La modélisation avec UML: introduction aux patrons de conception

ACDA – CPOO (M3105)

Mathieu Sassolas

IUT de Sénart Fontainebleau
Département Informatique

Année 2015-2016
Cours 4

UPEC
UNIVERSITÉ PARIS-EST CRÉTEIL
IUT SENART FONTAINEBLEAU

Plan de la séance

1. Les patrons de conception
2. Patrons de création
   - Fabrique abstraite (*abstract factory*)
   - Singleton
3. Patrons de structuration
   - Composite
   - Adaptateur (*adapter*)
   - Décorateur (*decorator*)
   - Façade (*facade*)
   - Proxy
4. Patrons de comportement (semaine prochaine)

Plan de la séance

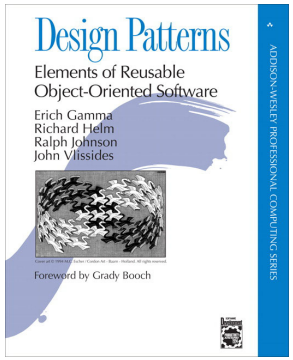
1. Les patrons de conception
2. Patrons de création
3. Patrons de structuration
4. Patrons de comportement (semaine prochaine)

Maintenant qu'on « connaît » UML...

- On a vu la syntaxe d'UML.
- Le meilleur moyen de s'exercer à la conception est de concevoir.
- Problème :
  - Professeur : « Construisez le diagramme de classe du système qui...
  - Étudiant : Je ne sais pas par où commencer ».
- Solution : fournir quelques recettes de base.

Historique des patrons de conception

- Lorsque les modèles ont commencé à s'unifier dans la syntaxe, on a remarqué que les même problèmes revenaient souvent.
- Les même solutions s'appliquent.
- Des motifs (*patterns*) apparaissent dans la conception (*design*) des programmes : les *design patterns*.



↪ Le *Gang of Four (GoF)* en identifie 23.

En français, le terme patron de conception est utilisé, car on s'en sert comme modèle.

Les types de patrons de conception

Construction : donnent des manières de créer de nouveaux objets.

Structuration : donnent des structure de programmes.

Comportement : donnent des moyens d'interaction entre objets (qui du coup contraignent la structure).

Cette année

On ne verra que quelques-un des patrons de conception.

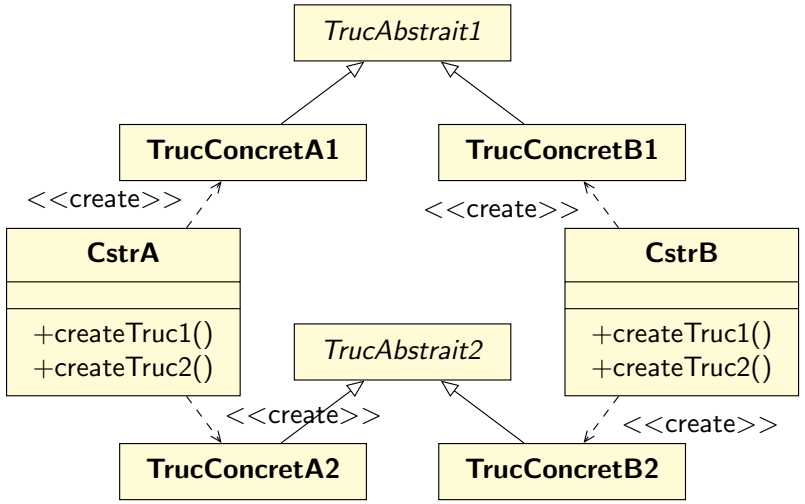
Souvent, on utilise ces patrons (ou plutôt motifs...) sans s'en rendre compte.

Plan de la séance

Plan de la séance

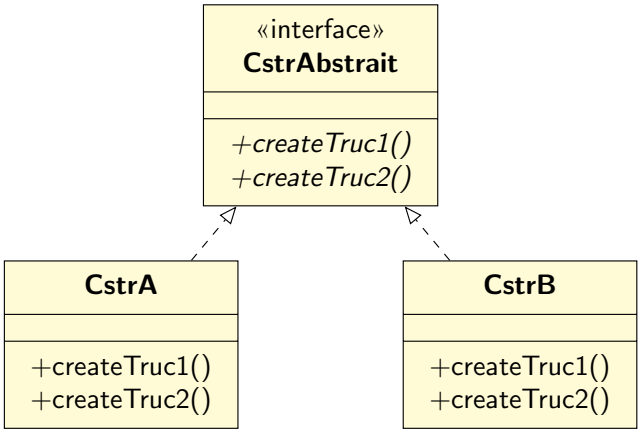
Problème

⇝ Des objets dont le type précis n'est pas forcément connu sont créés par des classes constructrices.



Idée de la solution

- Puisque les objets que l'on crée sont des concrétisations d'objets abstraits, on peut également abstraire les constructeurs.
- On abstrait les constructeurs par une interface.

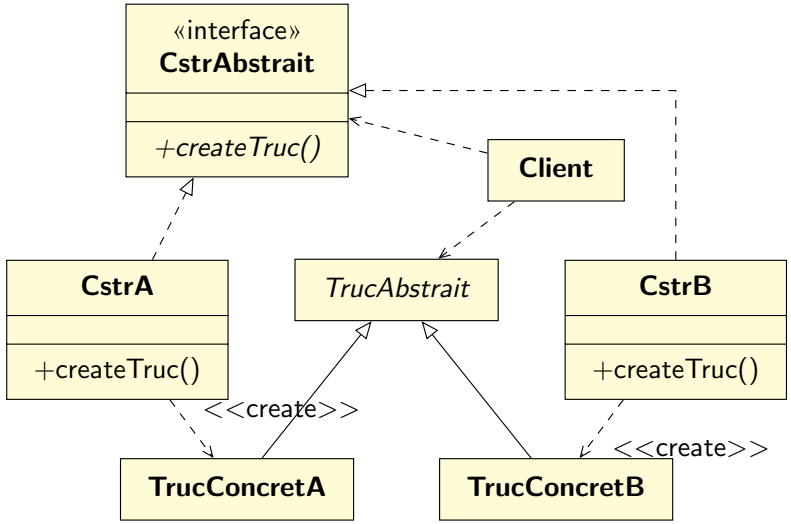


Avantages de cette abstraction

- Un « client » qui manipule des `Trucs` n'a pas besoin de connaître les détails de l'implémentation du `Truc`.
- On peut aisément étendre les `Trucs` avec une implémentation `TrucConcretC` simplement en fournissant la classe constructeur idoine `CstrC`.

Le diagramme de classe complet

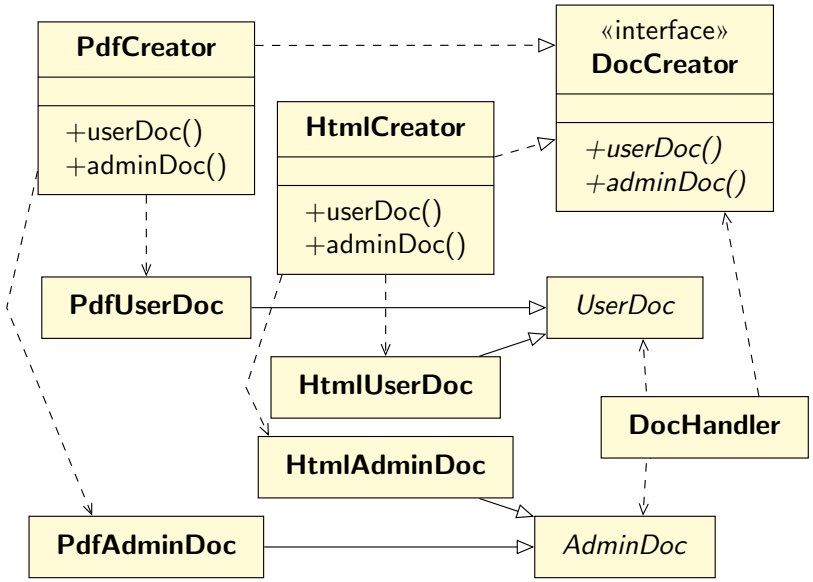
Avec un seul `Truc`



Exemple : le problème

- Dans notre système, on veut pouvoir fournir de la documentation en PDF ou en HTML. On envisage un jour d'ajouter d'autres formats.
- La documentation peut concerner l'utilisation de l'interface ou la maintenance du serveur. Ces documentations différentes sont manipulée différemment ; par exemple, l'accès à la documentation de maintenance peut inclure une vérification des permissions.
- Elle est générée à chaque demande afin de tenir compte des différentes versions de tous les modules installés.

Exemple



Plan de la séance

Problème

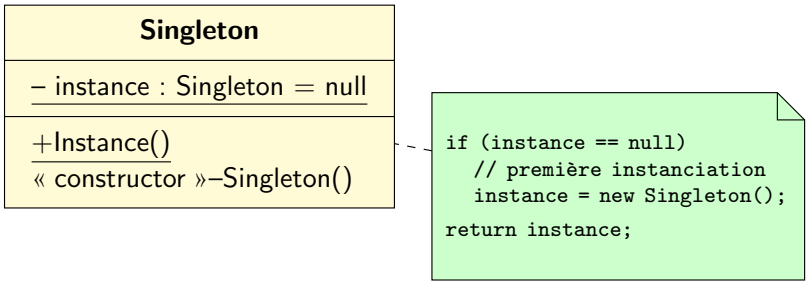
- Une classe dont on ne veut qu'une seule instance.
- Exemples courants : classe principale, classe de connexion à la base de donnée, constructeurs d'objets (cf fabrique abstraite)...

ClasseÀInstanceUnique

Solution

Idée

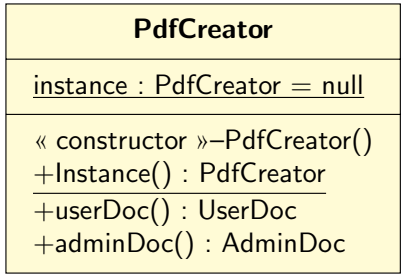
- La classe dispose d'un attribut de classe (*static*) du même type qui pointe vers l'unique instance.
- À la première instanciation, cet attribut est mis à jour.
- Pour les suivantes, on renvoie en fait le premier.
- Le véritable constructeur est privé.



Remarques

- L'appel de l'instance de la classe se fait donc par Singleton.Instance() et non par new Singleton().
- On utilise un autre « constructeur » Instance() car sinon on aurait une récursion infinie lors de la première instanciation.

Exemple



Exemple

PdfCreator

instance : PdfCreator = null

```
public class PdfCreator implements DocCreator {
  private static PdfCreator instance = null;
  private PdfCreator(){}
  public static PdfCreator Instance(){
    if (instance == null)
      // première instanciation
      instance = new PdfCreator();
    return instance;
  }
  public UserDoc userDoc(){...}
  public AdminDoc adminDoc(){...}
}
```

Exemple

PdfCreator

instance : PdfCreator = null

```
public class PdfCreator implements DocCreator {
  private static PdfCreator instance = null;
...
thePdfCreator = PdfCreator.Instance();
AdminDoc freshPdfAdmDoc = thePdfCreator.adminDoc();
...
      instance = new PdfCreator();
    return instance;
  }
  public UserDoc userDoc(){...}
  public AdminDoc adminDoc(){...}
}
```


Plan de la séance

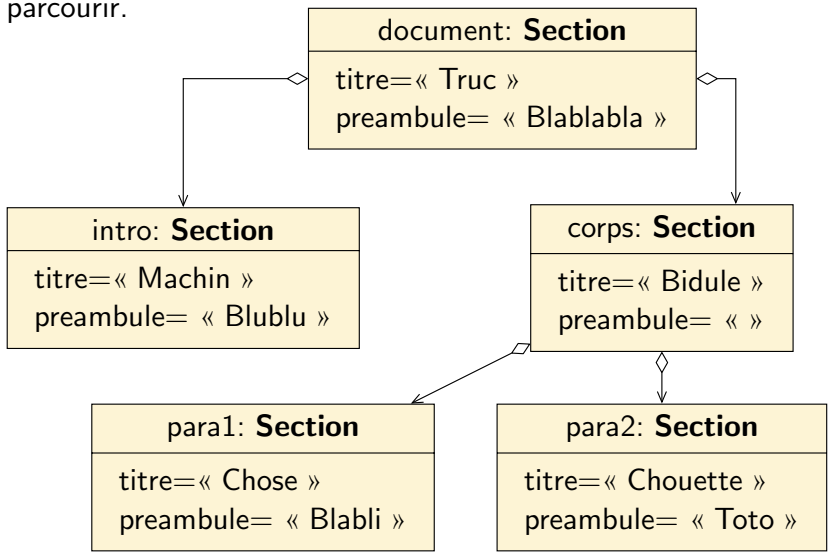
1. Les patrons de conception
2. Patrons de création
3. Patrons de structuration
   - Composite
   - Adaptateur (*adapter*)
   - Décorateur (*decorator*)
   - Façade (*facade*)
   - Proxy
4. Patrons de comportement (semaine prochaine)

Plan de la séance

1. Les patrons de conception
2. Patrons de création
3. Patrons de structuration
   - Composite
   - Adaptateur (*adapter*)
   - Décorateur (*decorator*)
   - Façade (*facade*)
   - Proxy
4. Patrons de comportement (semaine prochaine)

Problème

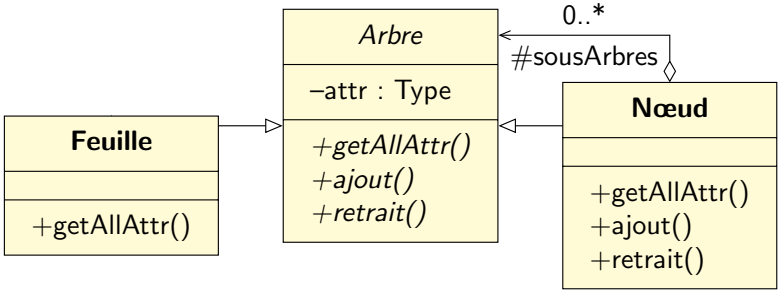
⇝ Nos objets sont organisés en arbre. Il faut pouvoir les parcourir.



Solution

Idée

- On divise en classe feuille et classe nœud qui héritent d'une classe principale.
- Les accès se font récursivement dans les descendants.
- Il faut prévoir des accesseurs pour ajouter/enlever des composants.

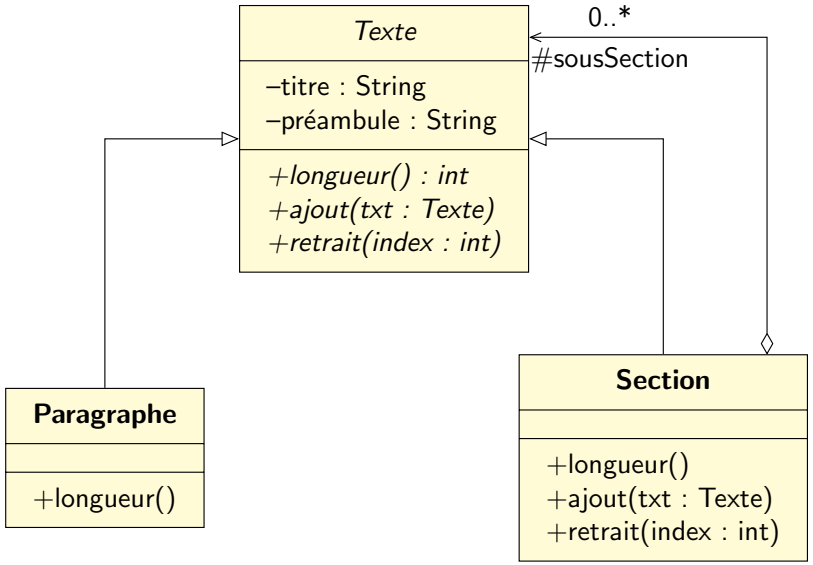


Solution

```
public class Feuille extends Arbre {
  ...
  public Type getAllAttr(){return attr;}
...
}
```

```
public class Nœud extends Arbre{
  ...
  public Type getAllAttr(){
    Type res = attr;
    for (Arbre desc: sousArbres) {
      res.append(desc.getAllAttr());
    };
    return res;
  }
  ...
}
```


Exemple



Exemple

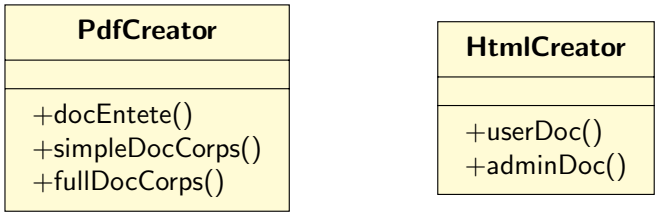
```
public class Paragraphe extends Texte {
  ...
  public int longueur(){
    return (titre.length() + préambule.length());
  }
```

```
public class Section extends Texte{
  ...
  public int longueur(){
    int res = 0;
    res = res + titre.length() + préambule.length();
    for (Texte subsec: sousSections) {
      res= res + subsec.longueur();
    };
    return res;
  }
  ...
}
```

Plan de la séance

Problème

⇝ Les différentes implémentations d'une même fonctionnalité sont fournies avec des noms différents, voire en plusieurs parties.



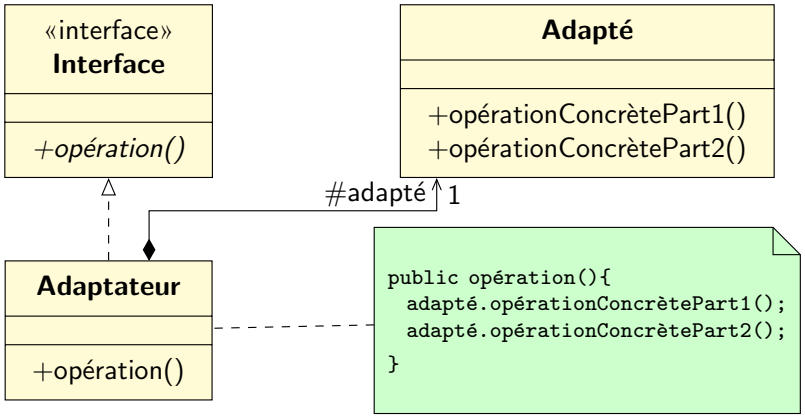
C'est le cas lorsque...

- On travaille avec des classes/composants créés par quelqu'un d'autre (importées).
- On a mal conçu le système au premier abord (il aurait fallu utiliser une fabrique abstraite).

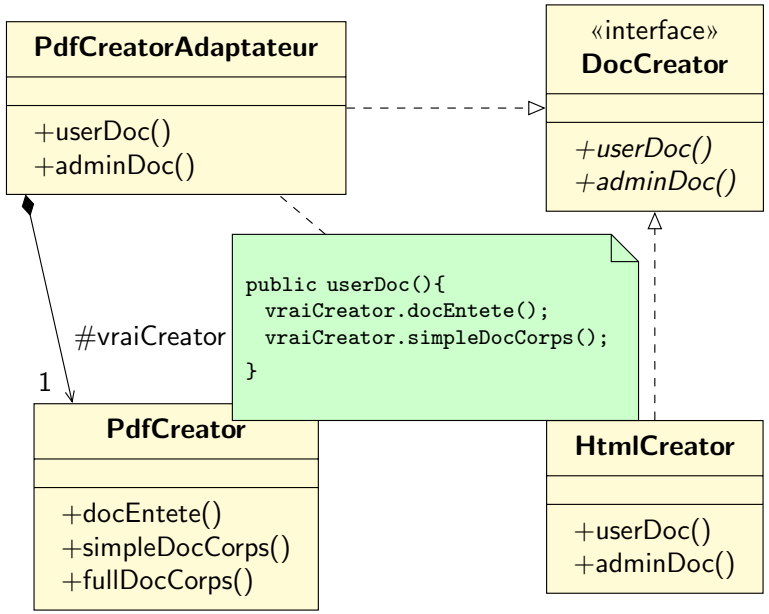
Solution

Idée

Une classe `Adaptateur` s'occupe de la traduction entre les méthodes demandées par l'interface et les méthodes implémentées par l'`Adapté`.



Exemple



Plan de la séance

Problème

On veut étendre une (ou deux) opération de base dans certains cas autrement que par héritages.

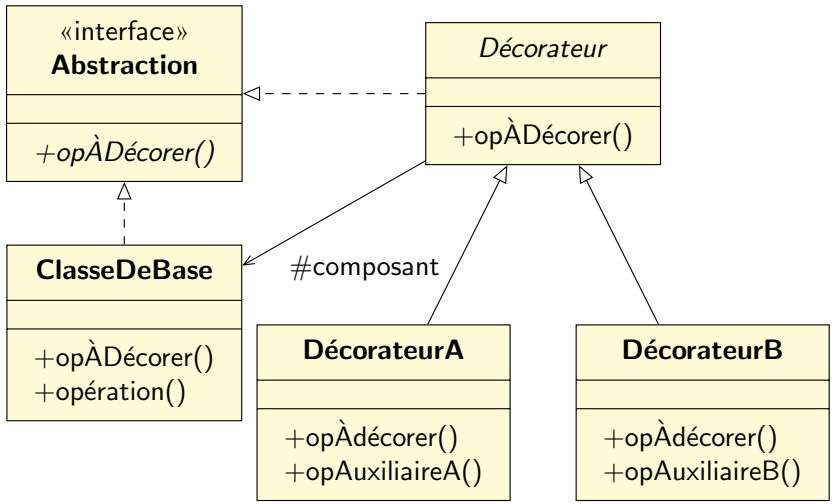
Pourquoi pas l'héritage ?

- Pas très joli : hériter juste pour modifier une fonction, c'est moche.
- Moins flexible : si l'on veut ajouter d'autres extensions, on change drastiquement le modèle.

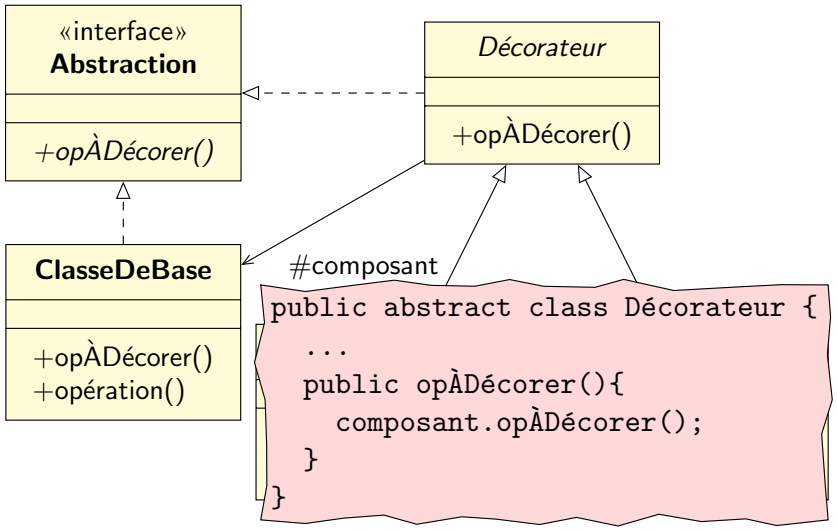
Solution
Idée

- On définit un `Décorateur` qui pointe vers l'objet de base et redéfinit seulement l'opération en question.
- On peut définir plusieurs versions (d'autres extensions) en héritant du décorateur.
- Le décorateur et la classe de base sont abstraits par une interface.

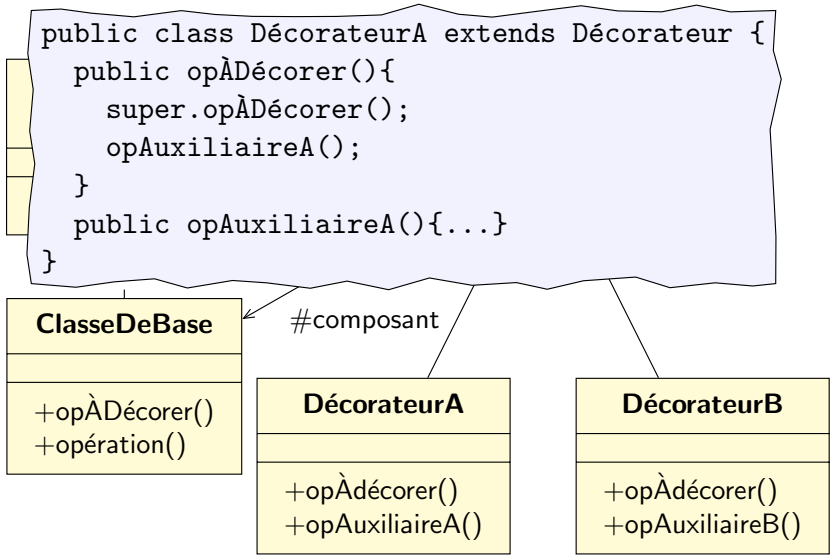
Solution
Diagramme de classes



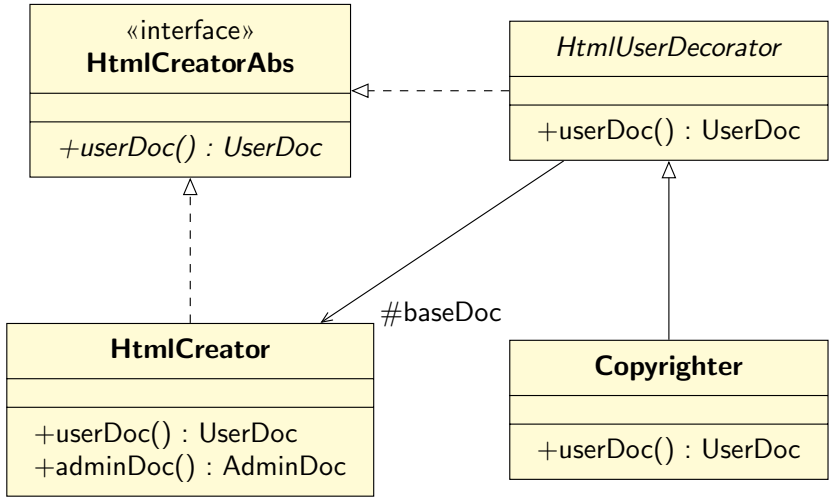
Solution
Diagramme de classes



Solution
Diagramme de classes



Exemple



Exemple

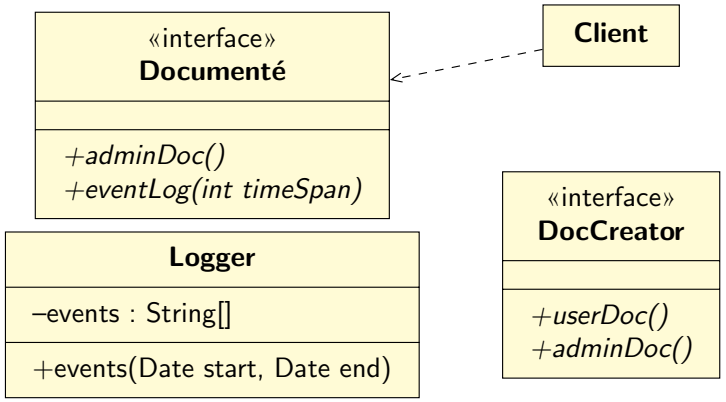

```
public abstract class HtmlUserDecorator {
  protected HtmlCreator baseDoc;
  public HtmlUserDecorator(HtmlCreator theDoc){
    this.baseDoc = theDoc;
  }
  public UserDoc userDoc(){
    return baseDoc.userDoc();
  }
}
```

```
...
public UserDoc userDoc() {
  UserDoc res = super.userDoc();
  res.append("<!-- Copyright 2014 IUT S&M -->");
  return res;
}
...
```

Plan de la séance

Problème

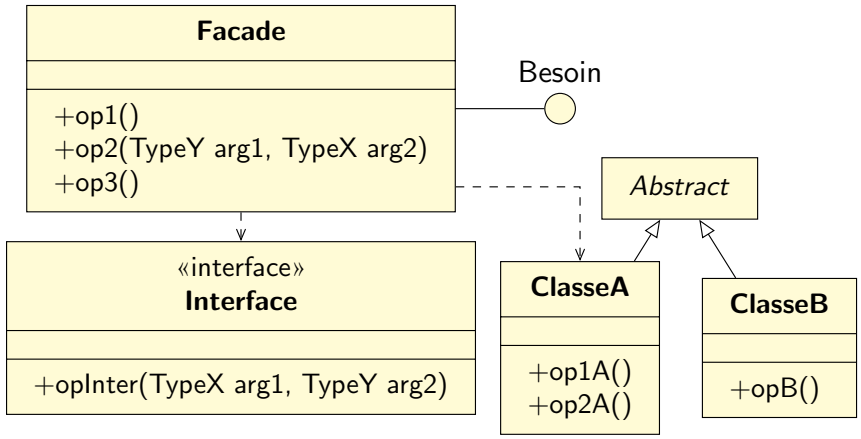
- Besoin d'une interface qui implémente certaines fonctions.
- Ces fonctions sont implémentées mais par différentes classes/interfaces.
- Ou bien il ne manque pas grand chose pour les implémenter depuis ces fonctions.



Solution

Idée

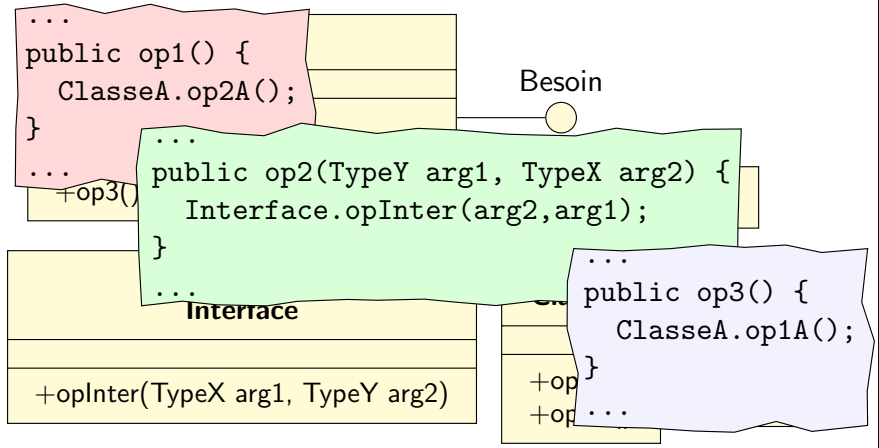
Regrouper dans une classe fournissant l'interface tout ce dont on a besoin en le « piochant » ailleurs.



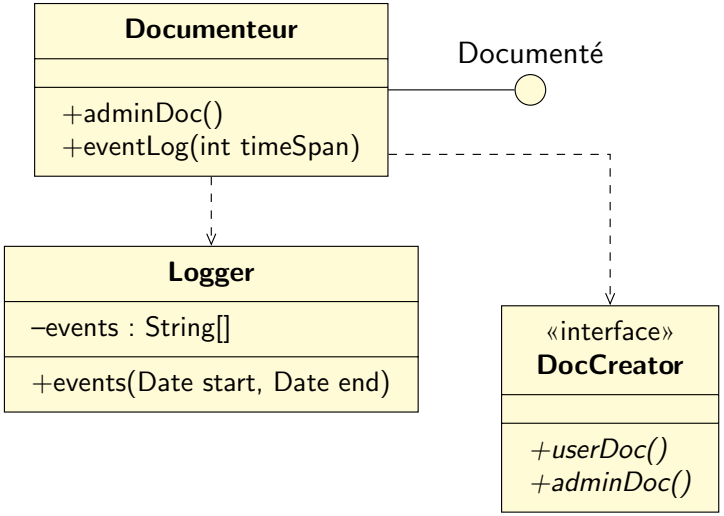
Solution

Idée

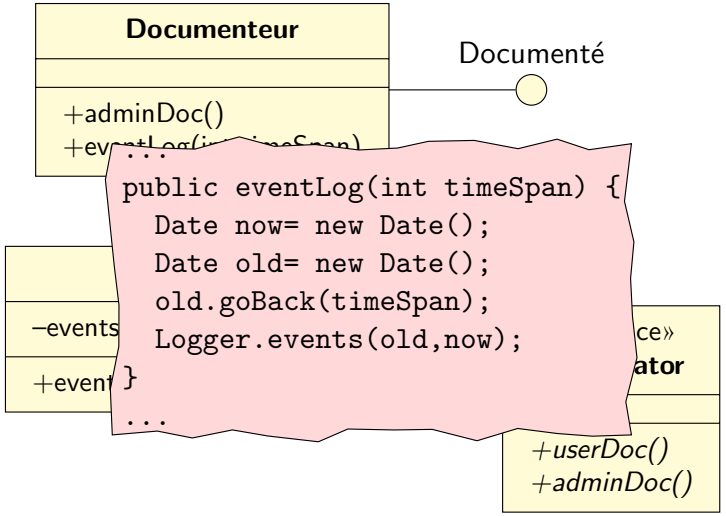
Regrouper dans une classe fournissant l'interface tout ce dont on a besoin en le « piochant » ailleurs.



Exemple



Exemple



Plan de la séance

Problème

- Un objet est coûteux à invoquer, on ne veut pas le créer vraiment tant qu'on n'en a pas besoin.
- Ou un objet est sur une autre ressource, que l'on ne veut pas utiliser à chaque fois.
- Ou bien un objet nécessite des droits d'accès que l'on veut vérifier à chaque accès ; on souhaite également séparer la vérification de l'accès.

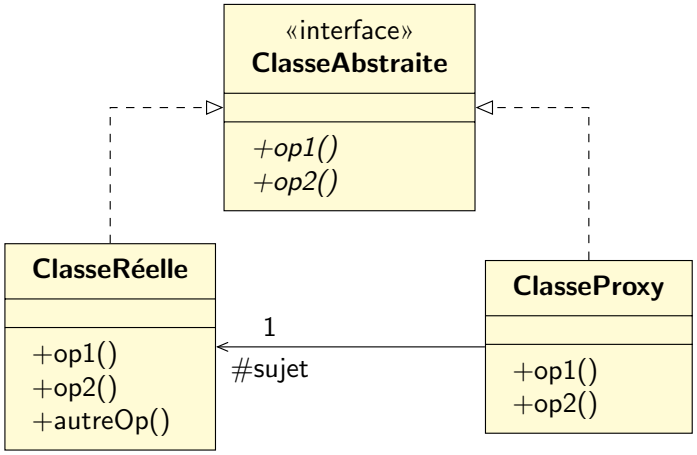
Exemple

C'est le cas des `Query` vers les bases de donnée en Java : la requête n'est vraiment exécutée que sur demande expresse.

Solution

Idée

La classe est abstraite par une interface implémentée à la fois par le proxy et par la « vraie » classe.




```
public class ClasseProxy implements ClasseAbstraite {
  protected ClasseRéelle sujet = null;
  public op1(){
    if (sujet == null) {
      sujet = new ClasseRéelle();
      // Création
    }
    sujet.op1(); // Délégation
  }
  public op2(){
    if (sujet == null) {
      println("Erreur");
    }else{
    sujet.autreOp();
    sujet.op2(); // Délégation
    }
  }
}
```


Plan de la séance