Proxy

Cas d'utilisation d'un DP Proxy pour limiter le « coût » de gestion de certains objets.

Dans le programme suivant la construction d'un objet `CFileImage` est coûteuse puisqu'elle oblige à lire le fichier.
Modifier le programme, de façon à ce que la construction coûteuse d'un objet `CFileImage` ne se fasse que lorsque le client appelle une des méthodes `IsValid()` ou `Print()`.

```
#include <memory.h>   // for memset
#include <iostream.h> // for cout
#define BUF_SIZE 10000

struct CFileImage
{
        CFileImage( const char * fname );
        bool IsValid(){ return m_bValid; }
        void Print();
private:
        bool   m_bValid;
        char   m_buffer[BUF_SIZE];
        size_t m_buffersize;
};

CFileImage::CFileImage( const char * fname ):
        m_bValid( false ), m_buffersize( 0 )
{
        memset( m_buffer , 0 , BUF_SIZE );
        FILE * fhandle = fopen( fname , "r" );
        if( fhandle )
        {
                m_bValid = true;
                m_buffersize = fread( m_buffer, sizeof( char ), BUF_SIZE, fhandle );
                fclose( fhandle );
        }
}

void CFileImage::Print()
{
        if( m_bValid )
                cout << m_buffer;
        cout.flush();
}

void main()
{
        CFileImage * pf = new CFileImage("toto.txt");
        pf->Print();
        delete pf;
        // attendre un caractère puis entrée
        char wait;
        cin >> wait;
}
```

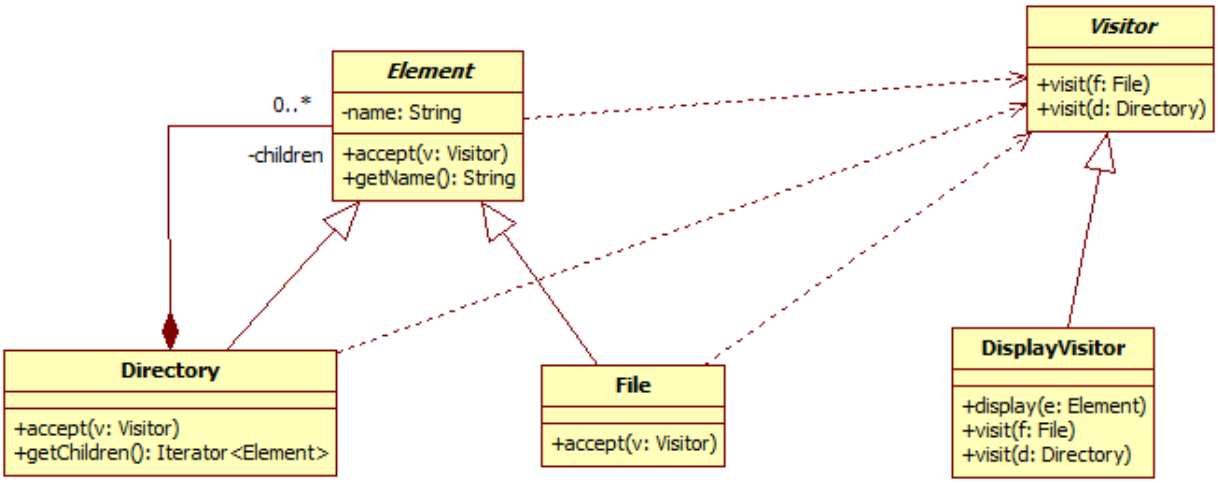
Solution proxy

```
#include <memory.h>   // for memset
#include <iostream.h> // for cout
#define BUF_SIZE 10000
struct IFileImage
{
        virtual bool IsValid()=0;
        virtual void Print()=0;
};
struct CFileImage : public IFileImage
{
        CFileImage( const char * fname );
        bool IsValid(){ return m_bValid; }
        void Print();
private:
        bool   m_bValid;
        char   m_buffer[BUF_SIZE];
        size_t m_buffersize;
};
CFileImage::CFileImage( const char * fname ):
        m_bValid( false ), m_buffersize( 0 )
{
        memset( m_buffer , 0 , BUF_SIZE );
        FILE * fhandle = fopen( fname , "r" );
        if( fhandle )
        {
                m_bValid = true;
                m_buffersize = fread( m_buffer , sizeof( char ) , BUF_SIZE , fhandle );
                fclose( fhandle );
        }
}
void CFileImage::Print()
{
        if( m_bValid )
                cout << m_buffer;
        cout.flush();
}
struct CFileImageProxy : public IFileImage
{
        CFileImageProxy( const char * fname ):
                m_fname( fname ), m_pRealFileImage( NULL ) {}
        bool IsValid();
        void Print();
        ~CFileImageProxy();
private:
        const char * m_fname;
        CFileImage * m_pRealFileImage;
        void LoadRealFileImage();
};
bool CFileImageProxy::IsValid()
{
        LoadRealFileImage();
        return m_pRealFileImage->IsValid();
}
void CFileImageProxy::Print()
{
        LoadRealFileImage();
        m_pRealFileImage->Print();
}
CFileImageProxy::~CFileImageProxy()
{
        if( m_pRealFileImage )   delete m_pRealFileImage;
}
void CFileImageProxy::LoadRealFileImage()
{
        if( ! m_pRealFileImage )  m_pRealFileImage = new CFileImage( m_fname );
}
void main()
{
        IFileImage * pf = new CFileImageProxy("toto.txt");
        pf->Print();
        delete pf ;
        // attendre un caractère puis entrée
        char wait;
        cin >> wait;
}
```

Visiteur

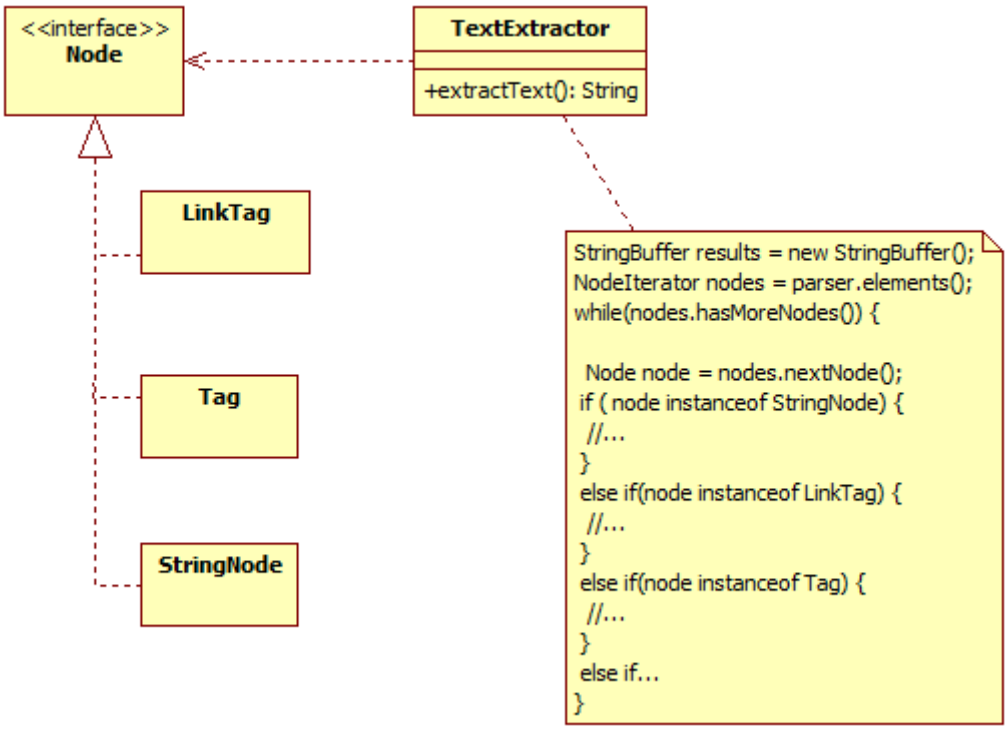
Mettre en place un système de fichier. Sur ces objets «métier», proposer une solution pour afficher le contenu d'un dossier. Plusieurs solutions peuvent être proposées.

Solution basique visiteur

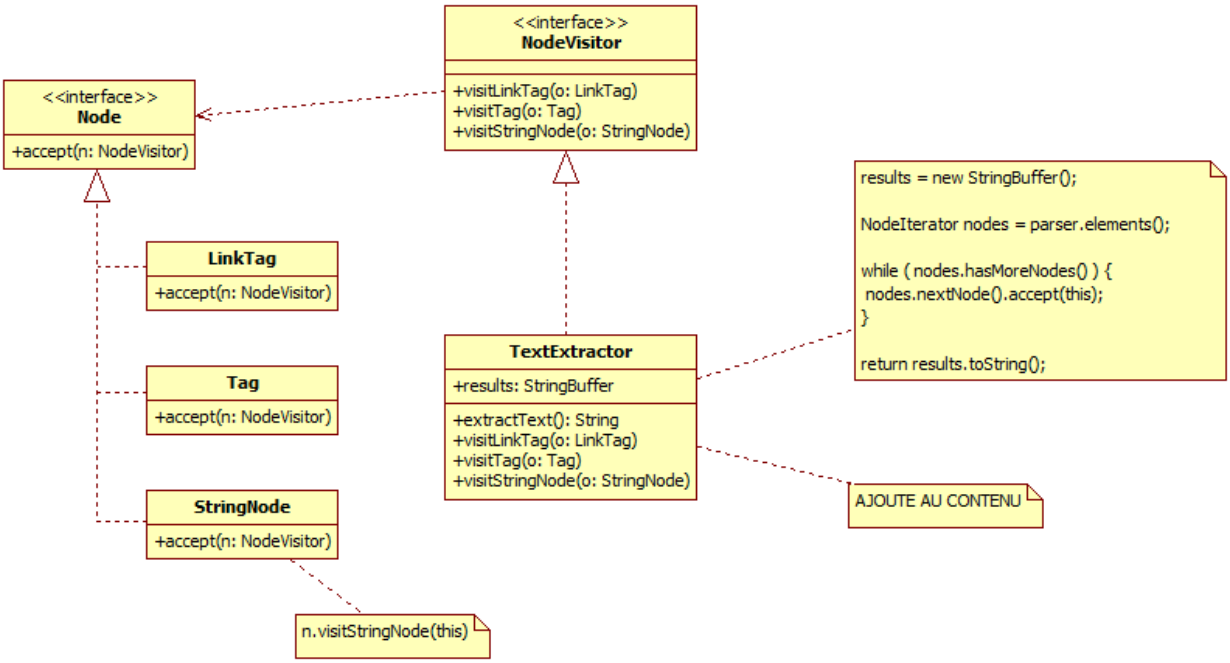


Itérateur – Visiteur

Utilisation d'un DP Visiteur. Cas de refactoring.

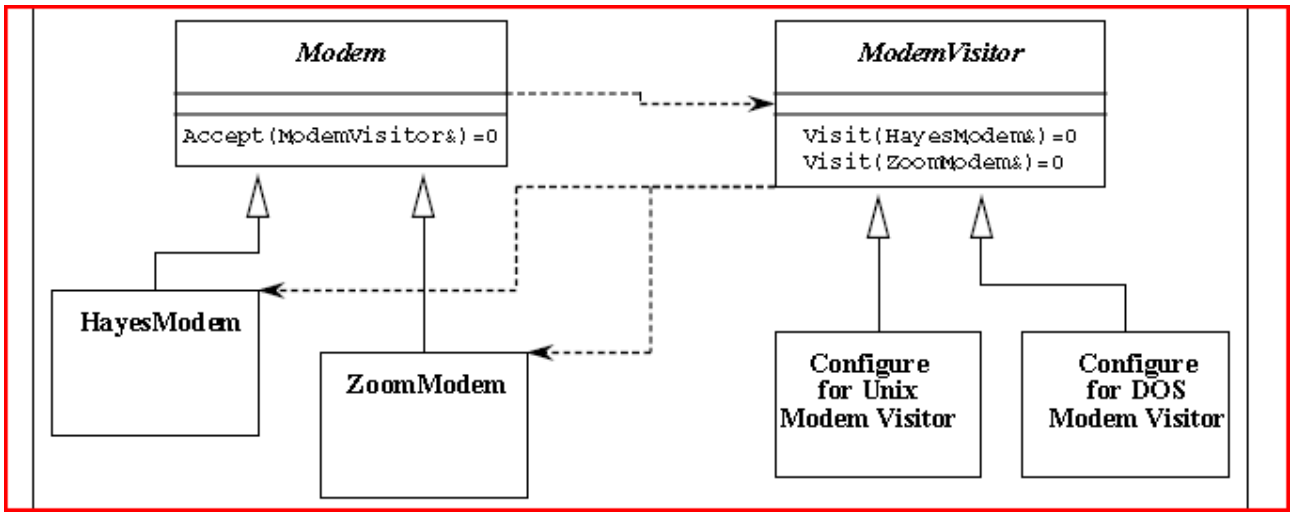


Solutions Iterator et accumulation

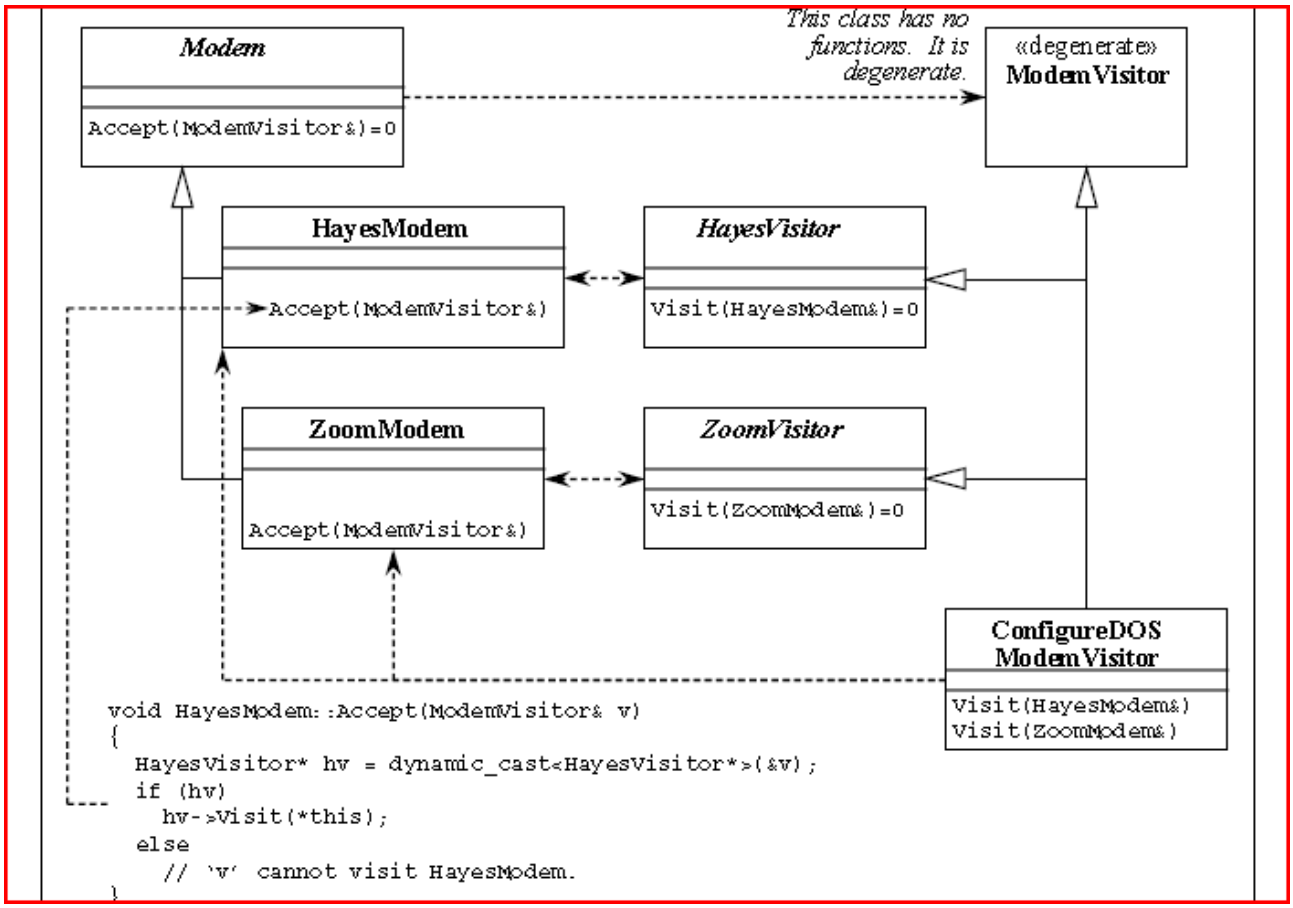


Visiteur … acyclique ?

Reprendre le DP Visiteur. Que penser des dépendances entre les classes ? Comment résoudre le problème ?

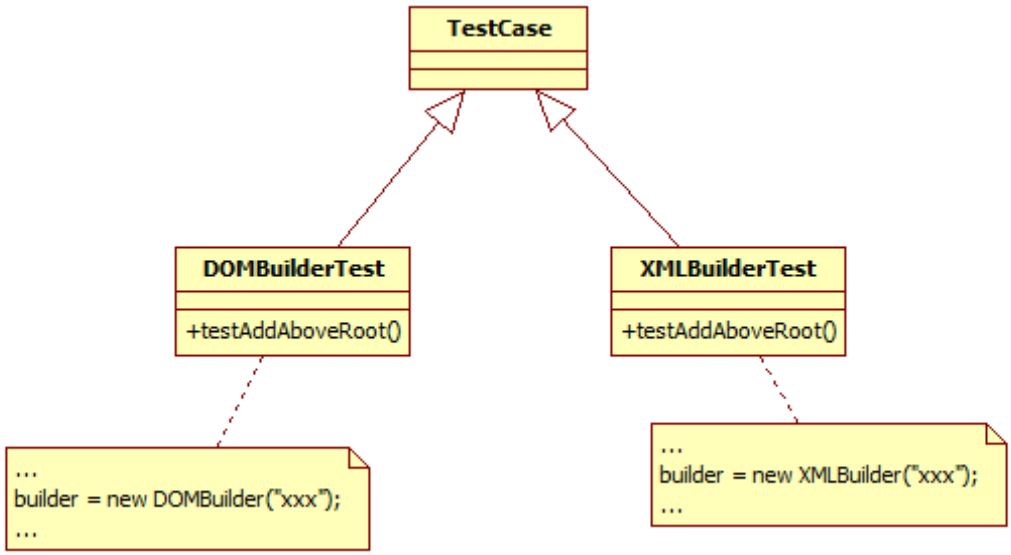


Solutions visiteur acyclique

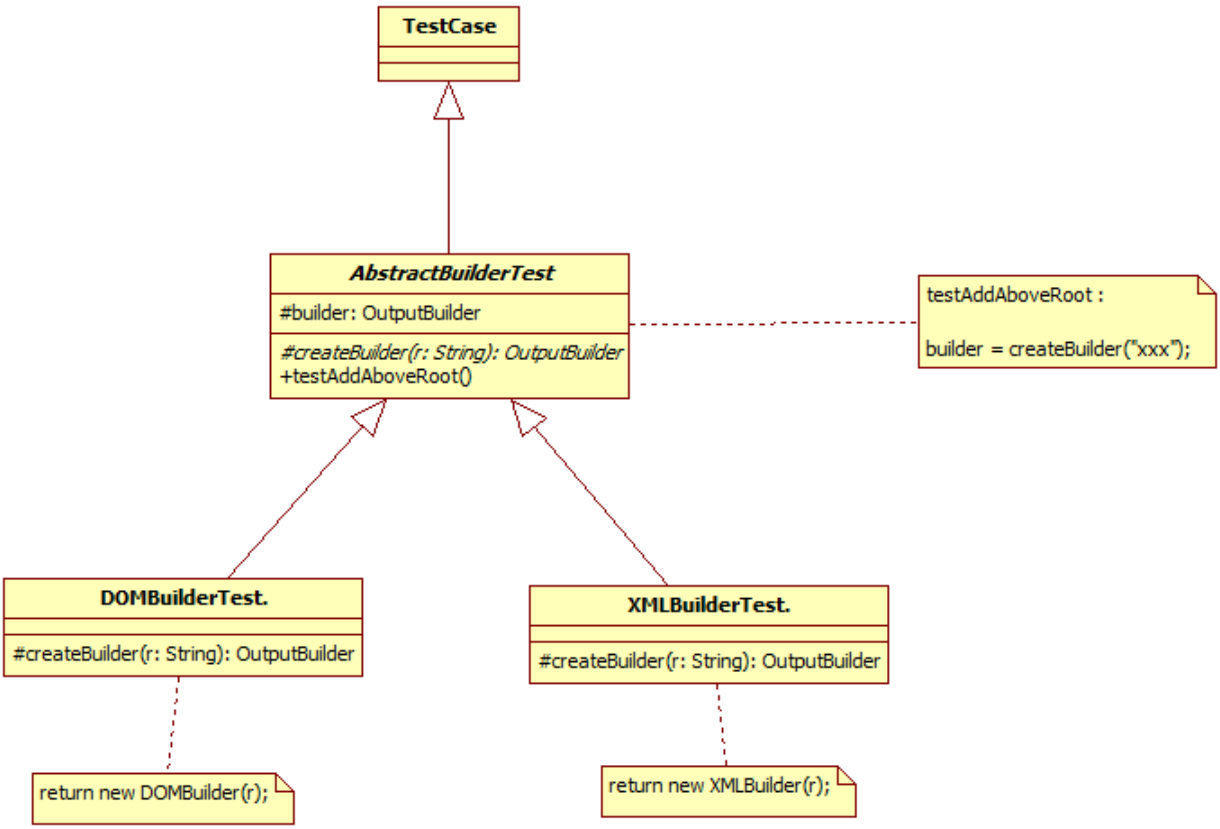


Methode Fabrique

Cas de refactoring : ...

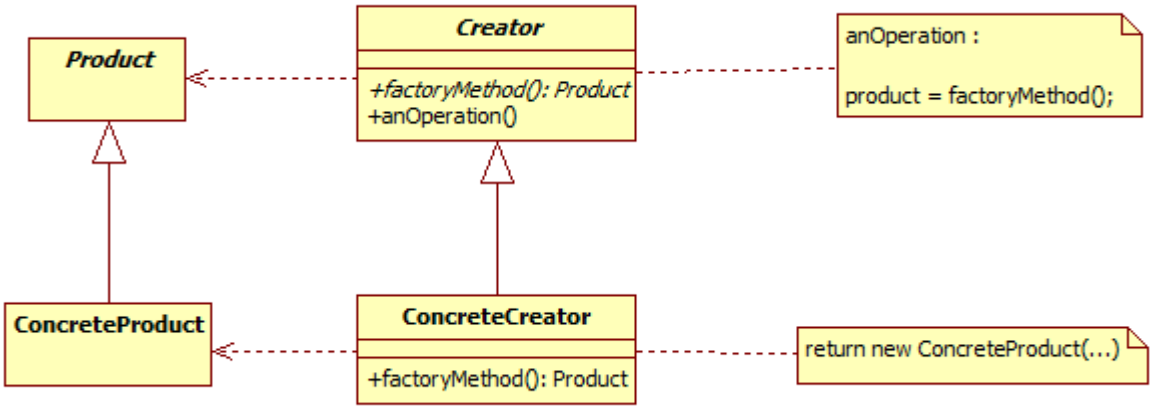


Solution Méthode fabrique

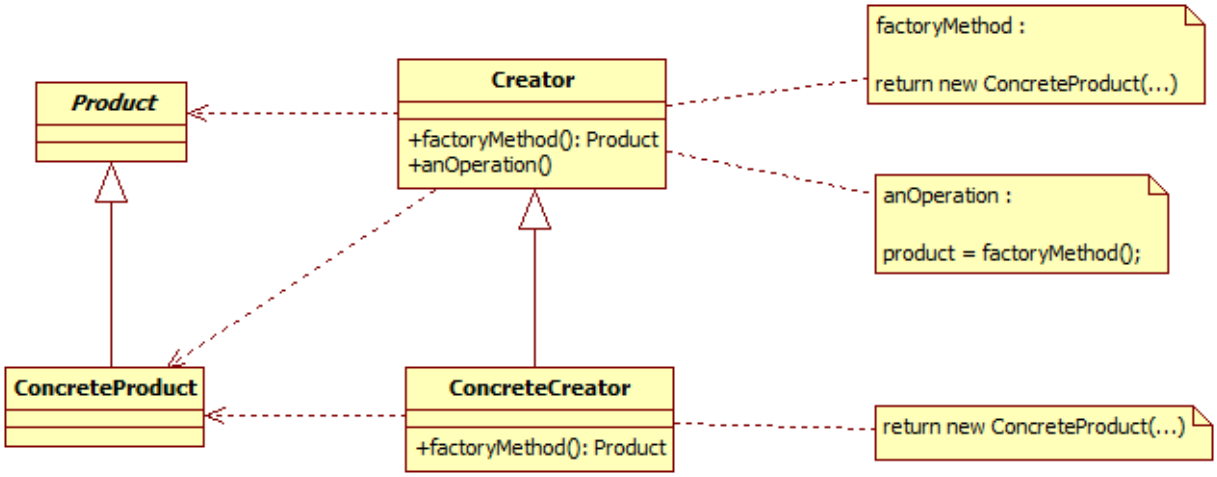


Un pattern peut être implémenté de différentes façons …

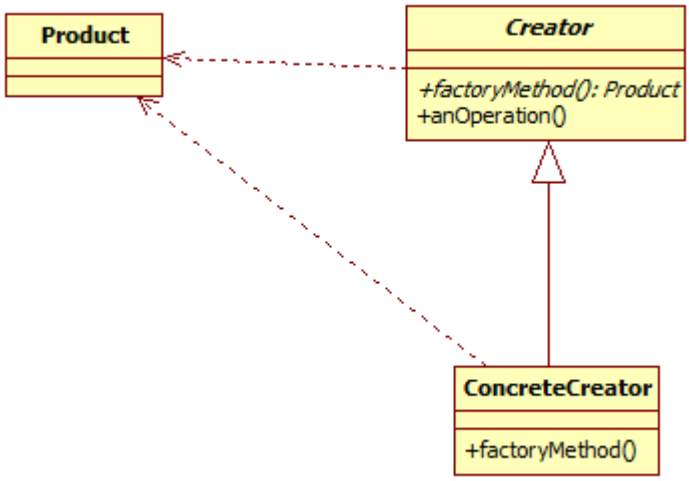
Forme traditionnelle



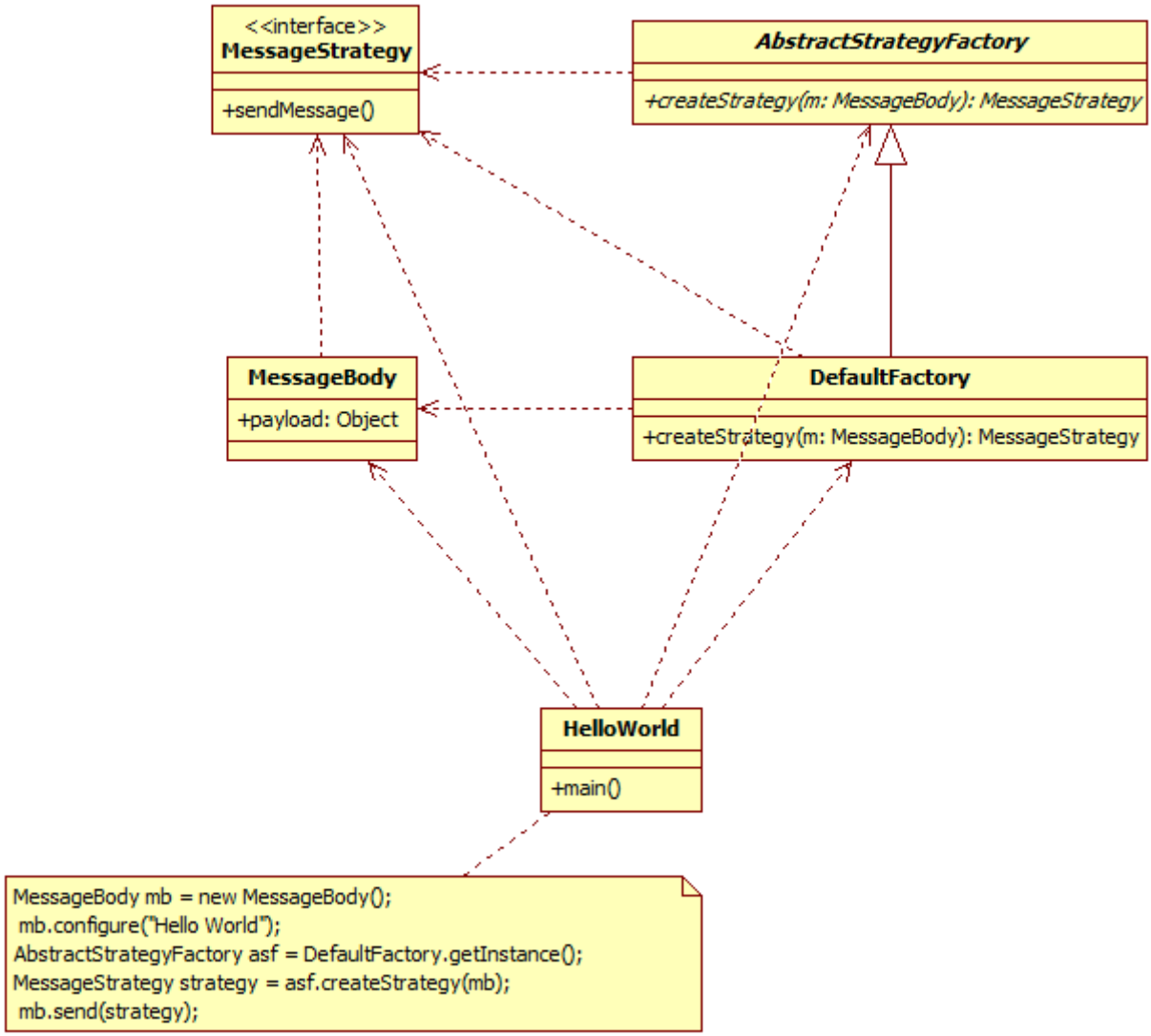
Avec implémentation par défaut



Sans sous-classe de produit



Etre « patterns happy » !



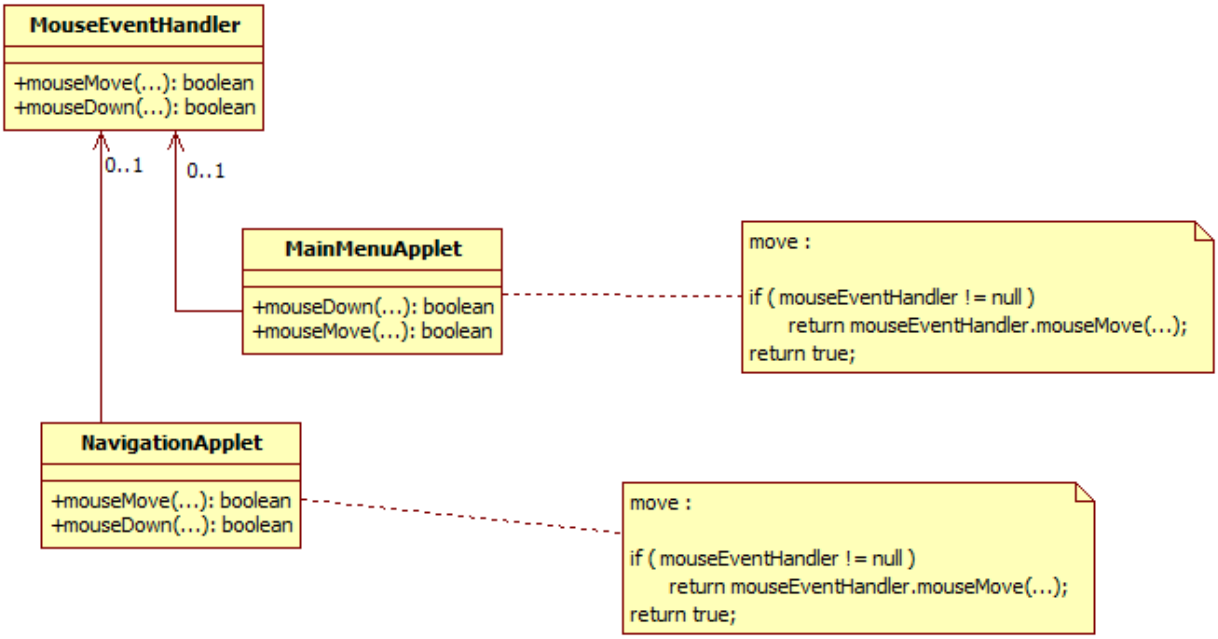
Code :

```
interface MessageStrategy {
        public void sendMessage();
}


abstract class AbstractStrategyFactory {
        public abstract MessageStrategy createStrategy(MessageBody mb);
}

class MessageBody {
        Object payload;
        public Object getPayload() {
                return payload;
```

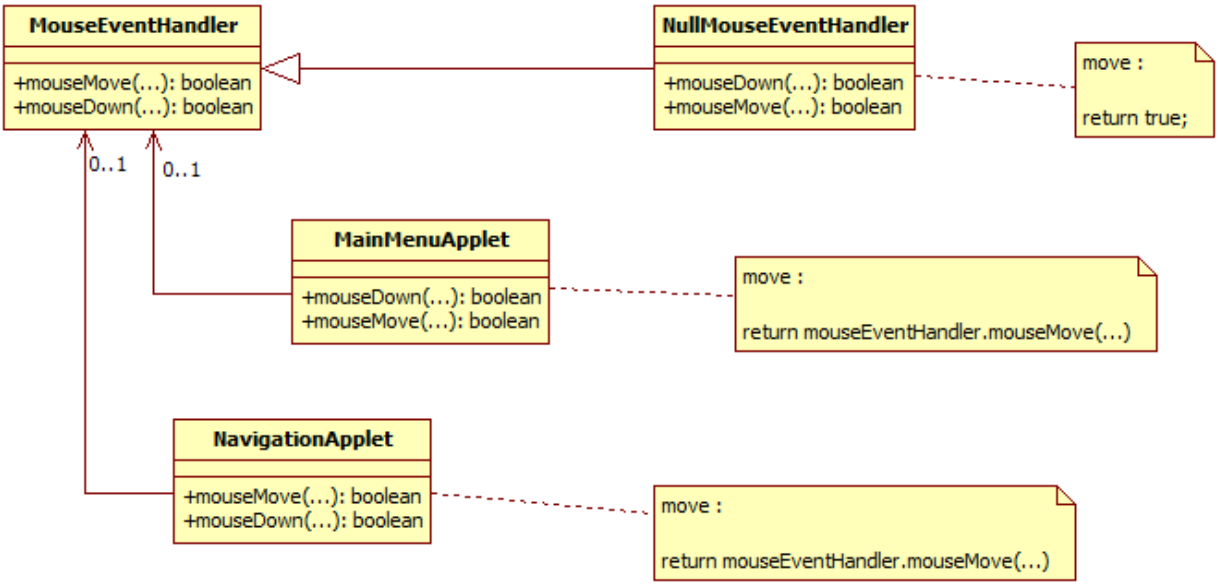
```
	}
	public void configure(Object obj) {
		payload = obj;
	}
	public void send(MessageStrategy ms) {
		ms.sendMessage();
	}
}

class DefaultFactory extends AbstractStrategyFactory {
	private DefaultFactory () {
	}
	static DefaultFactory instance;

	public static AbstractStrategyFactory getInstance() {
		if(instance == null) {
			instance = new DefaultFactory();
		}
		return instance;
	}

	public MessageStrategy createStrategy(final MessageBody mb) {
		return new MessageStrategy () {
			MessageBody body = mb;
			public void sendMessage() {
				Object obj = body.getPayload();
				System.out.println(obj);
			}
		}
	}
}

public class HelloWorld  {
	public static void main(String[] args) {
		MessageBody mb = new MessageBody();
		mb.configure("Hello World");
		AbstractStrategyFactory asf = DefaultFactory.getInstance();
		MessageStrategy strategy = asf.createStrategy(mb);
		mb.send(strategy);
	}
}
```

Null Object

Cas de refactoring

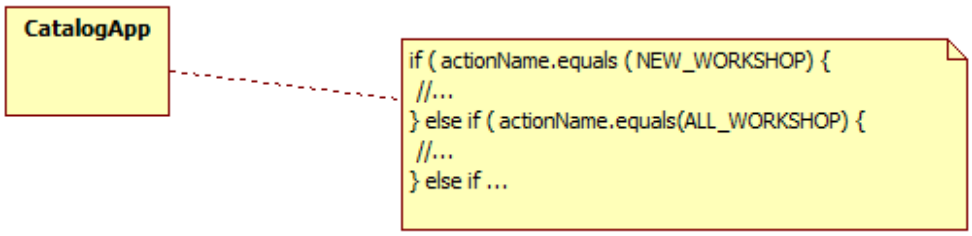


Solution null object

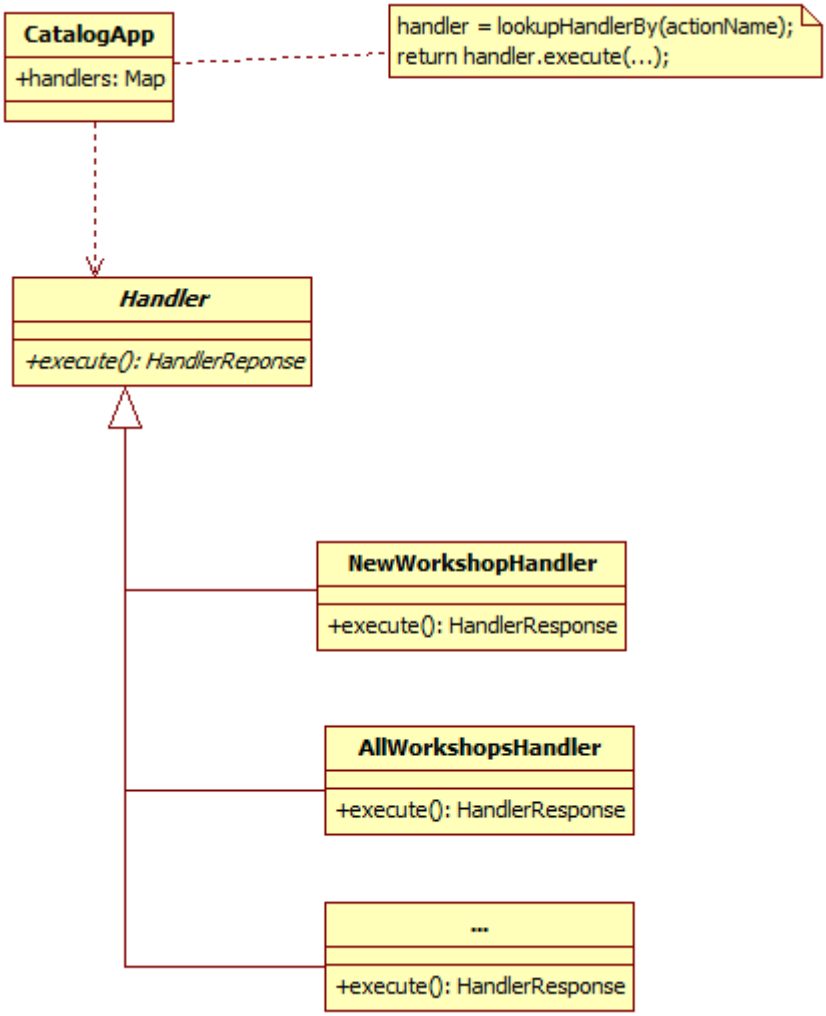


Commande

Logique conditionnelle pour déclencher des actions...

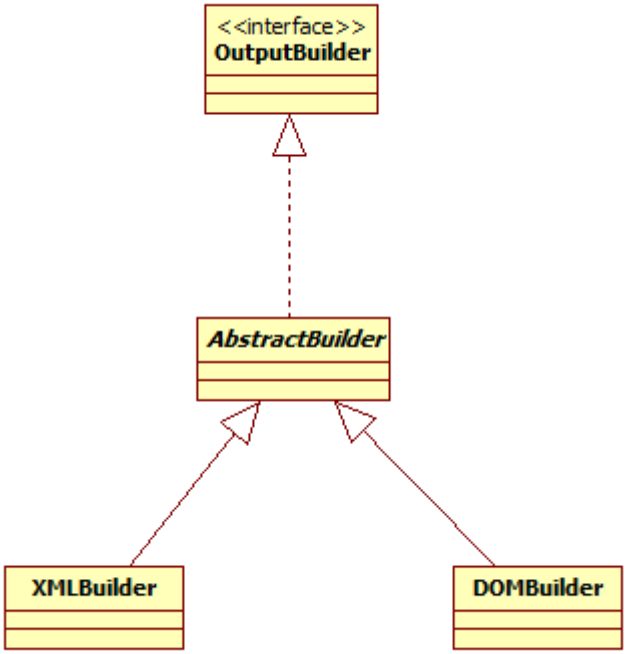


Solution commande



Adaptateur

Exemple de code qui génére du XML



DOMBuilder :

public class DOMBuilder extends AbstractBuilder ...

```
private Document document;
private Element root;
private Element parent;
private Element current;

public void addAttribute(String name, String value) {
 current.setAttribute(name, value);
}

public void addBelow(String child) {
Element childNode = document.createElement(child);
current.appendChild(childnode);
parent = current ;
current = childNode;
history.push(current);
}

public void addBeside(String sibling) {
if(current == root)
        throw new RuntimeException(CANNOT_ADD_BESIDE_ROOT);
Element siblingNode = deocument.createElement(sibling);
parent.appendChild(siblingNode);
```

```
current = siblingNode;
history.pop();
history.push(current);
}

public void addValue(String value) {
current.appendChild(document.createTextNode(value));
}
```

XMLBuilder :

```
public class XMLBuilder extends AbstractBuilder …

    private TagNode rootNode;
    private TagNode currentNode;

    public void addChild(String childTagName) {
        addTo(CurrentNode,childTagName);
    }

    public void addSibling(String siblingTagName) {
        addTo(currentNode.getParent(),siblingTagName);
    }

    public void addTo(TagNode parentNode, String tagName) {
        currentNode = new TagNode(tagName);
        parentNode.add(currentNode);
    }

    public void addAttribute(String name, String value) {
        currentNode.addAttribute(name, value);
    }

    public void addValue(String value) {
        currentNode.addValue(value);
    }
```

Solutions Adaptateur

Dans le code, il apparaît clairement que ces deux classes sont très similaires. La principale différence vient du fait qu'elles manipulent soit Element, soit TagNode. Le but de ce refactoring est des créer une interface commune pour TagNode et Element, de façon à ce que la duplication du code puisse être éliminée.

L'interface commune choisie est celle de TagNode. Préférence arbitraire.
TagNode devient une classe qui implémente l'interface commune.
De l'autre côté, ElementAdapter est une classe qui sera utilisée par DOMBuilder. Il faut reprendre le code de cette dernière classe.

Le code de DOMBuilder :

```
public class DOMBuilder extends AbstractBuilder ...

        public void addAttribute(String name, String value) {
         addAttribute(currentNode, name, value);
        }

        public void addAttribute(ElementAdapter current, String name, String value) {
         currentNode.getElement().setAttribute(name, value);
        }
    //...
```

puis :

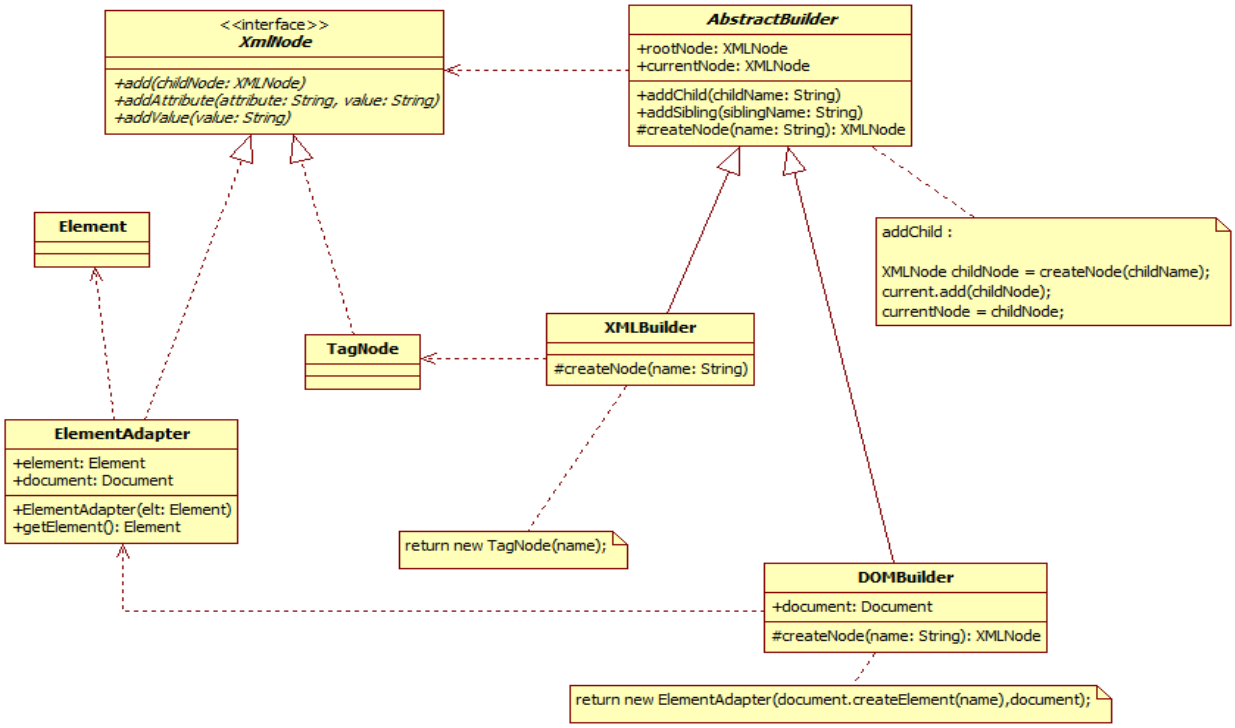
```
 public class DOMBuilder extends AbstractBuilder ...

        public void addAttribute(String name, String value) {
         currentNode.addAttribute(currentNode, name, value);
        }
    //...
```

en même temps que :

```
public class ElementAdapter …
        public void addAttribute(String name, String value) {
        getElement().setAttribute(name, value);
        }
    //...
```

Après toutes ces modifications, le code de DOMBuilder et XMLBuilder est très similaire. Il va donc être en grande partie remonté dans l'ancêtre.

Un point spécifique concerne la création des noeuds (TagNode ou Element). Cette création polymorphe est réglée par une méthode fabrique.

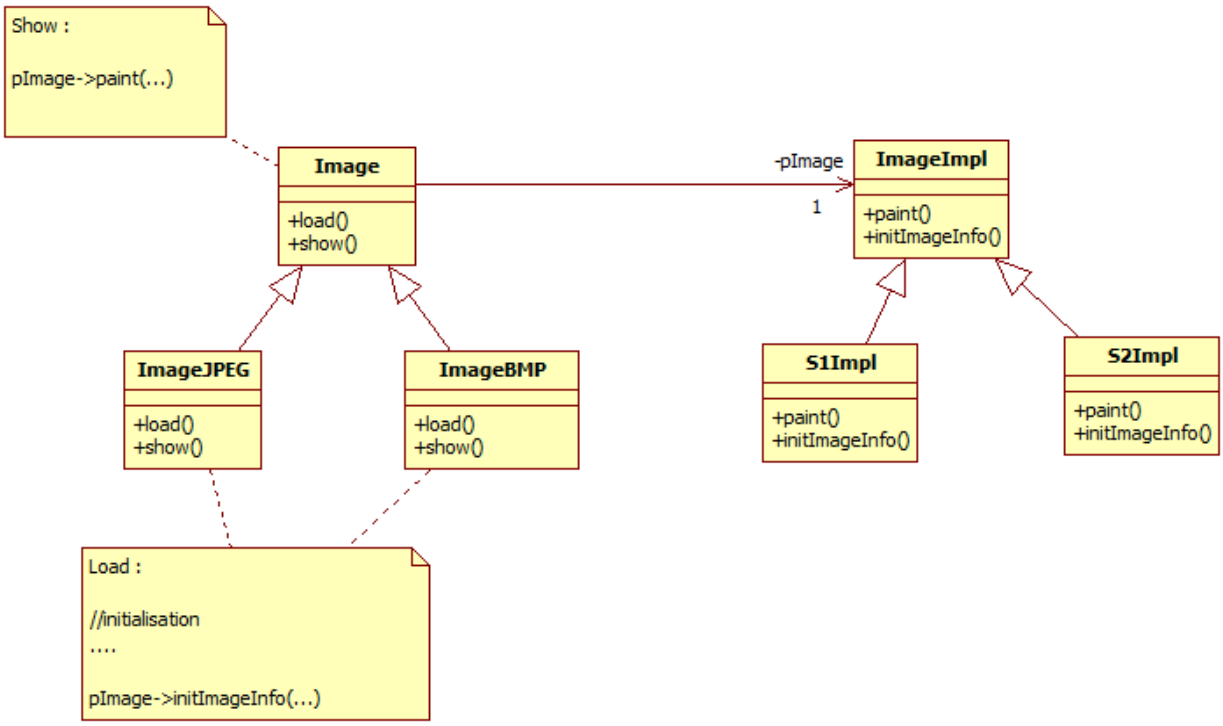

<<interface>>
XmlNode
+add(childNode: XMLNode)
+addAttribute(attribute: String, value: String)
+addValue(value: String)
AbstractBuilder
+rootNode: XMLNode
+currentNode: XMLNode
+addChild(childName: String)
+addSibling(siblingName: String)
#createNode(name: String): XMLNode
Element
TagNode
XMLBuilder
#createNode(name: String)
addChild :
XMLNode childNode = createNode(childName);
current.add(childNode);
currentNode = childNode;
ElementAdapter
+element: Element
+document: Document
+ElementAdapter(elt: Element)
+getElement(): Element
return new TagNode(name);
DOMBuilder
+document: Document
#createNode(name: String): XMLNode
return new ElementAdapter(document.createElement(name),document);

Pont

Nous voulons sauver / charger des images. Le concept d'image est représenté par la classe Image. Il y a plusieurs types d'images (format). Il faut aussi cibler plusieurs systèmes différents. Comment concevoir de façon à ce que le « client » des images ne dépende pas du système ?

Solutions pont

Une solution simple consisterait à créer une seule hiérarchie de classes, d'autant plus complexe qu'elle devrait évoluer.
En réalité, deux hiérarchies parallèles permettent de séparer l'interface de l'implémentation.



MVC / MVP

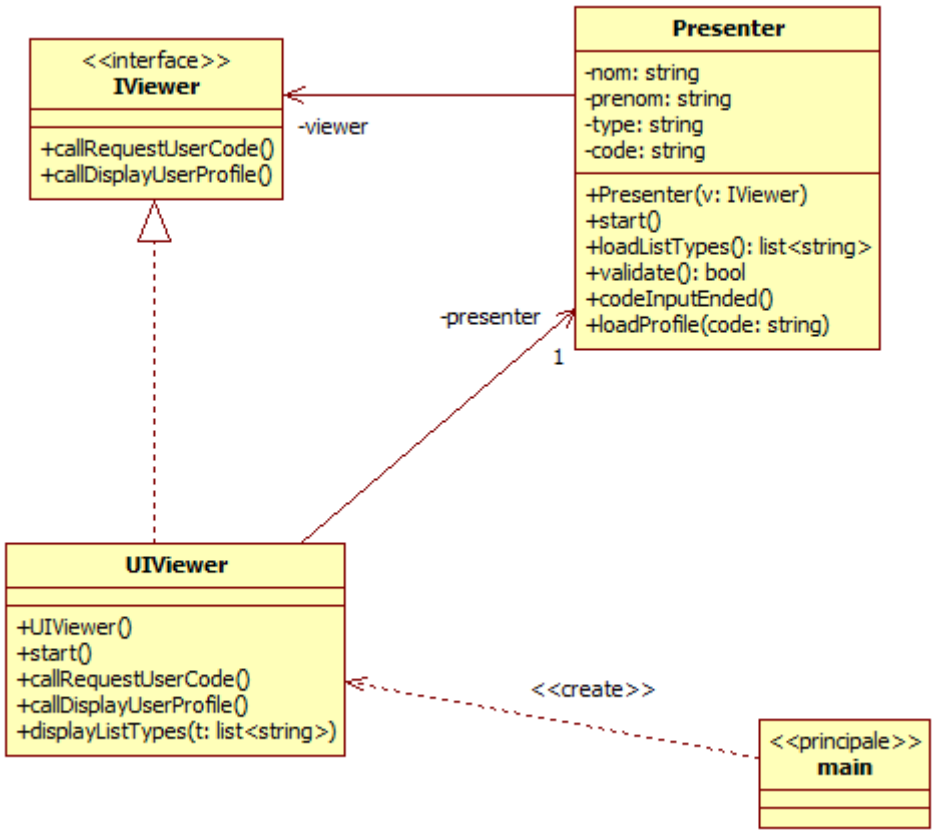
Appliquer le pattern MVC/MVP pour réaliser une application :

- demander le code d'un utilisateur
- demander son type (« local », « domaine »)
- afficher les informations de l'utilisateur
- les données utilisateur n'ont aucun intérêt ;) (afficher un truc bidon)
- interface console (ou mieux … ?)

Séquence :

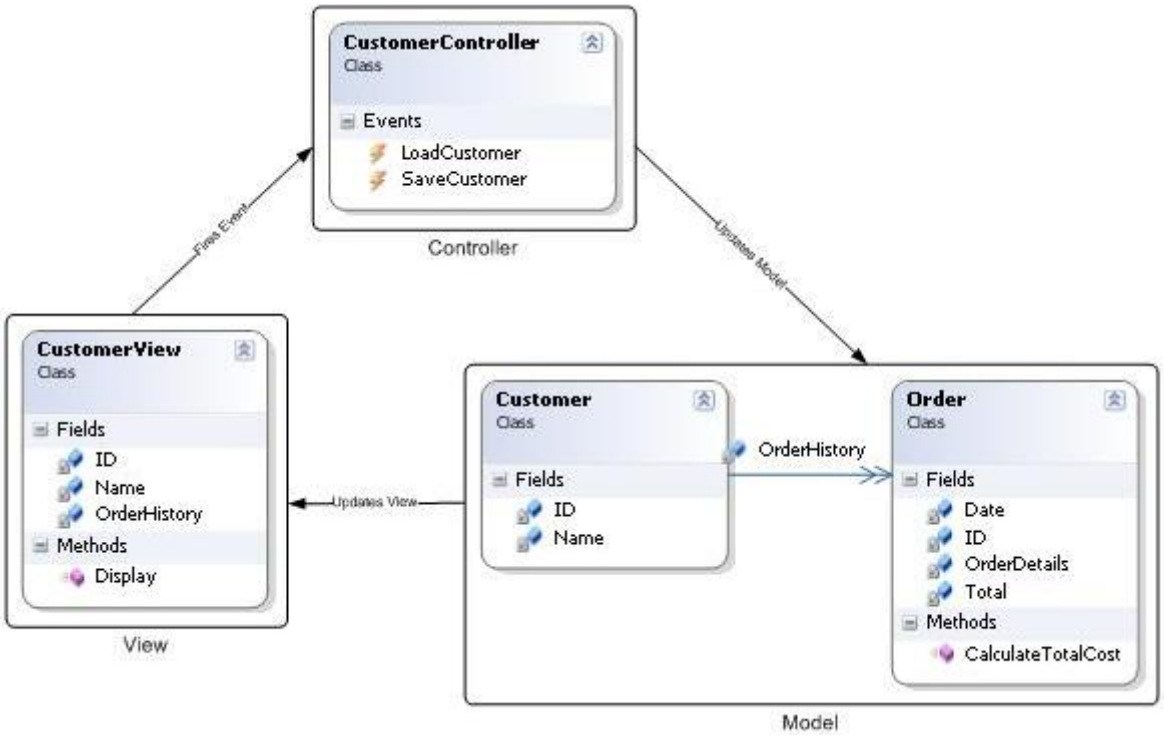
1) proposer les types possibles
2) attendre la réponse
3) demander le code utilisateur
4) valider et afficher les infos

Solution MVC / MVP :

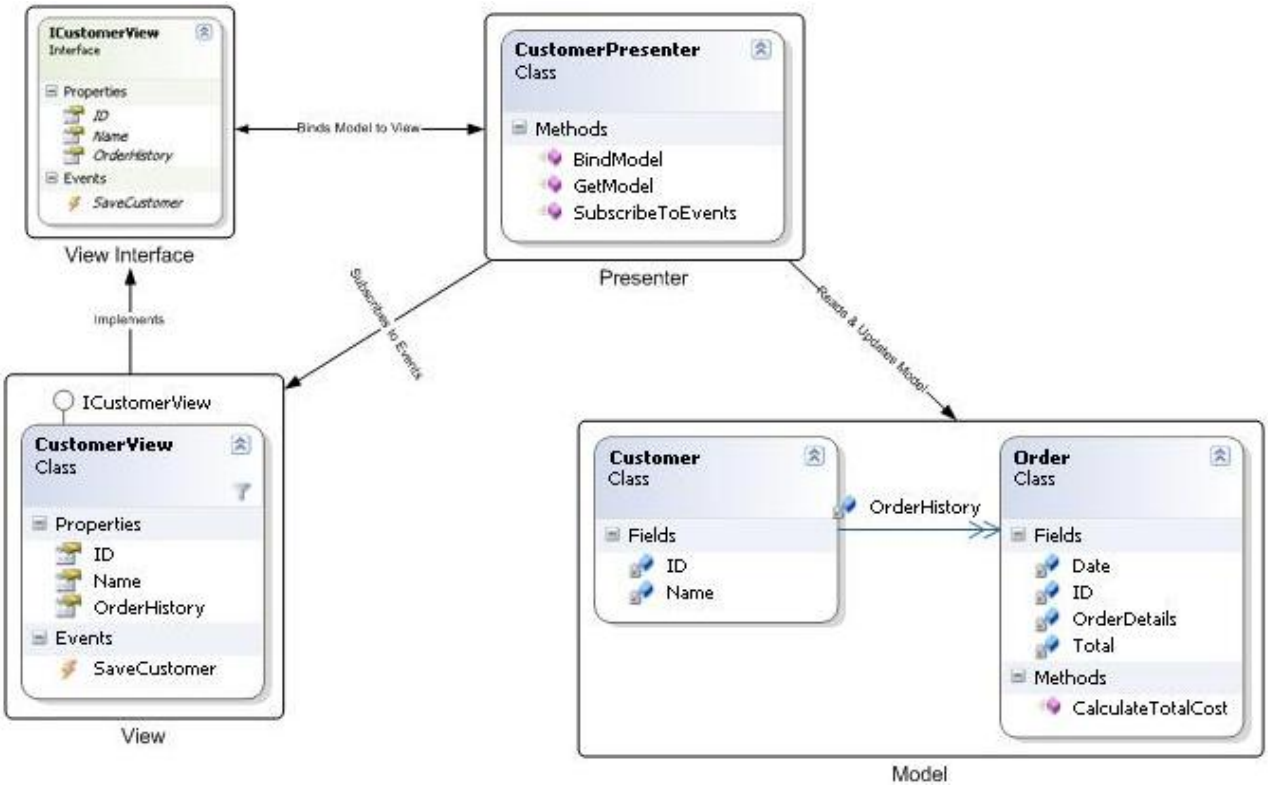


code pour la solution :

MVC


CustomerController
Class
Events
LoadCustomer
SaveCustomer
Controller
Fires Event
Updates Model
CustomerView
Class
Fields
ID
Name
OrderHistory
Methods
Display
View
Updates View
Customer
Class
Fields
ID
Name
OrderHistory
Order
Class
Fields
Date
ID
OrderDetails
Total
Methods
CalculateTotalCost
Model

MVP


ICustomerView
Interface
Properties
ID
Name
OrderHistory
Events
SaveCustomer
View Interface
Binds Model to View
CustomerPresenter
Class
Methods
BindModel
GetModel
SubscribeToEvents
Presenter
Implements
Subscribes to Events
Reads & Updates Model
ICustomerView
CustomerView
Class
Properties
ID
Name
OrderHistory
Events
SaveCustomer
View
Customer
Class
Fields
ID
Name
OrderHistory
Order
Class
Fields
Date
ID
OrderDetails
Total
Methods
CalculateTotalCost
Model

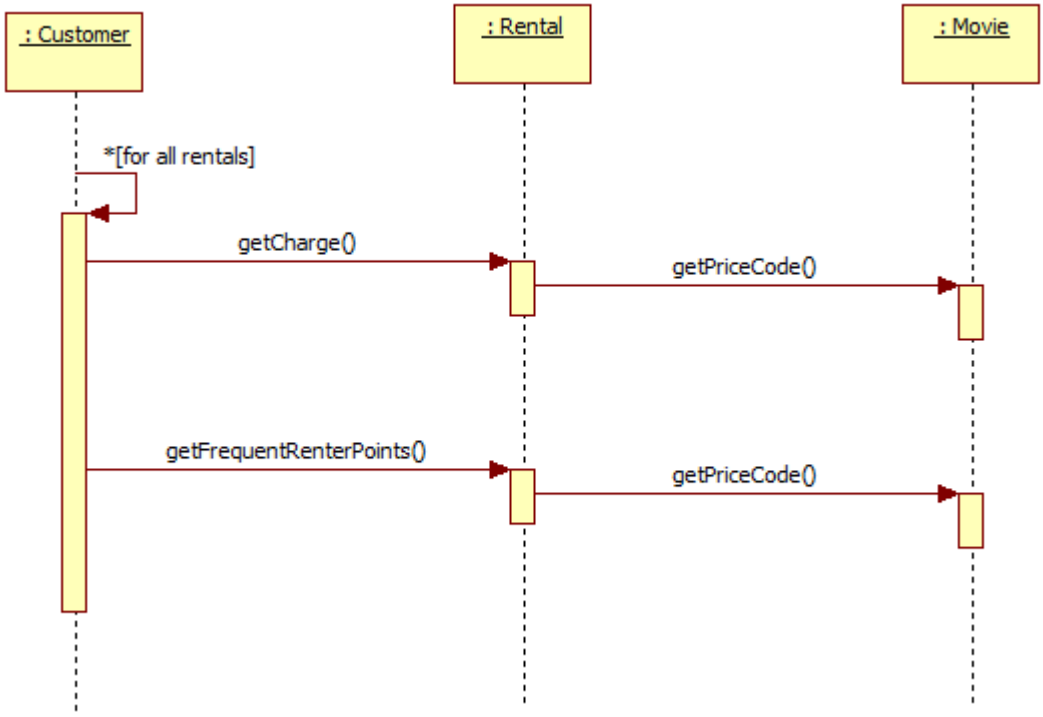
Refactoring


```
public String statement() {
        double totalAmount = 0 ;
        int frequentRenterPoints = 0 ;
        Enumeration rentals = _rentals.elements() ;
        String result = “Rental record for “ + getName() + ”\n” ;
        while ( rentals.hasMoreElements ( ) ) {
                double thisAmount = 0 ;
                Rental each = (Rental) rentals.nextElement ( ) ;

                //determine amount  for each line
                switch ( each.getMovie ( ) . getPriceCode ( ) ) {
                        case Movie.REGULAR :
                                thisAmount += 2 ;
                                if(each.getDaysRented ( ) > 2 )
                                        thisAmount += (each.getDaysRented ( ) - 2 ) * 1.5 ;
                        break;
                        case Movie.NEW_RELEASE :
                                thisAmount += each.getDaysRented ( ) * 3 ;
                        break;
                        case Movie.CHILDRENS :
                                thisAmount += 1.5 ;
                                if( each.getDaysRented ( ) - 3 ) * 1.5 ;
                        break;
                }

                //add frequent renter points
                frequentRenterPoints ++ ;
                //add bonus for a two days new release rental
                if ( (each.getMovie ( ).getPriceCode ( ) == Movie.NEW_RELEASE ) &&
                  each.getDaysRented ( ) > 1 ) frequentRenterPoints ++ ;

                //show figures for  this rental
                result += “\t” + each.getMovies ( ) .getTitle ( ) +”\t”+
                  String.valueOf ( thisAmount ) + “\n”;
        }
        //add footer lines
        result += “Amount owed  is “+String.valueOf ( totalAmount ) + “\n” ;
        result += “You earned “+ String.valueOf ( frequentRenterPoints ) +“frequent renter points“ ;
        return result ;
}
```

Comment faire évoluer la méthode statement, surtout s'il a fallu construire une fonction “htmlstatement” …. par copier-coller.

Etapes de solutions :

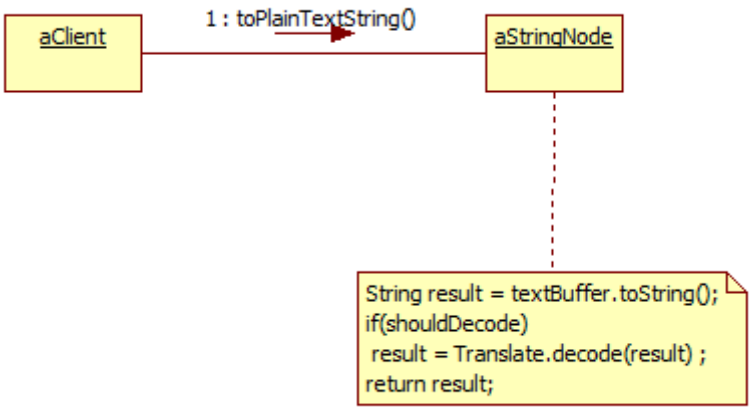
- ressortir le switch dans une nouvelle fonction
- renommer la variable locale … dans cette nouvelle fonction “getCharge”
- getDaysRented appartient à Rental.... donc déplacer le switch dans Rental
- éviter les variables locales (éfficacité ?)
- la boucle de frequentRenterPoints passe dans Rental



- le switch est basé sur un attribut d'un autre objet : passer le calcul dans “Movie”
- remplacement de la logique conditionnelle par des sous-classes de Movie

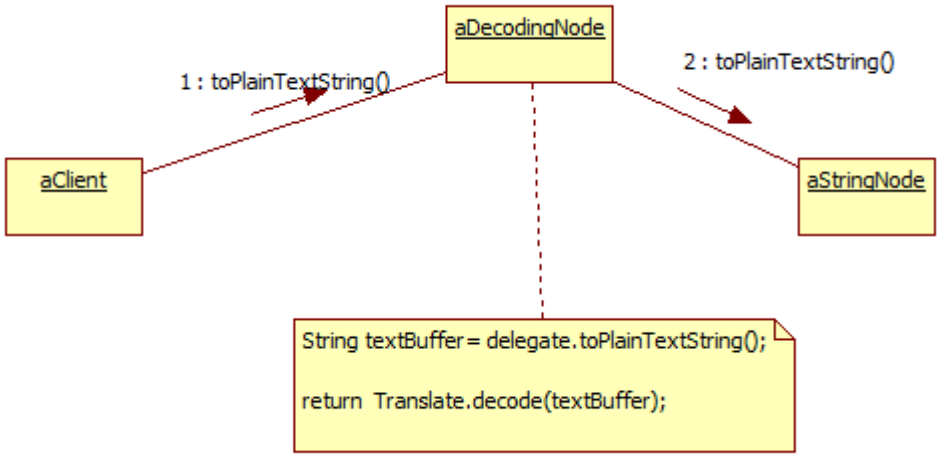
Decorateur

Voici un morceau de code :

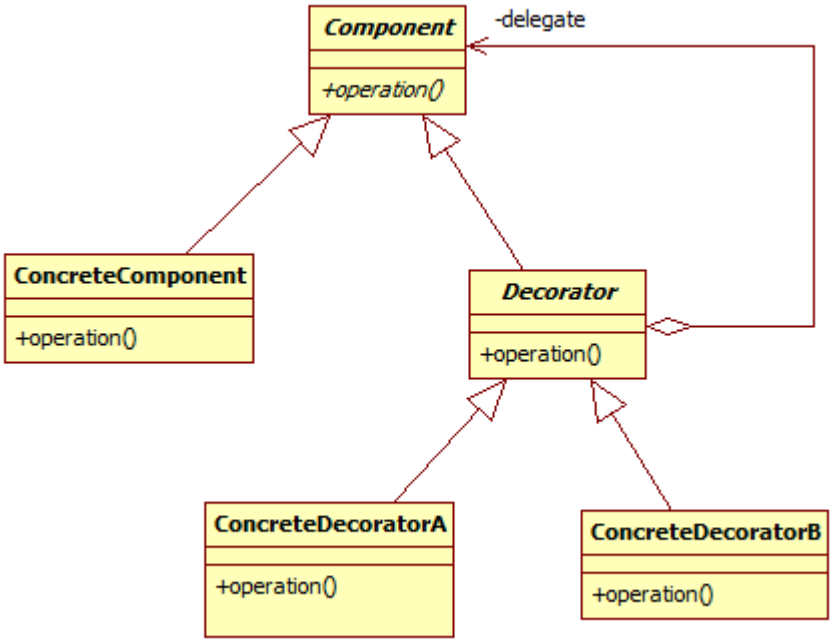


A force d'ajouter de nouveaux comportements ...le code va devenir difficile à contenir. Le paramétrage risque d'être compliqué aussi.

Solution :



Modèle de classe correspondant :


Component
+operation()
-delegate
ConcreteComponent
+operation()
Decorator
+operation()
ConcreteDecoratorA
+operation()
ConcreteDecoratorB
+operation()