conception et prog objet avancés

DEV.3.4– CPOO

Florent Madelaine

IUT de Sénart Fontainebleau
Département Informatique

Année 2022-2023
Cours 5

Les notions pour l'évaluation

1. Usage : DCU, synopsis.
2. Structure : DC sans les méthodes, DO.
3. Comportement : DC avec méthodes, Diagramme de Séquence.
4. Abstraction : DC avec classes abstraites et interfaces, API, couplage faible.
5. Patrons : solutions à des problèmes typiques.

Quelques patrons

Nous avons un temps limité et donc je ne peux pas parler de tous les patrons de conception.

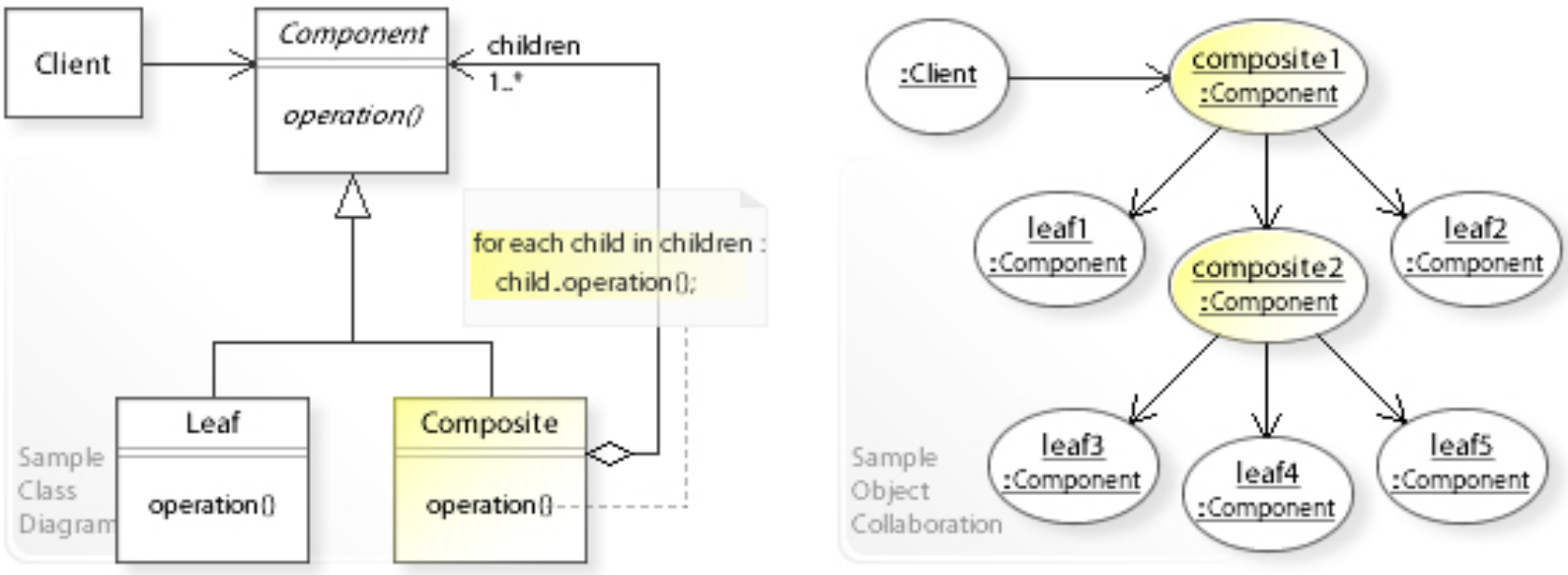
- Decorateur
- Composite

Problème

On veut traiter de manière uniforme les feuilles d'une structure arborescente comme les noeuds internes.
Ceci permet de faciliter l'écriture des méthodes récursives qui calculent des choses des feuilles vers la racine.

- Le cas de base se retrouve dans la classe feuille.
- Le cas de récurrence dans la classe noeud interne.

Idée



By Vanderjoe [CC BY-SA 4.0], from Wikimedia Commons

Exemple

On veut modéliser des fichiers et des répertoires. Sur un élément du système de fichier, on veut connaître le nombre de fichiers qu'il regroupe (en se comptant lui-même).

Solution

Component

```
interface  EverythingIsAFile{
    public int countFiles();
}
```

leaf

```
class DataFile implements EverythingIsAFile {
    public int countFiles(){
        return 1;
    }
}
```

composite

```
import java.util.*;

class Directory implements EverythingIsAFile {
    private List<EverythingIsAFile>  ls = new ArrayList<EverythingIsAFile>();
    public int countFiles() {
        int count = 1 ; // Me the directory, I count for 1.
        for (EverythingIsAFile item : ls) {
            count += item.countFiles();  // Delegation
        }
        return count;
    }
    public void mv(EverythingIsAFile item) {
        ls.add(item);
    }
    public void rm(EverythingIsAFile item) {
        ls.remove(item);
    }
}
```

Disclaimer

Le problème qui va suivre risque de vous choquer.

Don’t try it at home.

La meilleure fête

Le problème

- organisation sous forme d'arbre enraciné.
- une personne a un *fun factor* (entier ≥ 0).
- fête = ensemble de personnes qui ne sont pas voisins.
- La qualité de la fête est la somme des fun factors.

Question

Calculer la meilleure fête possible.

Calcul de bas en haut

Problème

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

Pourquoi pas l'héritage ?

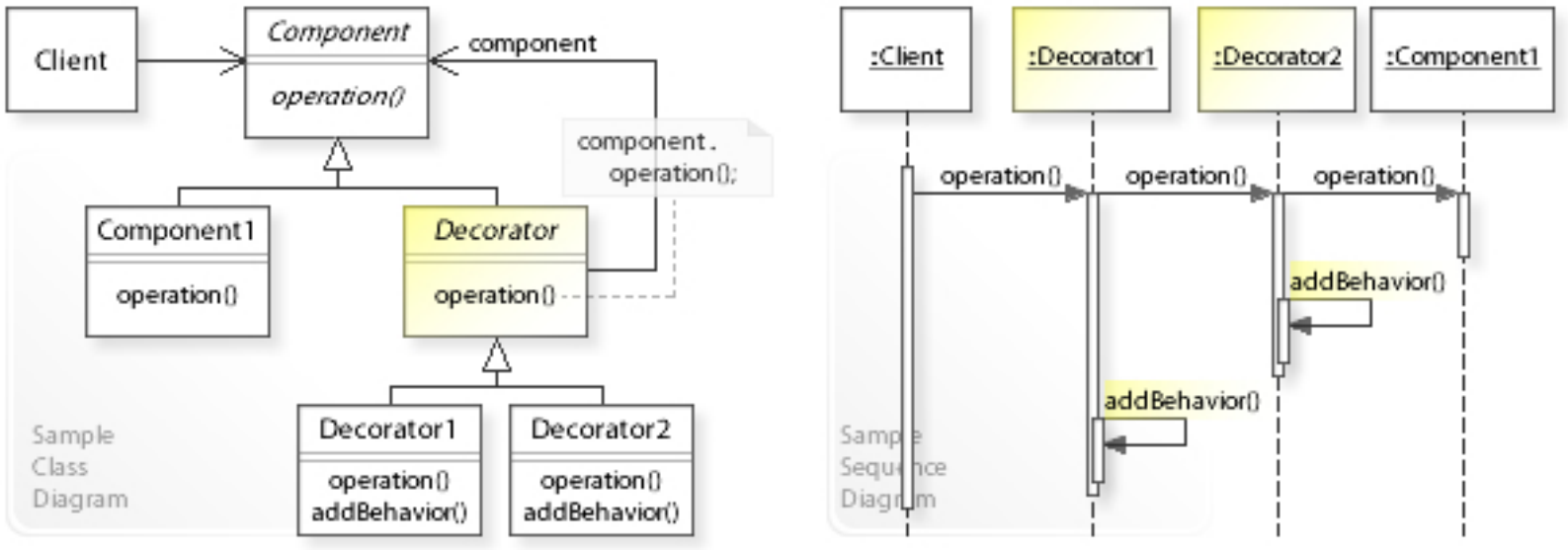
- Pas très joli : hériter juste pour modifier une fonction, c'est moche.
- Moins flexible : si l'on veut ajouter d'autres extensions, on change drastiquement le modèle.

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.

Idée



By Vanderjoe [CC BY-SA 4.0], from Wikimedia Commons

Exemple : fenêtres

```java
// The Window interface class
public interface Window {
    void draw(); // Draws the Window
    String getDescription(); // Returns a description of the Window
}

// Implementation of a simple Window without any scrollbars
class SimpleWindow implements Window {
    @Override
    public void draw() {
        // Draw window
    }
    @Override
    public String getDescription() {
        return "simple window";
    }
}
```

Exemple : le decorator abstrait

```java
// abstract decorator class - note that it implements Window
abstract class WindowDecorator implements Window {
    private final Window windowToBeDecorated; // the Window being deco

    public WindowDecorator (Window windowToBeDecorated) {
        this.windowToBeDecorated = windowToBeDecorated;
    }
    @Override
    public void draw() {
        windowToBeDecorated.draw(); //Delegation
    }
    @Override
    public String getDescription() {
        return windowToBeDecorated.getDescription(); //Delegation
    }
}
```

Exemple : le decorator avec scrollbar verticale

```
// The first concrete decorator which adds vertical scrollbar function
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window windowToBeDecorated) {
        super(windowToBeDecorated);
    }

    @Override
    public void draw() {
        super.draw();
        drawVerticalScrollBar();
    }

    private void drawVerticalScrollBar() {
        // Draw the vertical scrollbar
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", including vertical scrollba
    }
}
```

Exemple : le decorator avec scrollbar horizontale

```
// The second concrete decorator which adds horizontal scrollbar funct
class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window windowToBeDecorated) {
        super(windowToBeDecorated);
    }

    @Override
    public void draw() {
        super.draw();
        drawHorizontalScrollBar();
    }

    private void drawHorizontalScrollBar() {
        // Draw the horizontal scrollbar
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", including horizontal scroll
    }
}
```

Les patrons de créations

singleton faire en sorte d'avoir une classe qui n'a jamais plus d'une instance

factory déléguer la construction d'un objet dont on ne veut pas vraiment connaître le type exact (étendant une classe abstraite)

abstract factory pousser la logique encore plus loin en ne voulant même pas connaître le type exact de la fabrique.

Problème

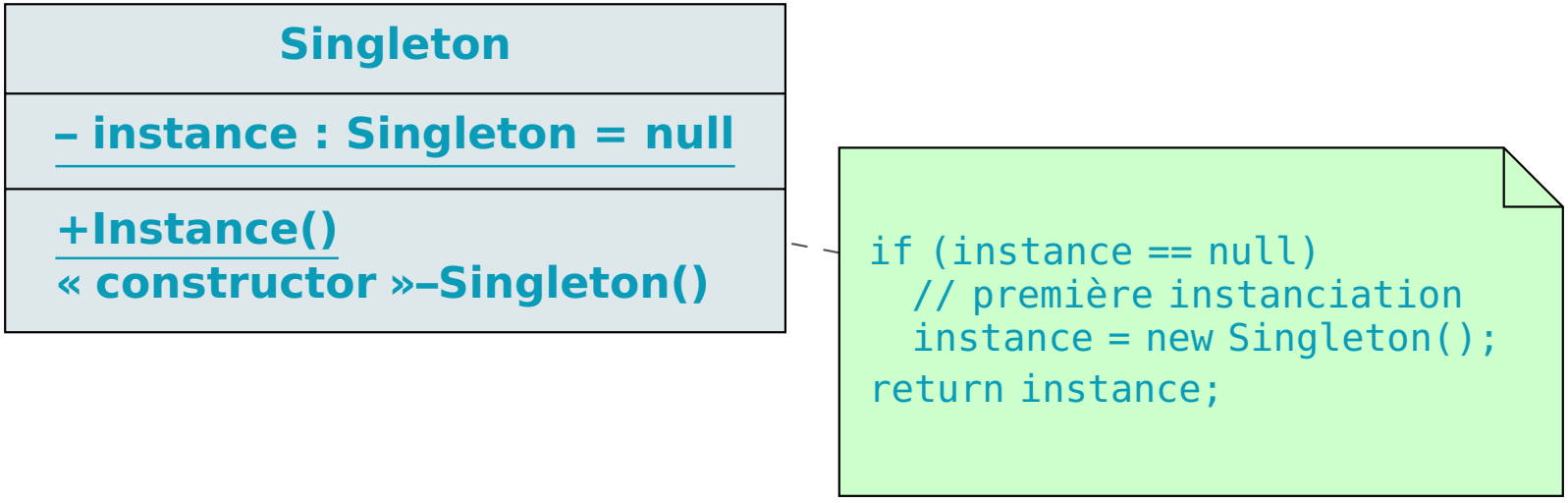
- Une classe dont on ne veut qu'une seule instance.
- Exemples courants : classe principale, classe de connexion à la base de donnée, constructeurs d'objets (cf fabrique abstraite). . ..

ClasseÀInstanceUnique

Solution

Idée

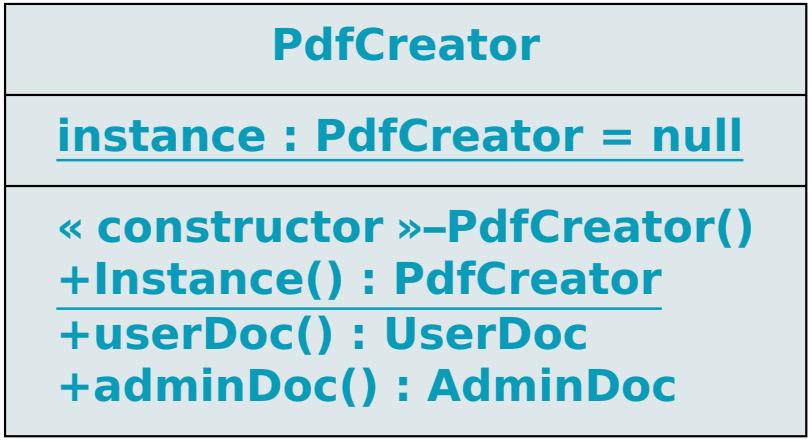
- La classe dispose d'un attribut de classe (*static*) du même type qui pointe vers l'unique instance.
- À la première instanciation, cet attribut est mis à jour.
- Pour les suivantes, on renvoie en fait le premier.
- Le véritable constructeur est privé.



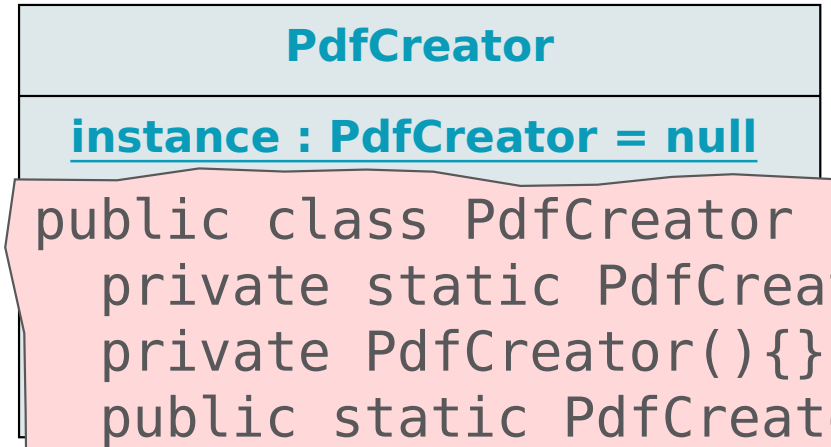
Remarques

- 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

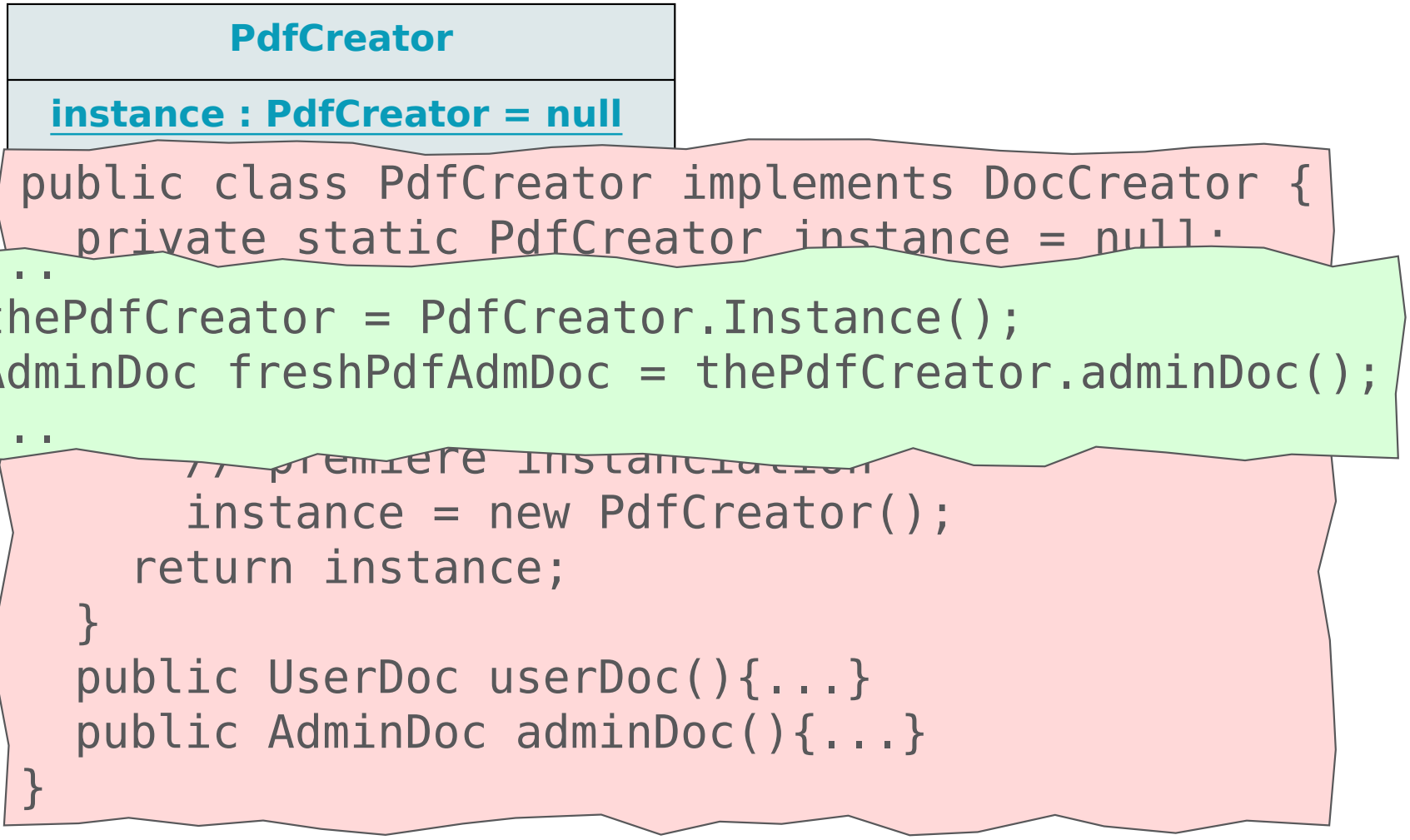

PdfCreator
instance : PdfCreator = null
« constructor »–PdfCreator()
+Instance() : PdfCreator
+userDoc() : UserDoc
+adminDoc() : AdminDoc

Exemple




```
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(){...}
}

Autre exemple

Dans le JDK, la classe `Java.lang.Runtime` donne accès à l'environnement et permet par exemple d'obtenir des informations sur le *nombre de processeurs* ou encore la *mémoire disponible*.
Notez qu'il n'y a pas besoin d'avoir plusieurs instances. En fait il vaut mieux avoir une seule instance puisque cette classe agit avec la machine virtuelle java.

Exemple

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

    /**
     * Returns the runtime object associated with the current Java applica
     * Most of the methods of class <code>Runtime</code> are instance
     * methods and must be invoked with respect to the current runtime obj
     *
     * @return  the <code>Runtime</code> object associated with the curren
     *          Java application.
     */
    public static Runtime getRuntime() {
        return currentRuntime;
    }

    /** Don't let anyone else instantiate this class */
    private Runtime() {}
```

Exemple (suite)

```
    /**
     * Terminates the currently running Java virtual machine by initiating its
     * shutdown sequence.  This method never returns normally.  The argument
     * serves as a status code; by convention, a nonzero status code indicates
     * abnormal termination.
     ...
     */
    public void exit(int status) {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkExit(status);
        }
        Shutdown.exit(status);
    }

    /**
     * Registers a new virtual-machine shutdown hook.
     ...
     */
    public void addShutdownHook(Thread hook) {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPermission(new RuntimePermission("shutdownHooks"));
        }
        ApplicationShutdownHooks.add(hook);
    }

   ...

}
```

Fabrique par un exemple

L’exemple suivant est repris intégralement de la page wikipedia suivante.

`https://en.wikipedia.org/wiki/Factory_method_pattern#Java`

Exemple

```
public abstract class Room {
}

public class MagicRoom extends Room {
}

public class OrdinaryRoom extends Room {
}

public abstract class MazeGame {
    private final List<Room> rooms = new ArrayList<>();

    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        rooms.add(room1);
        rooms.add(room2);
    }

    abstract protected Room makeRoom();
}
```

Exemple (suite)

```
public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}

public class OrdinaryMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}

MazeGame ordinaryGame = new OrdinaryMazeGame();
MazeGame magicGame = new MagicMazeGame();
```

Intérêt de la fabrique

Pourquoi ?

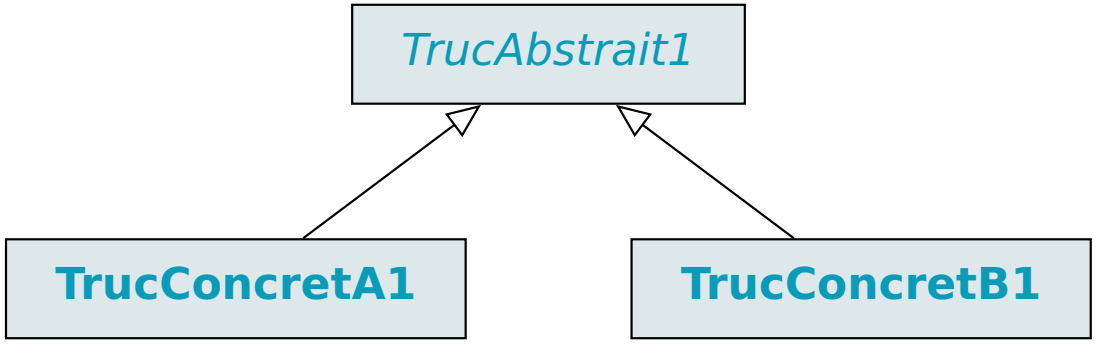
- On découple en sortant la création d'une classe dans une nouvelle classe, la factory.
- La classe qu'on souhaite construire est cachée derrière un type abstrait.
- Cette factory est responsable de fabriquer l'instance en appelant un constructeur concret.
- On peut ainsi avoir plusieurs classes concrètes du type abstrait à construire et le choix est dans la factory.
- Les classes clientes ne manipulent que le type abstrait.

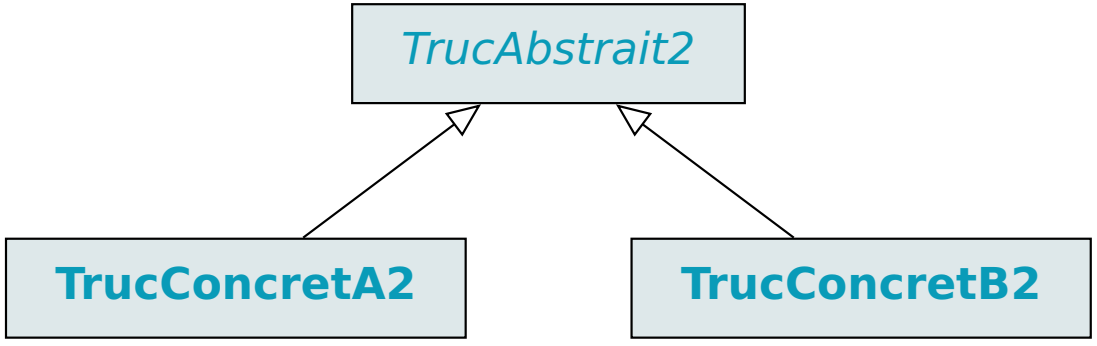
Problème

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

Problème

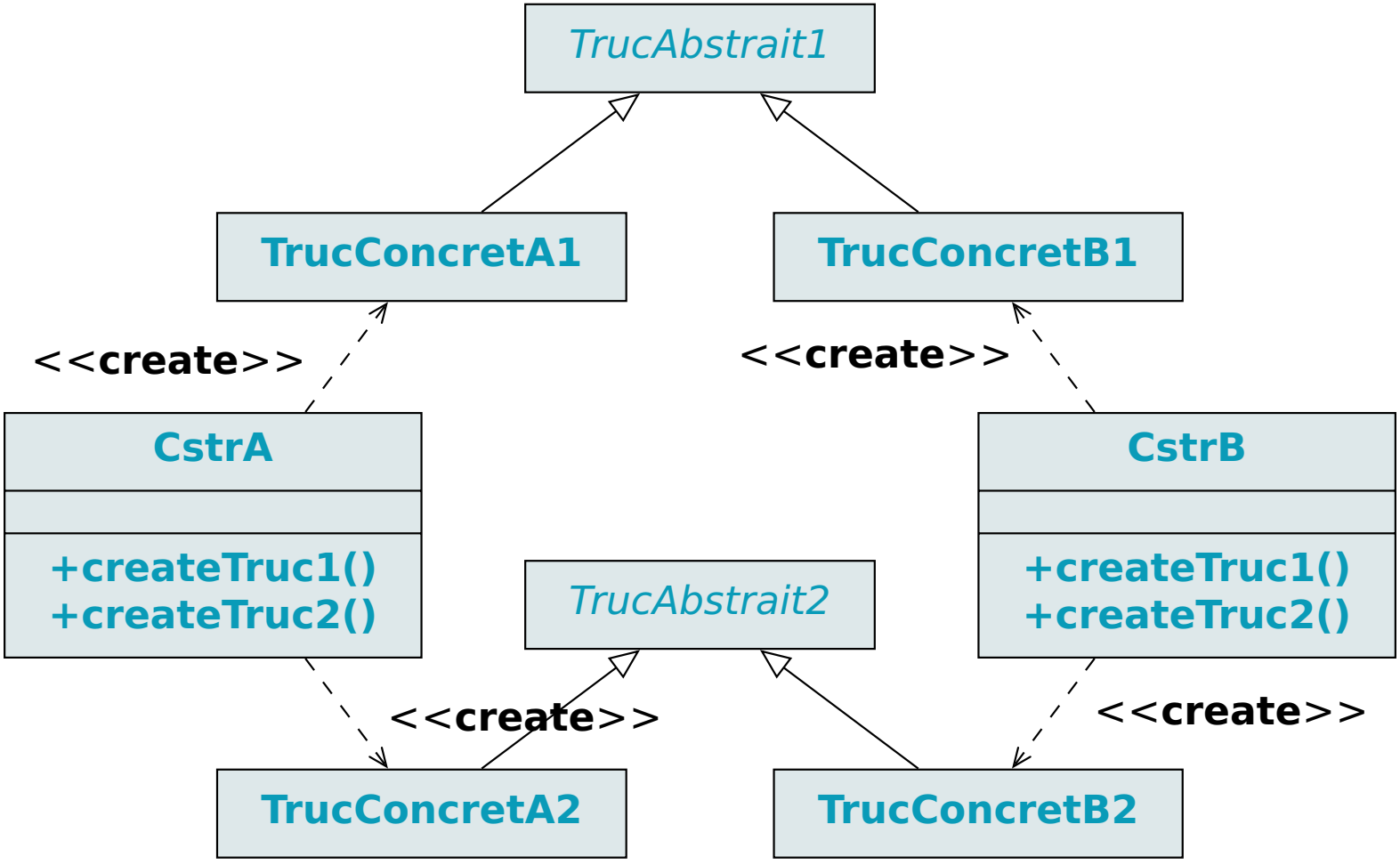
- Des objets dont le type précis n'est pas forcément connu sont créés par des classes constructrices.





Problème

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

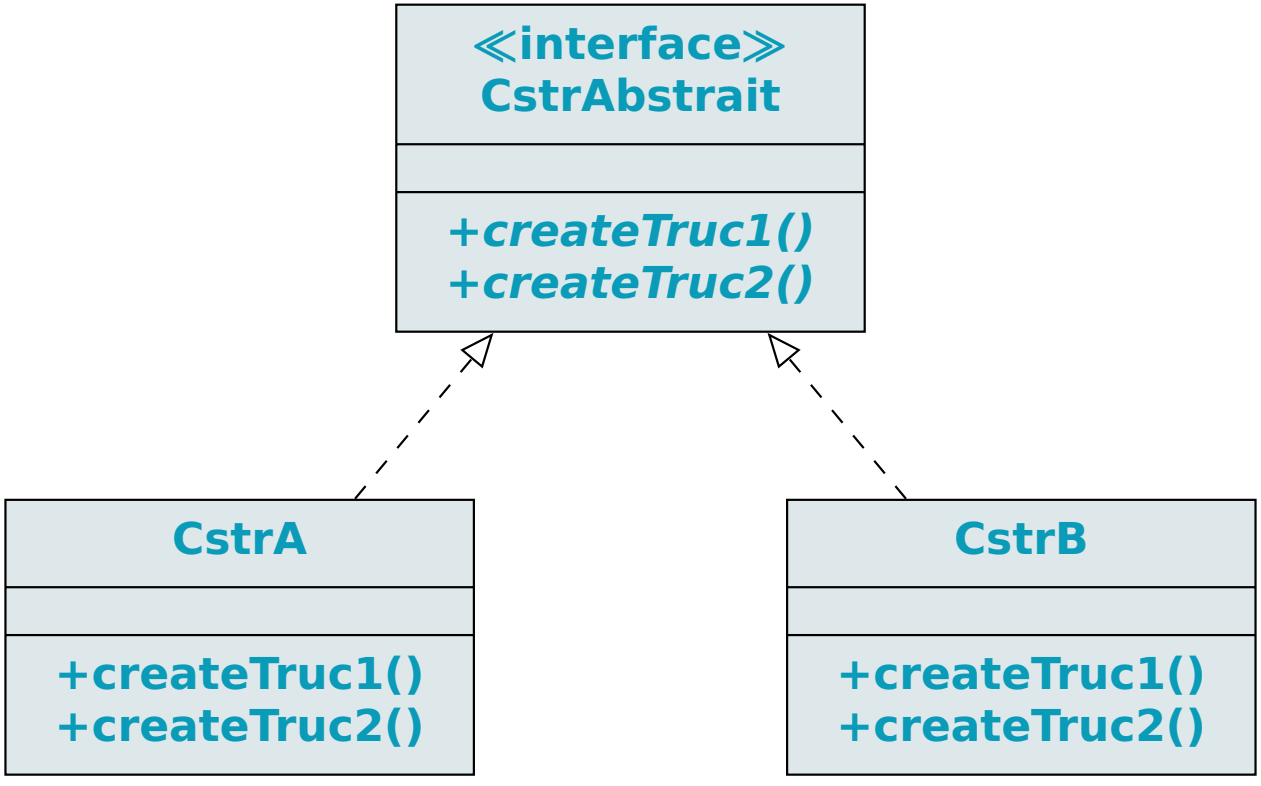


Idée de la solution

- 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.

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.

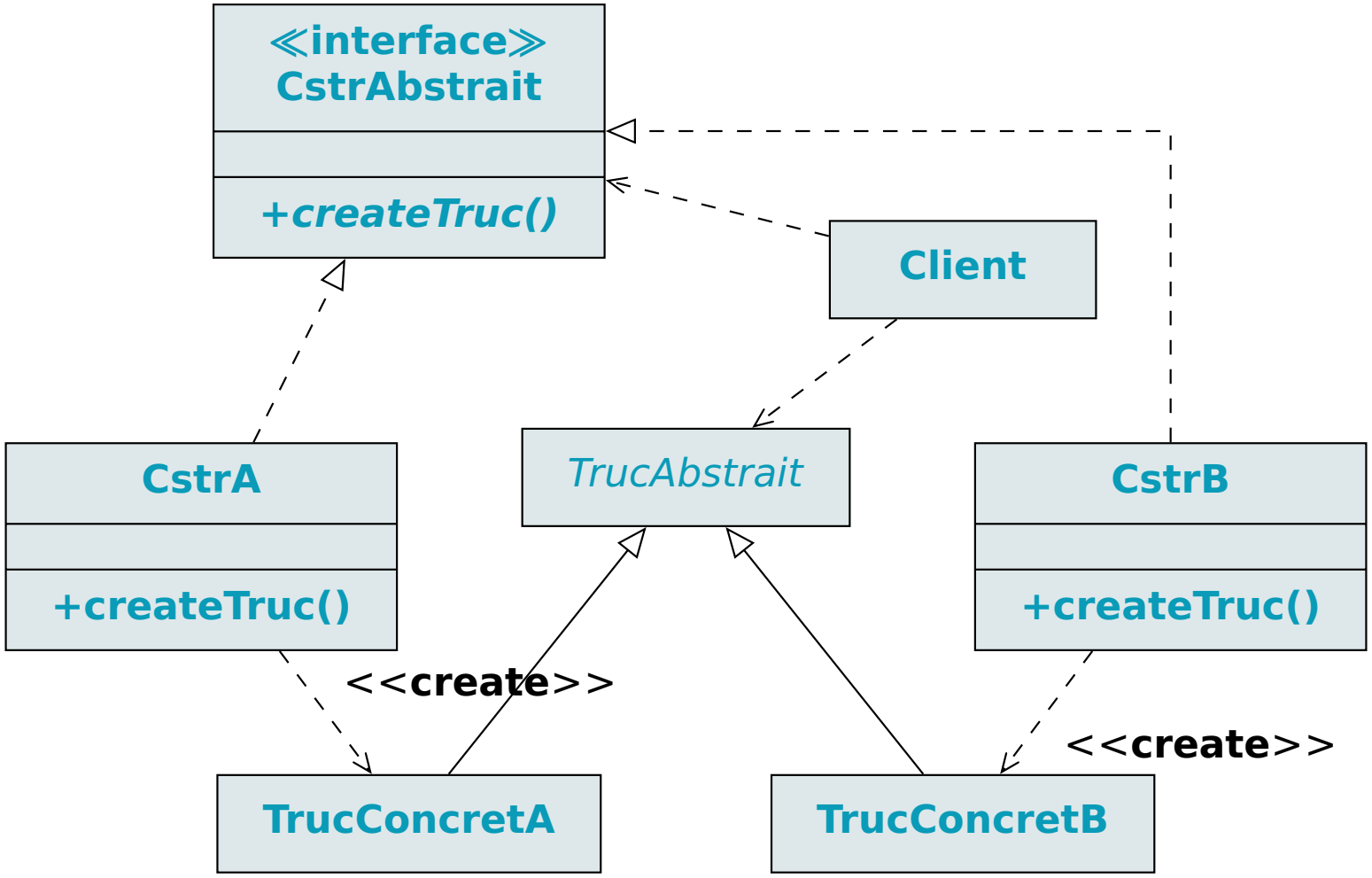


Avantages de cette abstraction

- 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

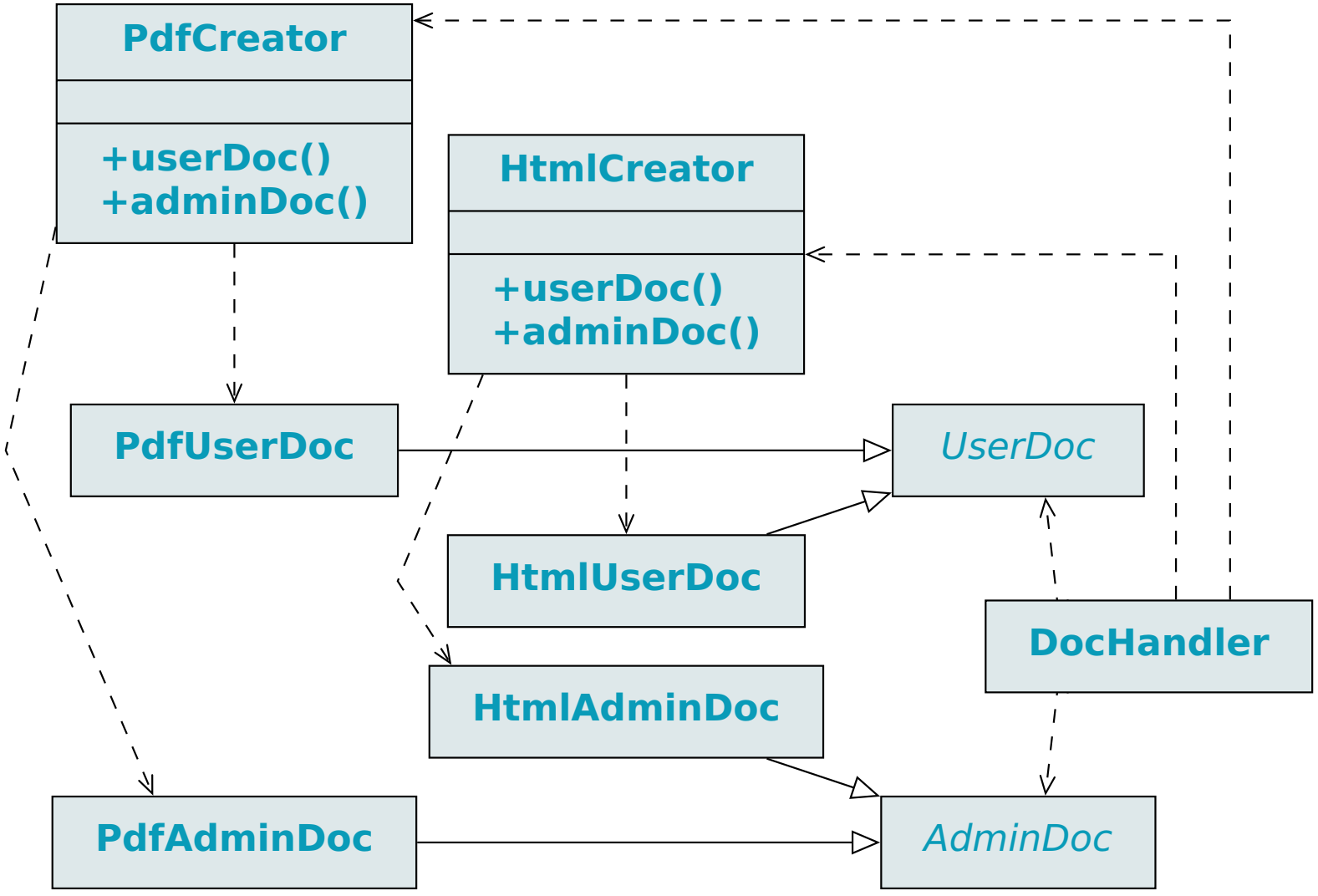
Avec un seul Truc



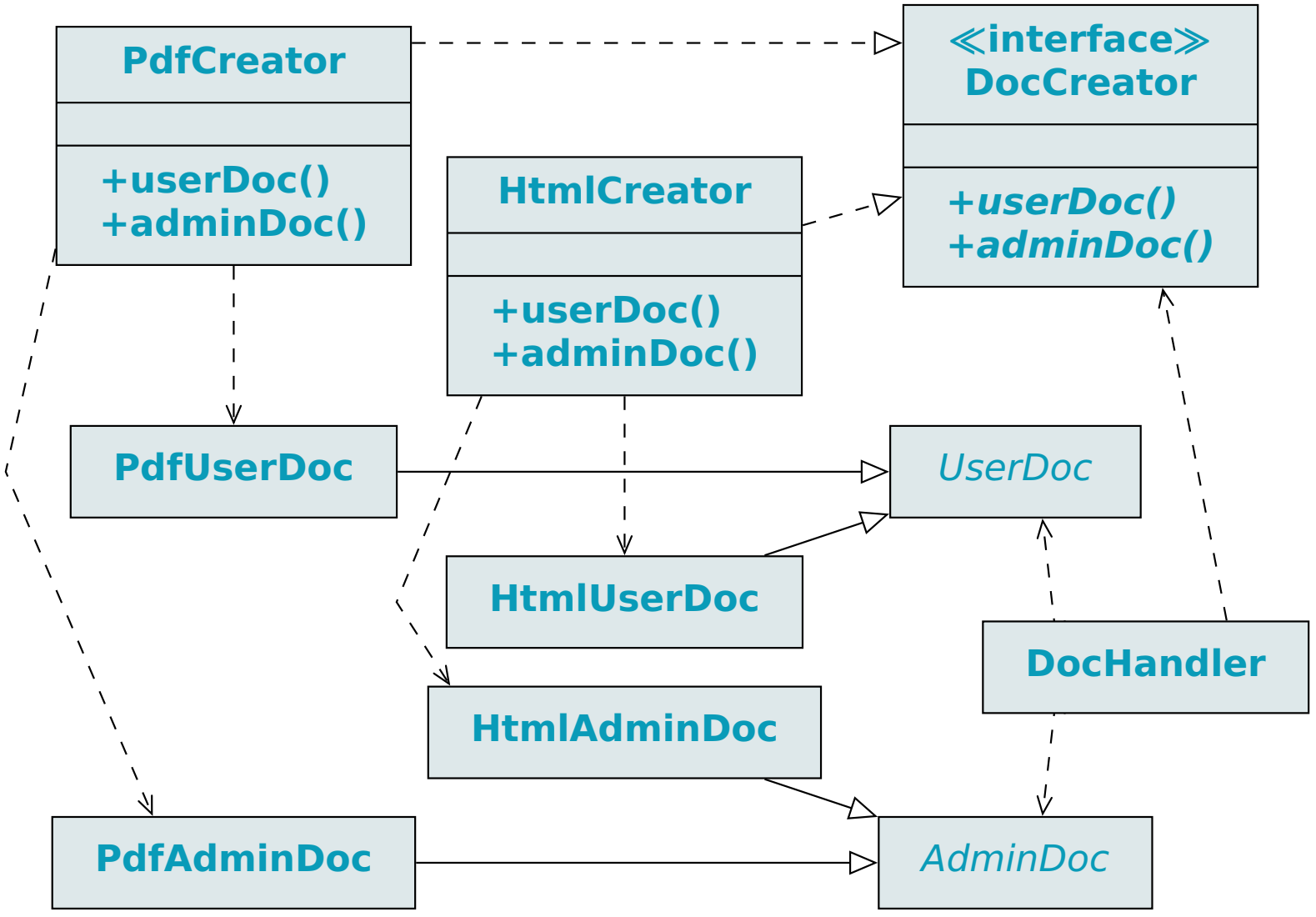
Exemple : le problème

- 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


PdfCreator
+userDoc()
+adminDoc()
HtmlCreator
+userDoc()
+adminDoc()
PdfUserDoc
UserDoc
HtmlUserDoc
DocHandler
HtmlAdminDoc
PdfAdminDoc
AdminDoc

Exemple


PdfCreator
+userDoc()
+adminDoc()
≪interface≫
DocCreator
+userDoc()
+adminDoc()
HtmlCreator
+userDoc()
+adminDoc()
PdfUserDoc
UserDoc
HtmlUserDoc
DocHandler
HtmlAdminDoc
PdfAdminDoc
AdminDoc

Les patrons de structuration

Quelques motifs « de plomberie ». Tous servent à découpler différentes partie d'un développement pour plus de modularité.

Adapter pour lier deux logiciels pas tout à fait compatibles.

Facade pour faire une API simplifiée qui cachent des choses.

Bridge pour découpler deux parties relativement indépendantes.

Façade

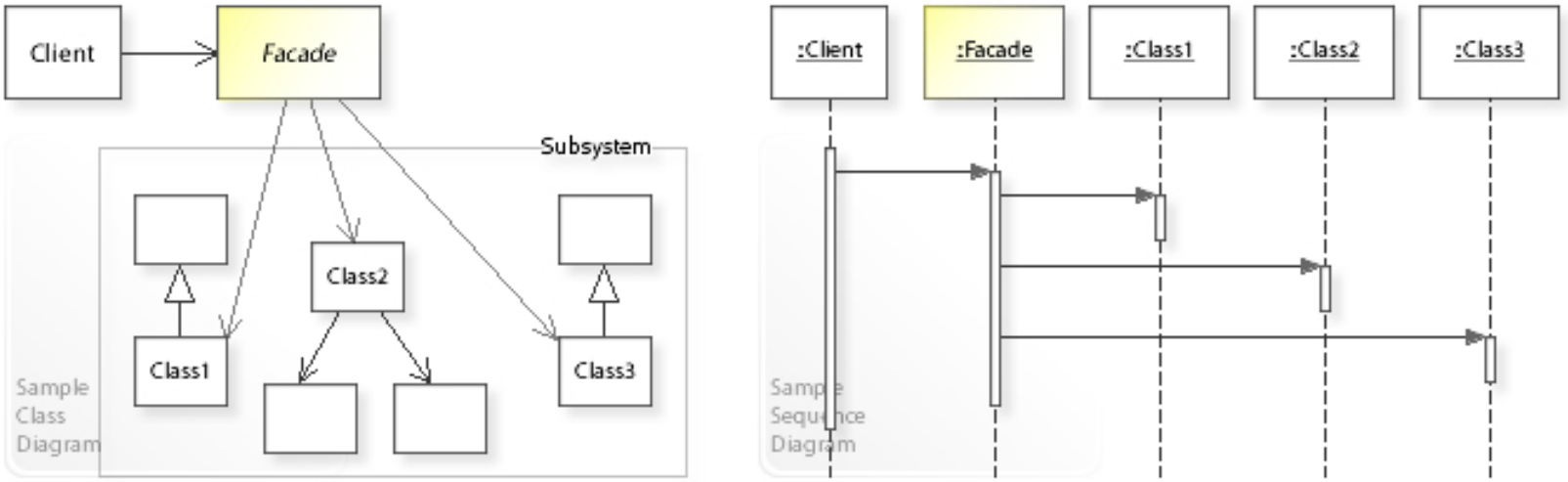
Problème

- 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

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

Facade



By Vanderjoe [CC BY-SA 4.0], from Wikimedia Commons

Facade pour faire une API simplifiée qui cachent des choses

Exemple

```
/* Complex parts */

class CPU {
    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }
}

class HardDrive {
    public byte[] read(long lba, int size) { ... }
}

class Memory {
    public void load(long position, byte[] data) { ... }
}

/* Facade */

class ComputerFacade {
    private CPU processor;
    private Memory ram;
    private HardDrive hd;

    public ComputerFacade() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }

    public void start() {
        processor.freeze();
        ram.load(BOs.jpgOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
        processor.jump(BOOT_ADDRESS);
        processor.execute();
    }
}

/* Client */

class You {
    public static void main(String[] args) {
        ComputerFacade computer = new ComputerFacade();
        computer.start();
    }
}
```

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.

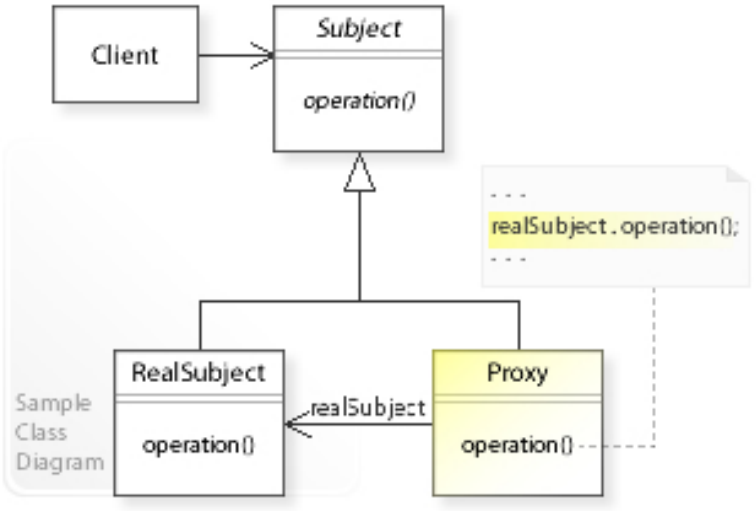
Exemple 1

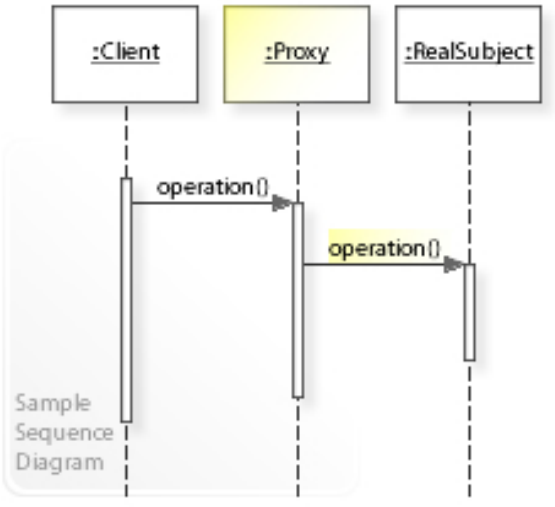
C'est le cas des `Query` vers les bases de donnée en Java : la requête n'est vraiment exécutée que sur demande expresse.

Solution

Idée

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





exemple

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

Autre Exemple

L’exemple qui suit est tiré de :

`https://github.com/iluwatar/java-design-patterns/tree/master/proxy`

Il s’agit de limiter le nombre de magiciens qui rentre dans la tour.

Tour d'ivoire

```java
public interface WizardTower {

  void enter(Wizard wizard);
}

public class IvoryTower implements WizardTower {

  private static final Logger LOGGER = LoggerFactory.getLogger(IvoryTo

  public void enter(Wizard wizard) {
    LOGGER.info("{} enters the tower.", wizard);
  }

}
```

Magicien

```
public class Wizard {

  private final String name;

  public Wizard(String name) {
    this.name = name;
  }

  @Override
  public String toString() {
    return name;
  }
}
```

Le Proxy

```
public class WizardTowerProxy implements WizardTower {

  private static final Logger LOGGER = LoggerFactory.getLogger(WizardTowerProx

  private static final int NUM_WIZARDS_ALLOWED = 3;

  private int numWizards;

  private final WizardTower tower;

  public WizardTowerProxy(WizardTower tower) {
    this.tower = tower;
  }

  @Override
  public void enter(Wizard wizard) {
    if (numWizards < NUM_WIZARDS_ALLOWED) {
      tower.enter(wizard);
      numWizards++;
    } else {
      LOGGER.info("{} is not allowed to enter!", wizard);
    }
  }
}
```

Exemple

```
WizardTowerProxy proxy = new WizardTowerProxy(new IvoryTower());
proxy.enter(new Wizard("Red wizard")); // Red wizard enters the tower.
proxy.enter(new Wizard("White wizard")); // White wizard enters the tower.
proxy.enter(new Wizard("Black wizard")); // Black wizard enters the tower.
proxy.enter(new Wizard("Green wizard")); // Green wizard is not allowed to
proxy.enter(new Wizard("Brown wizard")); // Brown wizard is not allowed to
```

Récapitulatif

Catégorie	Patrons de conception
Construction	Factory, AbstractFactory, Singleton
Structuration	Composite, Adapter, Bridge, Facade
Comportement	Iterator, Observer, Null Object
Architecture	MVC, DAO