CHAPITRE 14

Patrons de conception I:
Composite et Proxy

Sommaire

- Retour sur les patrons comme éléments architecturaux,
- Différents groupes de patrons,
- Concevoir un système de fichiers à l'aide de patrons:
  - Composite,
  - Proxy,
  - Visitor,
  - Template Method,
  - Singleton,
  - Mediator,
  - Observer,
  - Abstract Factory.

Les patrons comme éléments architecturaux

En UML, un patron de conception est modélisé comme une collaboration paramétrée

- Les paramètres de la collaboration sont les classes réelles qui vont participer à la collaboration,
- La hiérarchie de classes associée à la collaboration décrit la structure de la collaboration en utilisant les noms des paramètres,
- Les diagrammes d'interaction associés à la collaboration décrivent le comportement des éléments impliqués dans la collaboration en utilisant les noms des paramètres.

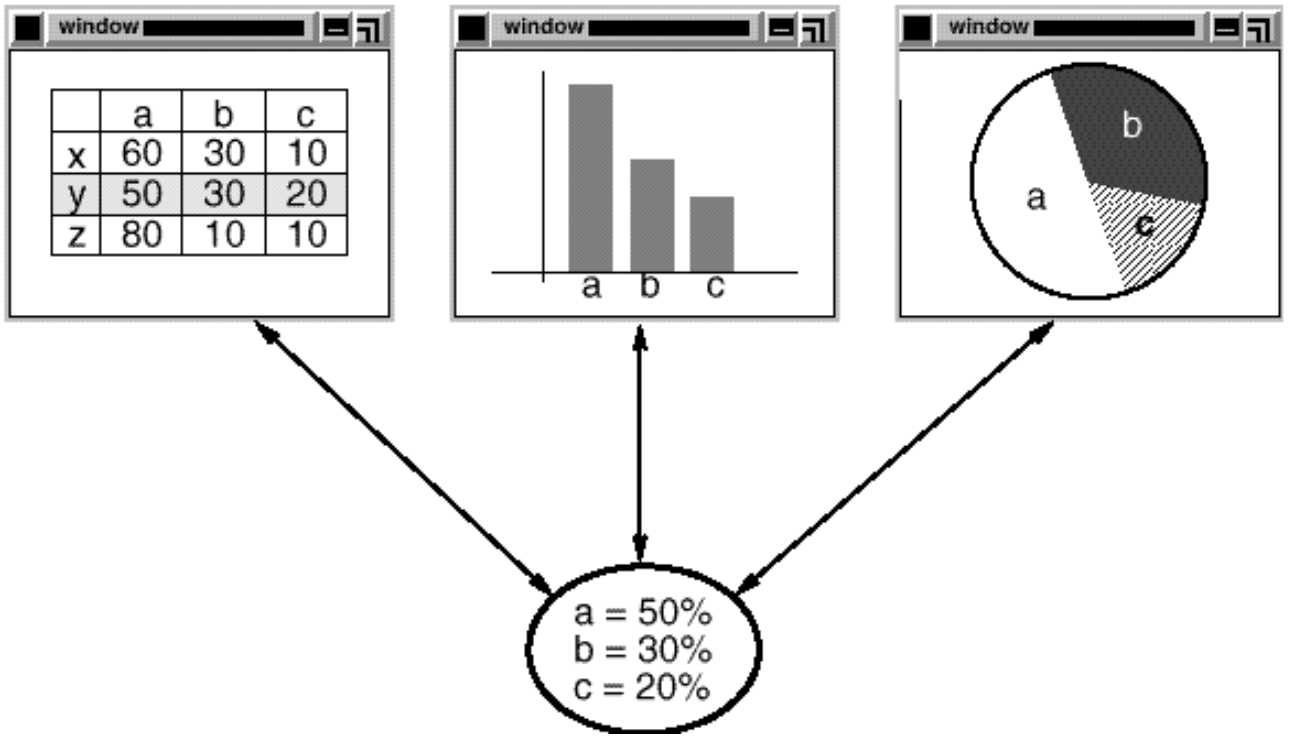
Exemples de patrons: les patrons utilisés dans le modèle Model-View-Controller (MVC)

Le modèle MVC contient trois types d'objets:

- Le modèle, qui contient les données de l'application,
- La vue, qui est la représentation du modèle à l'écran,
- Le contrôle, qui définit la façon dont l'interface usager réagit à une entrée faite par l'usager.

Le modèle MVC découple les vues et le modèle, et établit un protocole d'*abonnement /avertissement* (*subscribe/notify*) entre eux.

Exemples de patrons: les patrons utilisés dans le modèle Model-View-Controller (MVC)



Exemples de patrons: les patrons utilisés dans le modèle Model-View-Controller (MVC)

L'idée de découpler le modèle des vues permet de conserver une séparation nette entre le modèle et la façon de le représenter.

Cette idée est applicable à un problème plus général: découpler des objets de façon à ce qu'un changement à un objet puisse affecter un nombre variable d'autres objets sans que l'objet qui est changé ait à connaître en détail les autres objets.

Ce problème plus général est décrit par le patron de conception Observer.

Exemples de patrons: les patrons utilisés dans le modèle Model-View-Controller (MVC)

Une autre caractéristique du modèle MVC est de permettre l'imbrication de vues les unes dans les autres, afin de permettre la construction de vues plus complexes. MVC supporte les vues imbriquées à l'aide de la classe CompositeView, une sous-classe de View.

Un objet de la classe CompositeView se comporte exactement comme un objet de la classe View, et peut être utilisé partout où un objet de la classe View est utilisé, mais en plus, un objet de la classe CompositeView permet de gérer des vues imbriquées.

Exemples de patrons: les patrons utilisés dans le modèle Model-View-Controller (MVC)

Encore une fois, l'idée d'imbriquer des objets et de traiter l'objet résultant comme un seul objet individuel correspond à un problème de conception plus général.

Ce problème plus général est décrit par le patron de conception Composite.

Exemples de patrons: les patrons utilisés dans le modèle Model-View-Controller (MVC)

Le modèle MVC permet également de modifier la façon dont réagit une vue à une entrée faite par l'usager sans changer changer son apparence visuelle.

MVC encapsule le mécanisme de réponse dans un objet Controler. Les différents types de Controler sont organisés dans une hiérarchie de classe, et une vue utilise une instance d'une des sous-classes de Controler.

La relation View-Controler est décrite par le patron de conception Strategy.

Décrire un patron de conception

Pour décrire les patrons, Gamma et al. proposent un mode de documentation uniforme dans lequel on retrouve:

- Le nom du patron,
- Son intention,
- Une motivation,
- Son applicabilité,
- Sa structure,
- Les classes ou objets participants,
- Les collaborations entre les participants

De même que des commentaires sur les conséquences d'utilisation, sur certains compromis possibles, sur l'implantation et sur les patrons reliés.

Une ménagerie de patrons

		but		
		créationel	**structural**	**comportemental**
portée	**classe**	Factory method	Adapter	Interpreter
				Template Method
	objet	Abstract Factory	Adapter	Chain of Responsability
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Les 23 patrons de Gamma et al.

Objectifs

- Se familiariser avec l'utilisation des patrons de conception,
- Apprendre à identifier les bons patrons,
- Comprendre leur applicabilité,
- Apprendre à adapter un patron à nos besoins,
- Apprendre à évaluer efficacement les compromis durant la conception.

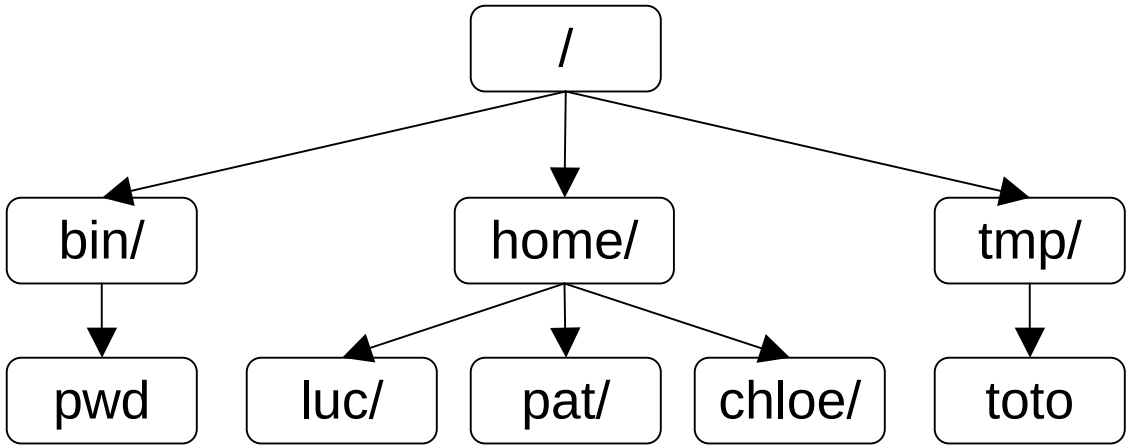
Cas d'utilisation: concevoir un système de fichiers

1. Structure du système,
2. Liens symboliques,
3. Fonctionnalité ouverte,
4. Protection mono-usager,
5. Protection multi-usagers
6. Avertissement de changement

1 - Structure du système de fichiers

Problème de conception:

- Représenter les éléments du système de fichiers (fichiers, répertoires),
- *Pour les usagers*: système de fichiers de grosseur et de complexité arbitraire,
- *Pour les programmeurs*: facile à manipuler et à étendre



Structure du système de fichiers

Une structure en arbre suggère un patron `Composite`

- On cherche un patron structural
- On veut maximiser l'uniformité et la flexibilité

Comment appliquer le patron

- Choisir les participants: `Component, Leaf et Composite`
- Choisir les opérations à traiter de façon uniforme

Patron Composite

Intention

Traiter les objets individuels et les objets multiples, composés récursivement, de façon uniforme.

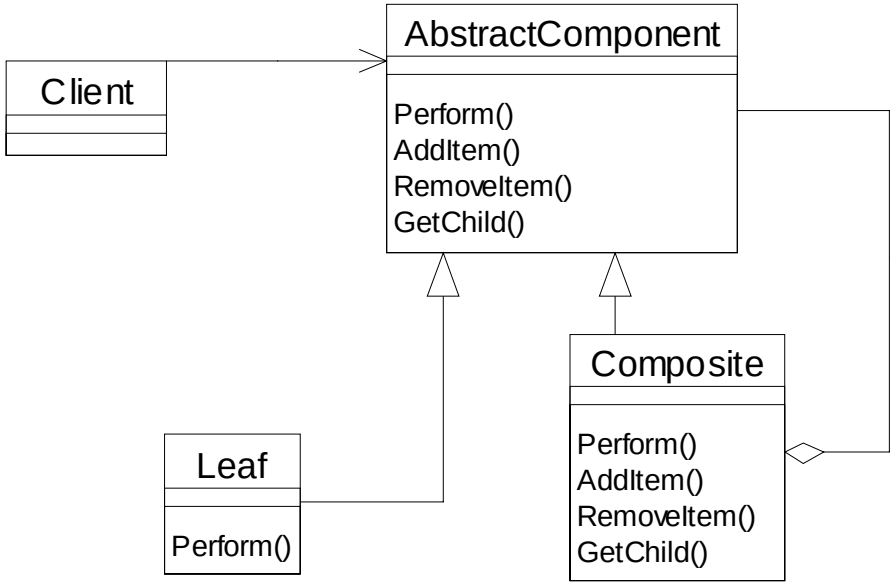
Applicabilité

Les objets doivent être composés récursivement,

et les objets dans la structure peuvent être traités uniformément,

et il ne devrait pas y avoir de distinction entre les objets individuels et composés,

Structure



Patron `Composite`

Conséquences

+ Uniformité: traite les composante uniformément sans égard à leur complexité
+ Extensibilité: les nouvelles sous-classes de Component fonctionnent partout où les anciennes fonctionnent.
- Coût: peut nécessiter un grand nombre d'objets

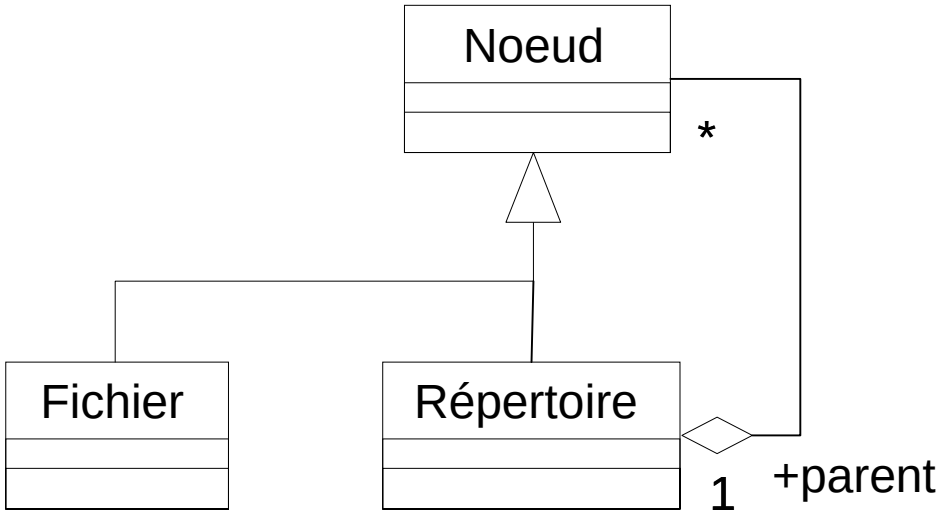
Implantation

- Les Components connaissent-ils leur parent ?
- L'interface est-elle uniforme entre les Leaves et Composites ?
- On ne doit pas allouer d'espace de stockage pour les enfants dans la classe de base Component
- Qui est responsable de détruire les enfants ?

Structure du système de fichiers

Établir la correspondance entre les participants au patron `composite` et les classes du système de fichier:

- Leaf, pour les objets qui n'ont pas d'enfants
  - ➡ Fichier, l'objet fichier
- Composite, pour les objets qui ont des enfants
  - ➡ Répertoire, l'objet répertoire
- Component, l'interface uniforme
  - ➡ Noeud



Structure du système de fichiers

Que peuvent faire les objets Fichier ?

- retourner leur nom et leur protection
- sérialiser leur contenu en entrée ou en sortie

Que peuvent faire les objets Répertoire ?

- retourner leur nom et leur protection
- énumérer leurs enfants
- adopter et abandonner des enfants

Structure du système de fichiers

Quelle interface uniforme Nœud devrait-il définir ?

- Retourner leur nom et leur protection
  - ➡ évidemment commun,
- Sérialiser leur contenu
  - ➡ moins évidemment commun,
- Énumérer leurs enfants
  - ➡ Nécessaire pour la récursion, il serait bon de cacher la structure interne,
  - ➡ Le patron Iterator pourrait être utilisé,
- Adopter et abandonner
  - ➡ Compromis entre la sécurité du typage et l'uniformité.

Structure du système de fichiers

Une interface uniforme pour adopter/abandonner simplifie les clients:

Dans la mesure où les objets Leaf peuvent traiter ces requêtes élégamment.

Exemple: `mkdir`

- "`mkdir nouvorep`"
  - ➡ Construit le nouveau sous-répertoire `nouvorep`
- "`mkdir sousdirA/sousdirB/nouvorep`"
  - ➡ Construit le nouveau sous-répertoire `nouvorep` dans `sousdirB`

Structure du système de fichiers

Une implantation naïve de `mkdir`

```
void mkdir( Répertoire* courant, string nouvorep )
{
   string nomfin = subpath( nouvorep );
   if( nomfin.length() == 0 )
      courant.adopter( new Répertoire(nouvorep) );
   else {
      string nomtete = head( nouvorep );
      Nœud* enfant = find( nomtete, courrant );
      if( enfant ) {
         mkdir( enfant, nomfin );
      } else {
         cerr << nomtete << " inexistant." << endl;
      }
}
```

Structure du système de fichiers

La méthode `find` cherche un enfant d'un nom donné. Cette méthode doit retourner un Nœud*:

```
Nœud* find( string nom, Répertoire& courant )
{
   Nœud* enfant = NULL;
   for(int i=0; enfant=courant.enfant(i); ++i )
      if( nom == enfant->nom() )
         return enfant;
   return NULL;
}
```

Structure du système de fichiers

Telle que définie, la fonction `mkdir` ne compile pas !

`mkdir` prend un Répertoire et non un Nœud

On pourrait s'en tirer en faisant un `dynamic_cast`:

```
Nœud* enfant = find( nomtete, courant );
if( enfant != NULL )
{
   Répertoire* rep =
               dynamic_cast<Répertoire*> enfant;
   if( rep != NULL )
      mkdir( rep, nomfin )
   else
      cerr << nomtete << "n'est pas un répertoire"
           << endl;
}
```

Structure du système de fichiers

Solution: traiter `adopter` et `abandonner` de façon uniforme

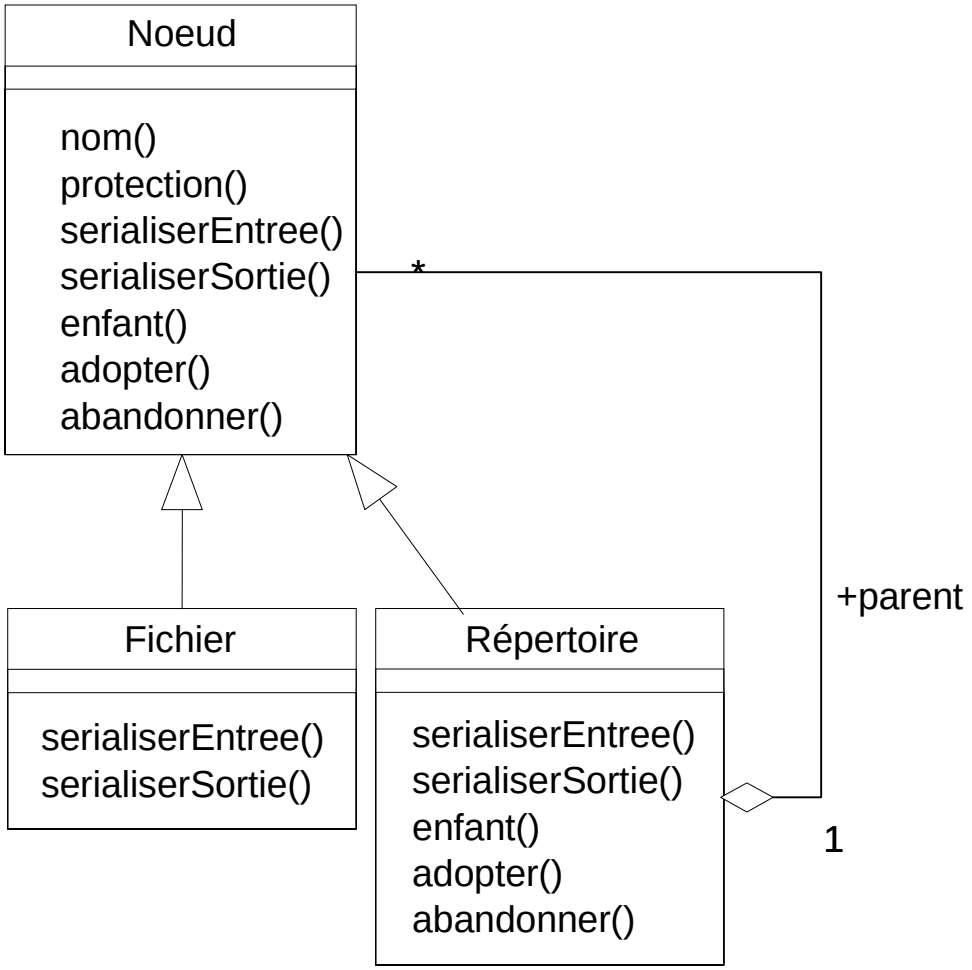
- On les déclare dans l'interface de Nœud
- On définit un comportement de défaut

```
class Nœud {
public:
   virtual void adopter( Nœud*)
   { cerr << nom() << "n'est pas un répertoire\n";}
   virtual void abandonner(Noeud* n)
   { cerr << n->nom() << "pas trouvé." << endl;}
};
```

La seule modification nécessaire à `mkdir`: changer sa signature:

```
void mkdir( Nœud* courrant, string nouvorep );
```

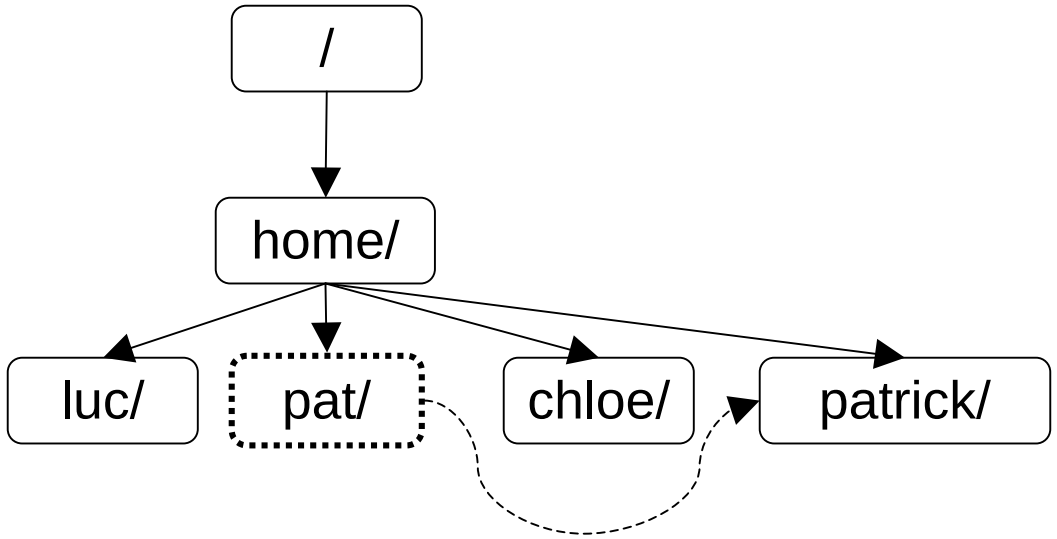
Structure du système de fichiers: résultat



2 – Liens symboliques

Problème de conception:

- Ajouter la fonctionnalité des liens symboliques de façon non invasive,
- Doit fonctionner au travers des répertoires, des systèmes de fichiers, et même des machines.



Liens symboliques

Identifier le bon patron de conception:

- Considérer de quelle façon les patrons de conception résolvent les problèmes de conception
  - ➡ C'est-à-dire lire le manuel: pas le temps !
- Parcourir les sections Intention (Intent) de chaque patron
  - ➡ La force brute
- Étudier comment les patrons sont inter-reliés (diagramme spaghetti, etc.)
  - ➡ Encore trop long, mais on se rapproche…

Liens symboliques

Identifier le bon patron de conception:

- Considérer les patrons visant le bon but (créationel, structural ou comportemental)
  - ➡ Un lien symbolique suggère un but structural
- Examiner les causes de reconception (p. 24)
  - ➡ On n'est pas encore rendu là, on veut juste concevoir
- Considérer ce que l'on veut rendre variable dans notre conception (table 1.2, p. 30)

Liens symboliques

Patrons de conception structural	Variabilité fournie par le patron
Adapter	Interface de l'objet
Bridge	Implantation de l'objet
Composite	Structure et composition de l'objet
Decorator	Responsabilités sans sous-classer
Facade	Interface à un sous-système
Flyweight	Coût de stockage des objets
Proxy	Mode d'accès à un objet ou sa localisation

Patron Proxy

Intention

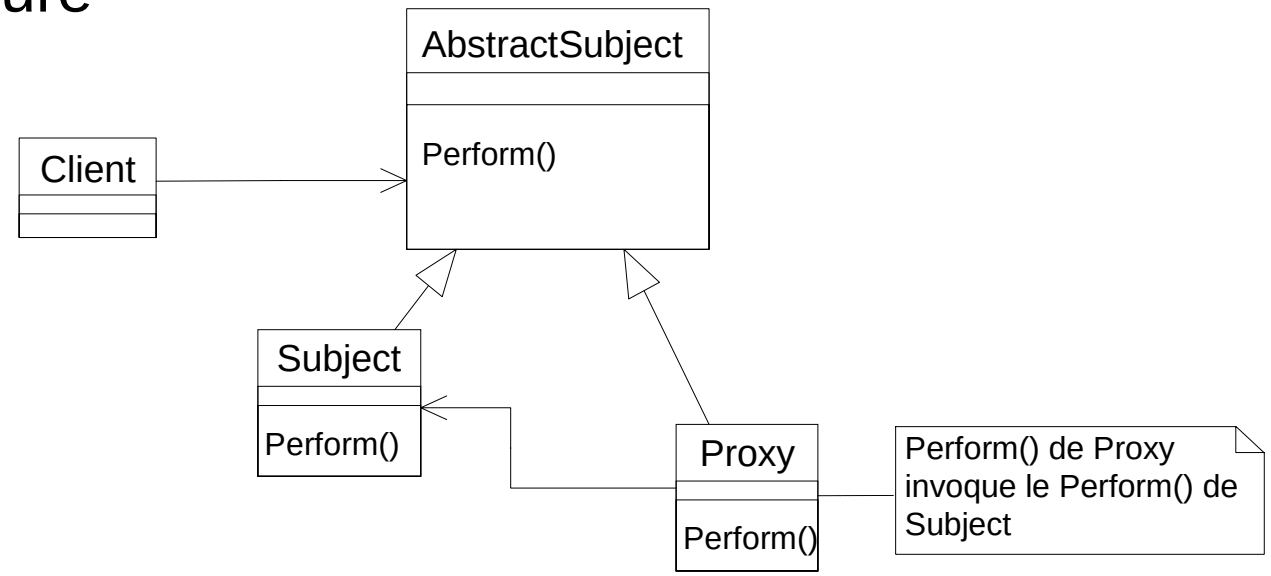
Fournir un remplaçant ou une doublure pour un autre objet afin de contrôler l'accès à ce dernier.

Applicabilité

Ce patron est applicable dès que le besoin d'une référence plus versatile ou plus sophistiquée qu'un simple pointeur se fait sentir.

L'interface du Proxy doit correspondre à l'interface du sujet.

Structure



Liens symboliques

Établir la correspondance entre les participants au patron `proxy` et les classes du système de fichier:

- `AbstractSubject`, l'interface à laquelle il faut correspondre
  - ➡ Noeud
- `Proxy`, la classe servant de remplaçant
  - ➡ Lien, l'objet lien symbolique
- `Subject`, l'objet auquel le `Proxy` réfère
  - ➡ ???

Problème: on ne veut pas lier `Subject` ni à `Fichier` ni à `Répertoire`

Liens symboliques

Solution: lire la description du participant Proxy dans le patron

> `[Proxy]` maintient une référence qui laisse le `Proxy` accéder au `Subject`. Le `Proxy` peut référencer un `AbstractSubject` si l'interface du `Subject` et de `l'AbstractSubject` sont compatibles.

Le `Subject` sera donc l'objet `Nœud`

> ➡ Ce choix ne serait pas possible sans l'interface uniforme choisie pour le patron `Composite`

Liens symboliques: implantation

Le lien délègue toutes les opérations à son `Subject`

Exemple:

```
class Lien : public Nœud {
public:
   Nœud* enfant(int n) {return _subject.enfant(n);}
};
```

Opération additionnelle spécifique à un lien:

```
Nœud* nœud() { return _subject; }
```

(pour les clients qui savent qu'ils sont en train de traiter un lien)

Liens symboliques: résultat