "observer.hpp"
#include <iostream>
#include <algorithm>

using namespace std;

Observer::Observer()
{}

Observer::~Observer()
{}

ConcreteObserverA::ConcreteObserverA()
{}

ConcreteObserverA::~ConcreteObserverA()
{}

void ConcreteObserverA::Update(Subject* pSubject)
{
    this->m_state = pSubject->GetState();
    cout << "The ConcreteObserverA is " << m_state << std::endl;
}

ConcreteObserverB::ConcreteObserverB()
{}

ConcreteObserverB::~ConcreteObserverB()
{}

void ConcreteObserverB::Update(Subject* pSubject)
{
    this->m_state = pSubject->GetState();
    cout << "The ConcreteObserverB is " << m_state << std::endl;
}

Subject::Subject()
{}

Subject::~Subject()
```

```
{}

void Subject::Attach(Observer* pObserver)
{
    this->m_lst.push_back(pObserver);
    cout << "Attach an Observer\n";
}

void Subject::Detach(Observer* pObserver)
{
    list<Observer*>::iterator iter;
    iter = find(m_lst.begin(),m_lst.end(),pObserver);
    if(iter != m_lst.end())
    {
        m_lst.erase(iter);
    }
    cout << "Detach an Observer\n";
}

void Subject::Notify()
{
    list<Observer*>::iterator iter = this->m_lst.begin();
    for(;iter != m_lst.end();iter++)
    {
        (*iter)->Update(this);
    }
}

string Subject::GetState()
{
    return this->m_state;
}

void Subject::SetState(string state)
{
    this->m_state = state;
}

ConcreteSubjectA::ConcreteSubjectA()
{}

ConcreteSubjectA::~ConcreteSubjectA()
{}

ConcreteSubjectB::ConcreteSubjectB()
{}

ConcreteSubjectB::~ConcreteSubjectB()
{}

observer.hpp
#ifndef _OBSERVER_H_
#define _OBSERVER_H_

#include <string>
#include <list>
```

```
using namespace std;

class Subject;

class Observer
{
public:
    Observer();
    ~Observer();
    virtual void Update(Subject*)=0;

private:
};

class ConcreteObserverA : public Observer
{
public:
    ConcreteObserverA();
    ~ConcreteObserverA();
    virtual void Update(Subject*);

private:
    string m_state;
};

class ConcreteObserverB : public Observer
{
public:
    ConcreteObserverB();
    ~ConcreteObserverB();
    virtual void Update(Subject*);

private:
    string m_state;
};

class Subject
{
public:
     Subject();
    ~Subject();
    virtual void Notify();
    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual string GetState();
    virtual void SetState(string state);

private:
    string m_state;
    list<Observer*> m_lst;
};

class ConcreteSubjectA : public Subject
{
public:
    ConcreteSubjectA();
```

```
    ~ConcreteSubjectA();
protected:
private:
};

class ConcreteSubjectB : public Subject
{
public:
    ConcreteSubjectB();
    ~ConcreteSubjectB();
protected:
private:
};

#endif
```

运行结果：



2.4. 设计说明

在main函数中输入状态模拟状态变化，初始添加三个观察者，将状态置为“old”，改变subjectA的状态后，observer的状态也会随之改变。

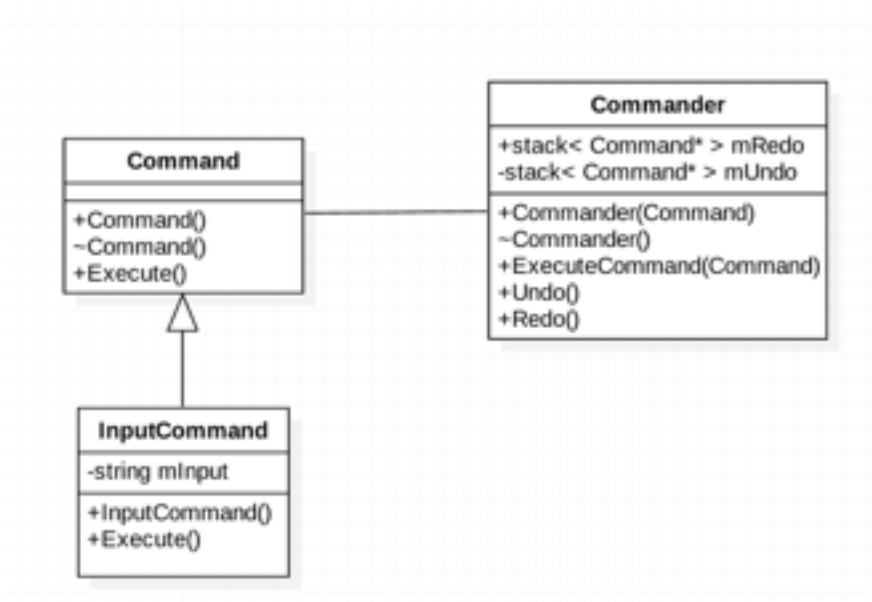
目前设计存在两个问题，一个是notify()函数不应该是在main函数里显式调用，而是应该隐式执行，另一个问题在于没有设计一个阈值，即要加上一个条件，来确定何时observer状态随subject改变，也就是要增加一个判断条件（老师举例，某同学做的热水达到某个温度开始通知观察者）。

3.Undo/Redo

3.0. 概述

命令模式把一个请求或者操作封装到一个对象中，把发出命令的责任和执行命令的责任分割开，委派给不同的对象，可降低行为请求者与行为实现者之间耦合度。从使用角度来看就是请求者把接口实现类作为参数传给使用者，使用者直接调用这个接口的方法，而不用关心具体执行的那个命令。

3.1. 类图



3.2. Hierarchical图

3.3. 代码及运行结果

main.cpp

```
#include "Command.hpp"


int main()
{

    Commander *p = new Commander( new InputCommand( "无人留言状态" ) );

    p->ExecuteCommand( new InputCommand( "求七点开热水器" ) );
    p->ExecuteCommand( new InputCommand( "数据库课本能帮我带一下么" ) );
```

```
    p->ExecuteCommand( new InputCommand( "啊中午去吃火锅吧！" ) );

    p->Undo();
    p->Undo();
    p->ExecuteCommand( new InputCommand( "今晚我不回寝室了" ) );
    p->Undo();
    p->Undo();

    //执行失败,undo 到最原始情况
    p->Undo();

    p->Redo();
    p->Redo();
    p->Redo();

    p->ExecuteCommand( new InputCommand( "这周二的课不用上了" ) );

    p->Undo();
    p->Redo();
    p->Redo();

    //执行失败,redo 到最新情况
    p->Redo();
    delete p;
    return 0;
}

Command.hpp
#ifndef _COMMAND_HPP
#define _COMMAND_HPP

#include <iostream>
#include <stack>
#include <string>

class Command
{
public:
    Command(){}
    virtual ~Command(){}
    virtual void Execute() = 0;
};


class InputCommand : public Command
{
public:
    InputCommand( const std::string input )
    {
        mInput = input;
    }
    ~InputCommand()
    {}
    void Execute()
```

```
    {
        std::cout << mInput << std::endl;
    }

private:
    std::string mInput;
};

class Commander
{
public:
    Commander( Command *pCmd )
    {
        mUndo.push( pCmd );
    }
    ~Commander()
    {
        while( false == mUndo.empty() )
        {
            mUndo.pop();
        }
        while( false == mRedo.empty() )
        {
            mRedo.pop();
        }
    }

    void ExecuteCommand( Command *pCmd )
    {
        pCmd->Execute();
        mUndo.push( pCmd );
    }

    void Undo()
    {
        std::cout << "-------上一条-------" << std::endl;
        if( mUndo.size() < 2 )
        {
            //无法读取上一条
            std::cout << "当前已是最早留言" << std::endl;
            return;
        }

        auto pCmd = mUndo.top();
        mRedo.push( pCmd );
        mUndo.pop();
        //pCmd指向最新栈顶元素
        pCmd = mUndo.top();
        pCmd->Execute();
    }

    void Redo()
    {
        std::cout << "-------下一条-------" << std::endl;
```

```
        if( mRedo.empty() )
        {
            //无法读取下一条
            std::cout << "当前已是最新留言" << std::endl;
            return;
        }

        auto pCmd = mRedo.top();
        pCmd->Execute();
        mRedo.pop();
        mUndo.push( pCmd );
    }

private:
    std::stack< Command* > mUndo;
    std::stack< Command* > mRedo;
};

#endif
```

运行结果：

3.4. 设计说明

运用command模式，实现了解耦，模拟寝室留言簿，可以查看当前留言，上一条留言，下一条留言，并且不断上翻或下翻。