

# DESIGN PATTERNS

printable

id	titre	acteur	besoin
1	Design patterns	développeur	comprendre qu'il existe des patrons de conception prêts à l'emploi
2	Patterns: Strategy	développeur	spécialiser pour mon besoin un algorithme ou objet conçu à

# DESIGN PATTERNS

ID: #1

En tant que développeur

Je veux comprendre qu'il existe des patrons de conception prêts à l'emploi

Afin de bénéficier des solutions trouvées par d'autres à des problèmes récurrents.

Priorité: Haute

Valeur: 1000

# ⊕ CRITÈRES D'ACCEPTATION

ID: #1

- Connaitre l'origine des Design patterns
- Identifier les patrons de conception comme un vocabulaire
- Identifier les liens entre les patrons de conception et OOP

# POURQUOI?

- Quelqu'un a déjà réglé nos problèmes.
  - L'OOP promet la réutilisation de code, les patterns la réutilisation d'expérience.
  - Connaître des patterns en OOP est comparable à connaître des exemples d'algorithmes

# C'EST TOUT?

- Définir un vocabulaire commun.
- Réfléchir au niveau au dessus.
- ...

# RAPPELS OBJET

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle



# STRATEGY



# PATTERNS: STRATEGY

ID: #2

En tant que développeur

Je veux spécialiser pour mon besoin un algorithme ou objet sans avoir à créer une classe par usage.

Afin de pouvoir bénéficier d'une plus grande souplesse qu'avec l'héritage, de réutiliser mon code et de décider à l'exécution du comportement des objets.

Priorité: Haute

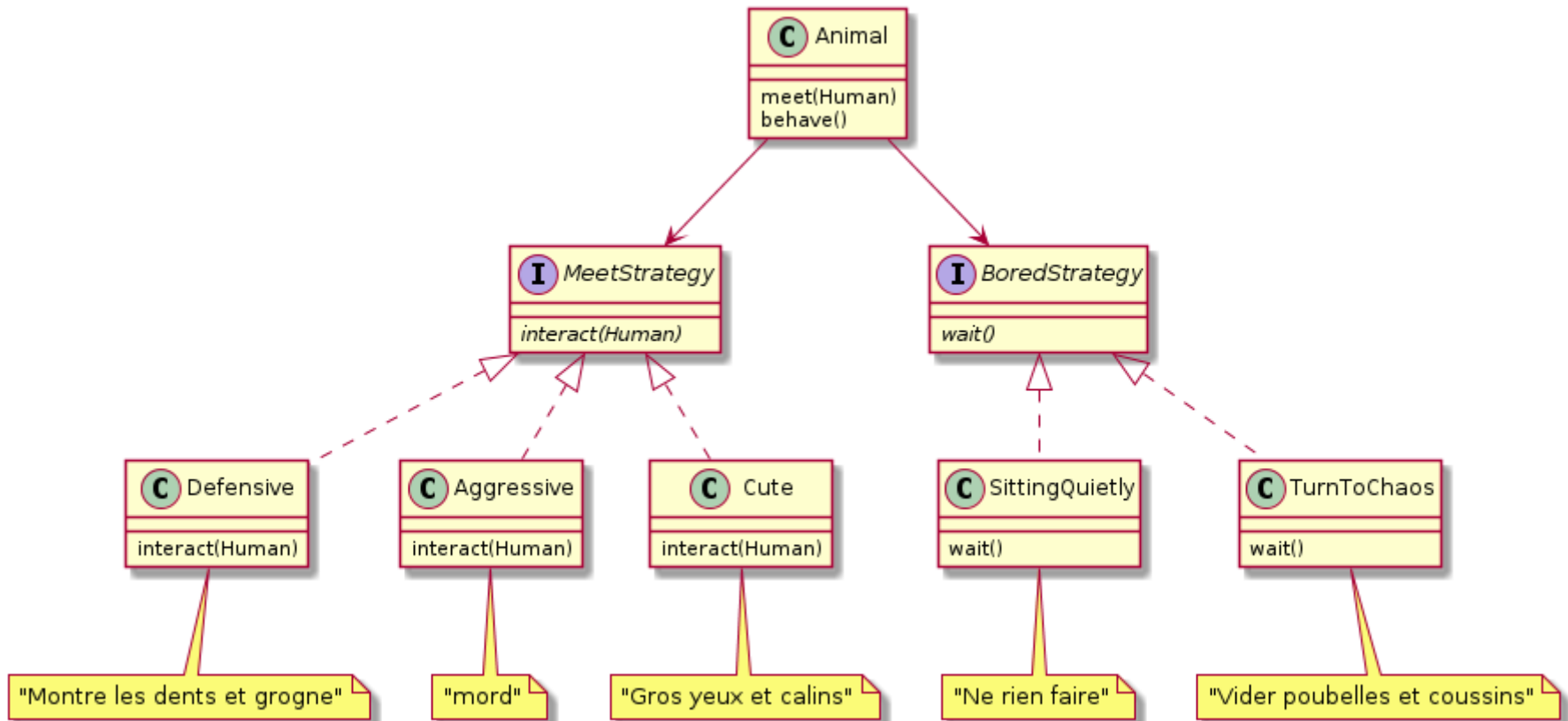
Valeur: 1000

# CRITÈRES D'ACCEPTATION

ID: #2

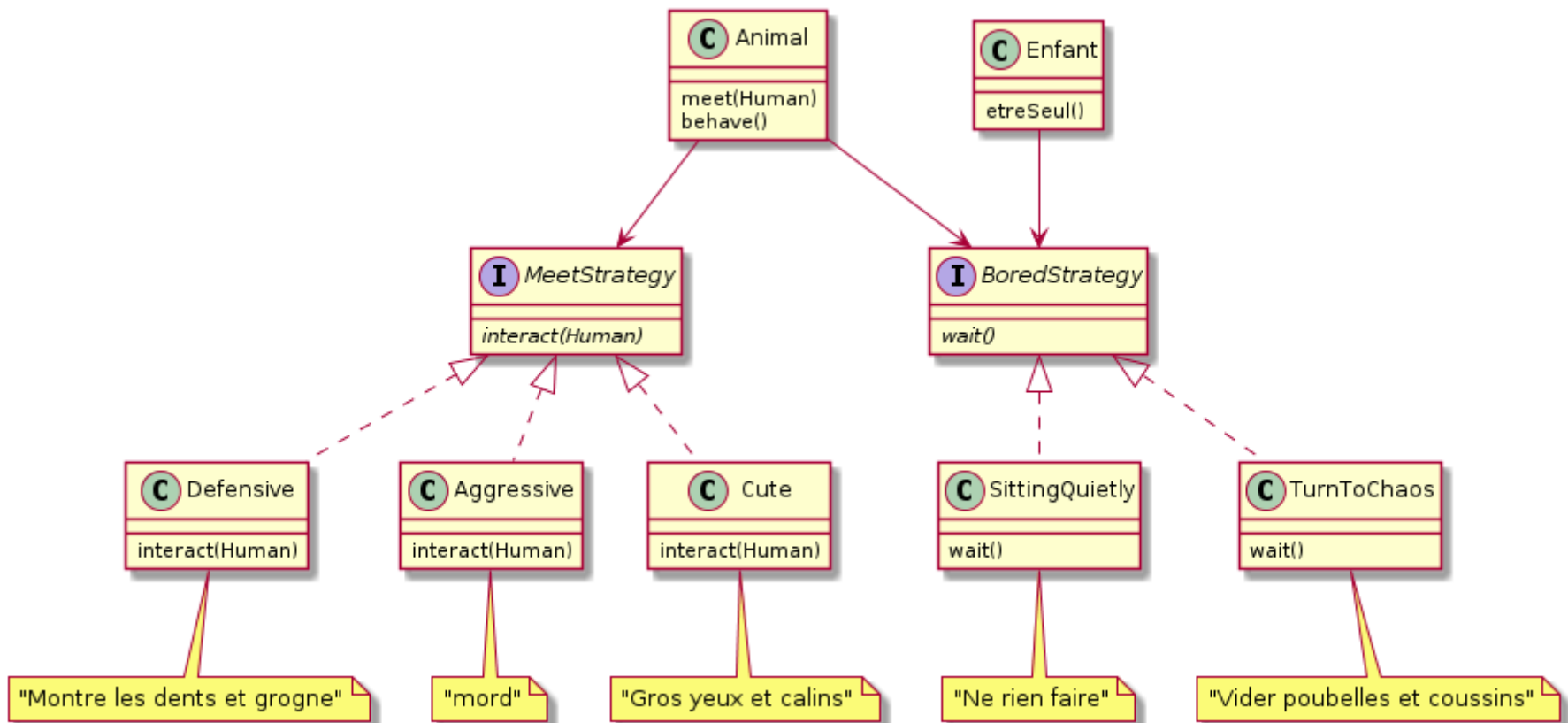
- Être capable de décrire le fonctionnement du pattern Strategy
- Trouver le pattern dans une classe ou plus du JDK
- Implémenter un algorithme en exploitant le pattern.

# UML



```
Animal chienDeGarde = Animal(Aggressive(), SitQuietly());
Animal deSalon= Animal(Cute(), SitQuietly());
Animal maltraite = Animal(Defensive(), TurnToChaos());
```

# RÉUTILISABLE?



```
robot.act(Aggressive())
```

# **EXAMPLES:**

- `Collections.sort(List<T>, Comparator<? super T>)`
- `Collections.max(Collection<? extends T>, Comparator<? super T>)`
- `ThreadPoolExecutor(...)  
ThreadFactory,  
RejectedExecutionHandler)`



# ADAPTER+FACADE

# PATTERNS: ADAPTER

ID: #3

En tant que développeur

Je veux être capable d'utiliser un objet du type A  
dans le cadre d'un outil attendant un B

Afin de continuer à utiliser mon code pour des  
objets d'un type qu'il ne connaît pas.

Priorité: Moyenne

Valeur: 1000

# 🎯 CRITÈRES D'ACCEPTATION.

ID: #3

- Être capable de décrire le fonctionnement du pattern Adapter
- Utiliser un object "legacy" produisant une `Enumeration` dans un `"for( X x: xs) "`
- Connaitre la différence entre un Class Adapter et un Object Adapter



# PATTERNS: FACADE

ID: #4

En tant que développeur

Je veux être capable d'utiliser un objet qui s'occupera de déléguer mes demandes à un groupe d'objets plus complexes

Afin de réduire la complexité d'usage et la dépendance de mon code aux composants d'un système complexe.

Priorité: Basse

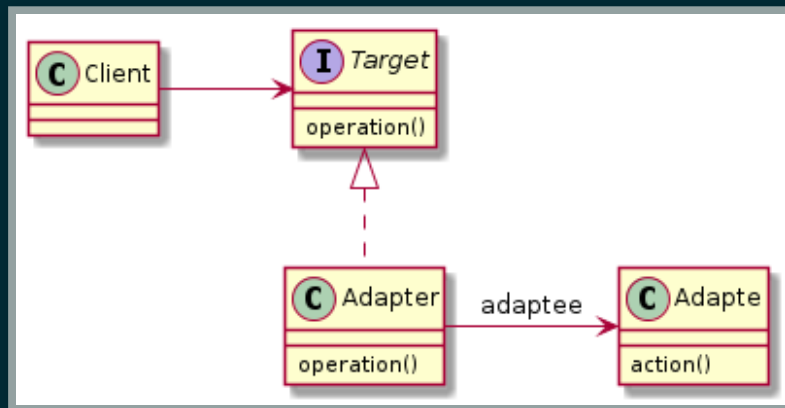
Valeur: 200

# 🎯 CRITÈRES D'ACCEPTATION.

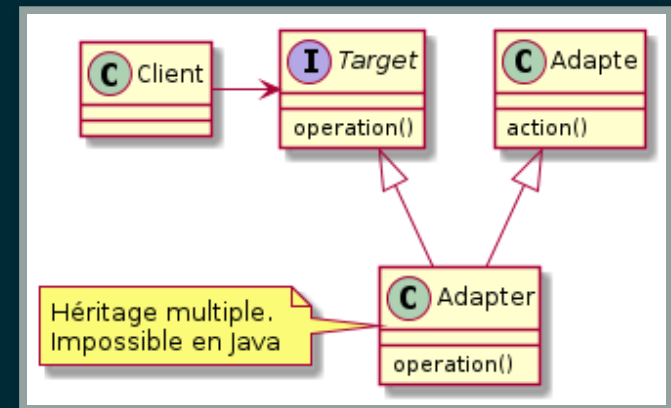
ID: #4

- Être capable de décrire le fonctionnement du pattern Facade
- Connaitre la différence entre un Class Adapter et un Object Adapter
- Masquer un système complexe derrière une facade.

# UML

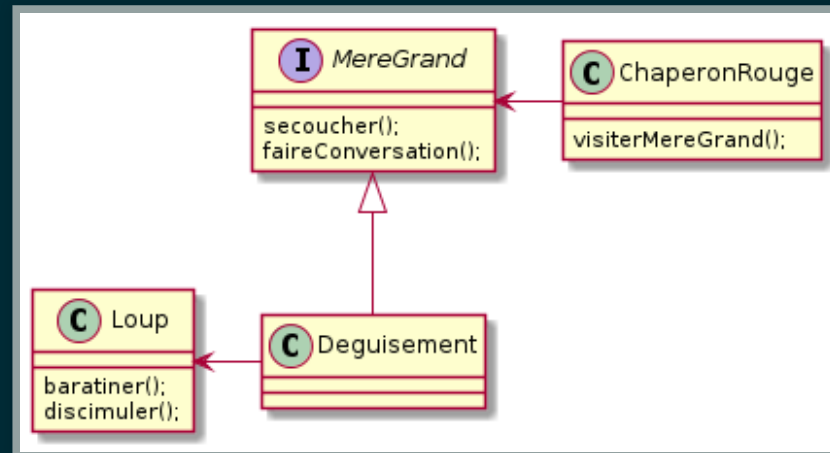


Object adapter



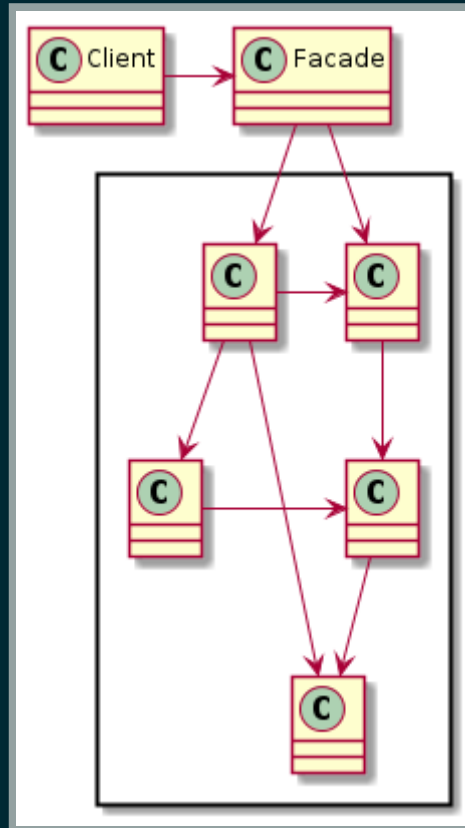
Class adapter

# UML



```
MereGrand meregrand = Deguisement(Loup());
meregrand.seCoucher();
victime = ChaperonRouge(meregrand);
victime.visiterMereGrand();
```

# FACADE



# **EXAMPLES:**

Adapter:

- `InputStreamReader(InputStream)`
- `Arrays.asList()`

Facade:

- Guichet d'une administration idéale.

# OBSERVER

A la base d'un autre pattern Pub/sub. (ou bus de messages)

# PATTERNS: OBSERVER

ID: #5

En tant que développeur

Je veux disposer d'un moyen d'émettre des événement/valeurs vers des éléments arbitraires dont je ne souhaite pas dépendre.

Afin de ne pas avoir à modifier ma source d'événements lorsqu'un nouvel usage se présente.

Priorité: Moyenne

Valeur: 1000

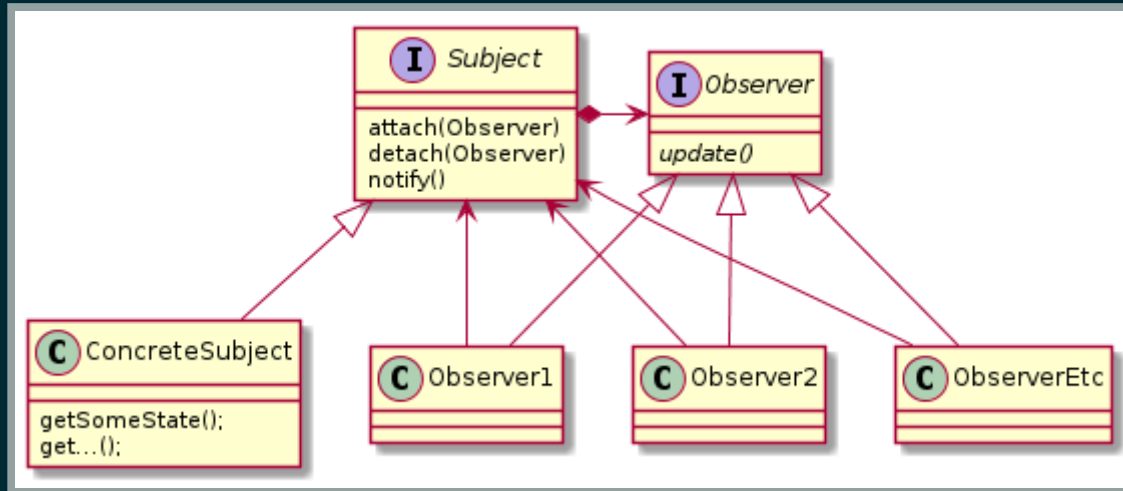


# CRITÈRES D'ACCEPTATION

ID: #5

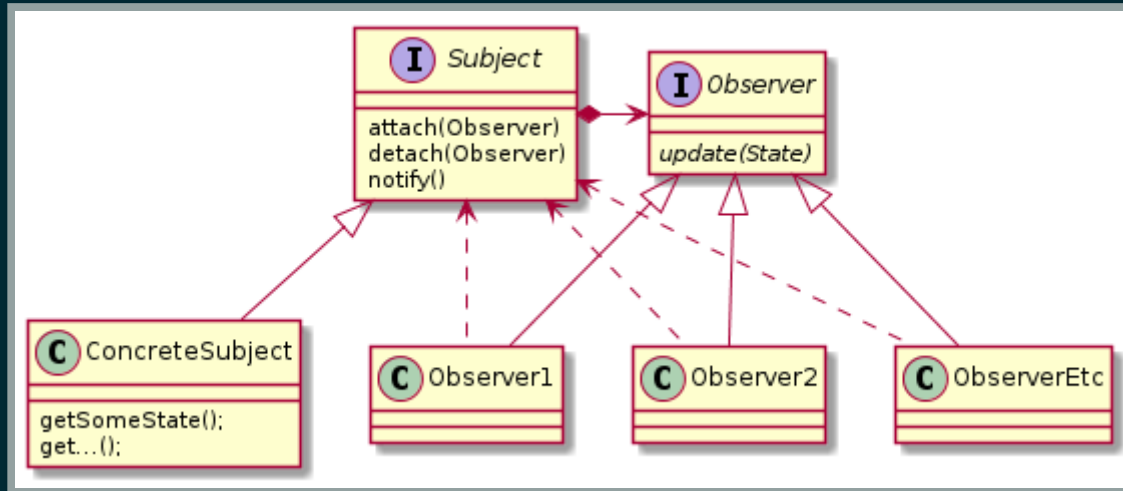
- Décrire le fonctionnement du pattern Observer
- Identifier des usages courants du pattern
- Utiliser le pattern pour fournir des statistiques sur une source de données.
- Différencier le mode push du mode pull
- Identifier les défauts de l'implémentation du pattern dans le JDK

# PULL



```
Observer1{
    update(){println("hey:"+((cast)subject).getSomething());}
}
Observer2{
    update(){println("ooooooooohhhh "+((cast)subject).getSometh
}
Observer1(subject); Observer2(subject);
subject.setSomething(3);
// => hey: 3
// => oooooooooohhhh 3
```

# PUSH



```
Observer1{
    update(val){println("hey:"+val);}
}
Observer2{
    update(val){println("ooooooohhhh "+val);}
}
Observer1(subject); Observer2(subject);
subject.setSomething(3);
// => hey: 3
// => ooooooohhhh 3
```

# **EXAMPLES:**

- `java.util.Observable`
- La plupart des interfaces graphiques ( `onclick`, `onkeypressed` )...

# JAVA.UTIL.OBSERVABLE

Java fournit une classe Observable depuis le JDK1. Elle n'est pas sans problèmes:

- Il faut étendre la classe: pas de choix dans l'héritage
- Certaines méthodes sont `protected`. Pour s'en servir il faut étendre
- Unique implémentation. Que faire si on ne souhaite pas utiliser un Vector ou faire du multithreading?



# DECORATOR

# PATTERNS: DECORATOR

ID: #6

En tant que développeur

Je veux pouvoir ajouter des comportements à un objet existant avant de le fournir à un traitement

Afin de définir le comportement d'une méthode à la carte sans définir une hiérarchie complexe de classes avec beaucoup de duplications où il faudra intervenir à chaque nouvelle méthode.

Priorité: Moyenne

Valeur: 800

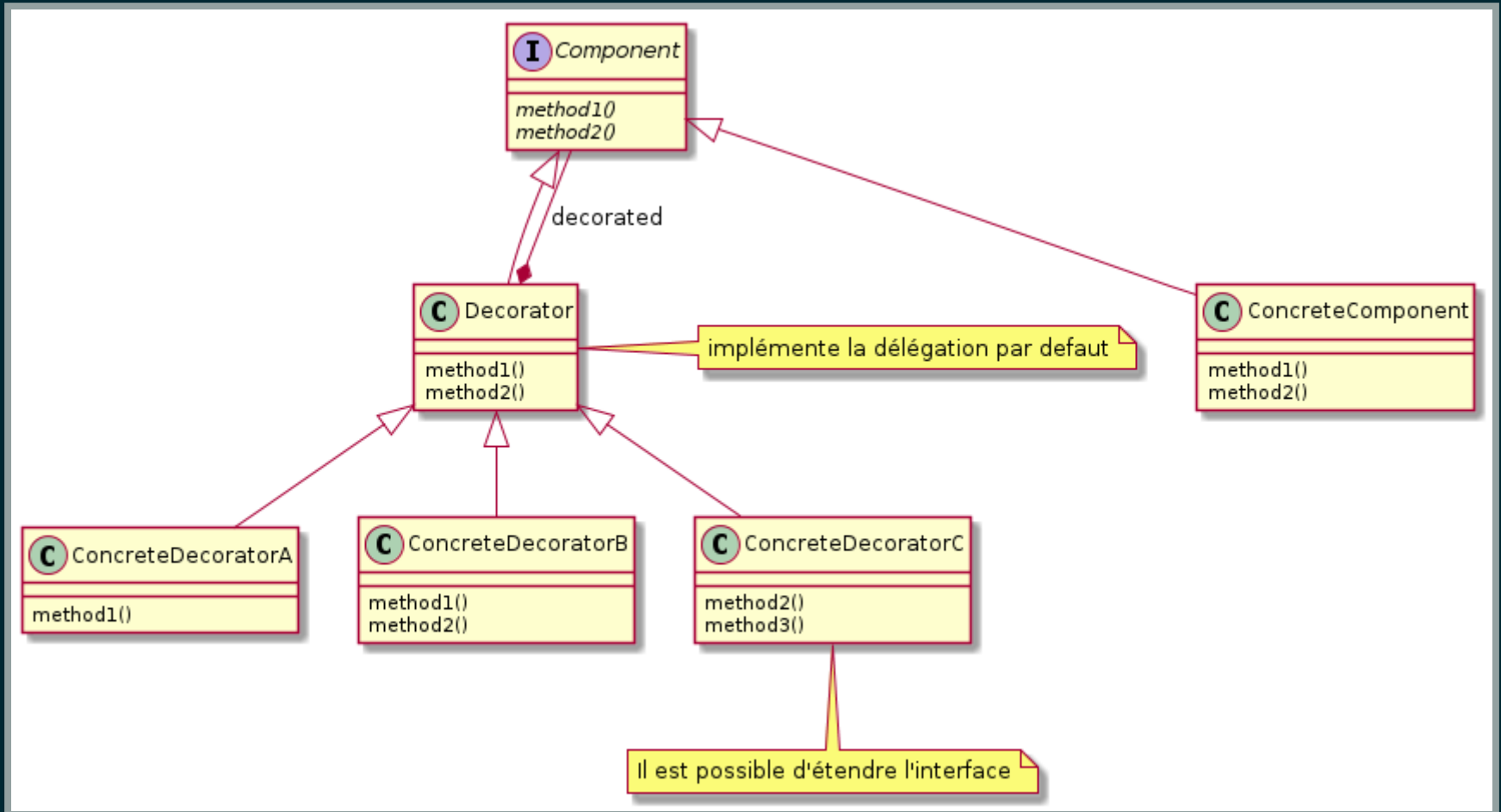
# CRITÈRES D'ACCEPTATION

ID: #6

- Décrire le pattern Decorator
- Connaître un exemple dans le JDK
- Utiliser le pattern dans un cas d'exemple



# UML



```
myComponent= ConcreteDecoratorC(ConcreteDecoratorA(ConcreteCompon
myComponent.method2( )
```



# **EXAMPLES:**

- `Collections.checkedXXX()`
- `java.io.*InputStream`



**FACTORY**

# PATTERNS: FACTORY

ID: #7

En tant que développeur  
Je veux déléguer l'instanciation d'objets  
Afin de me concentrer sur mon traitement,  
confiant que j'obtiendrai une instance correcte et  
cohérente avec mon traitement.

Priorité: Haute

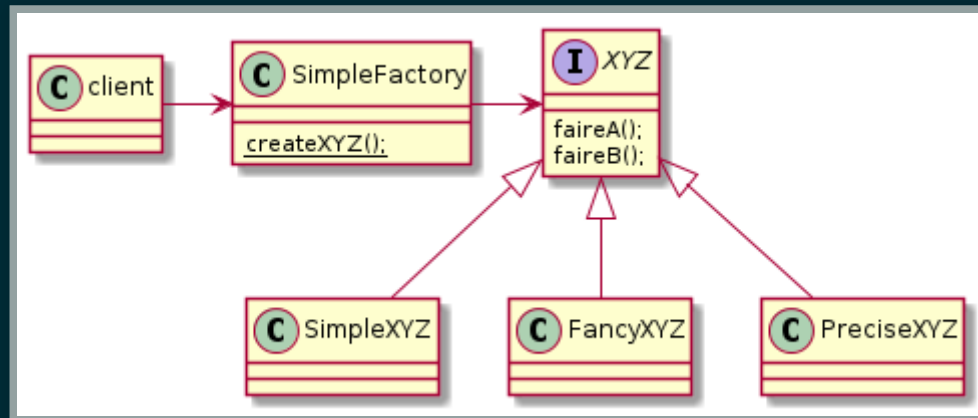
Valeur: 800

# CRITÈRES D'ACCEPTATION

ID: #7

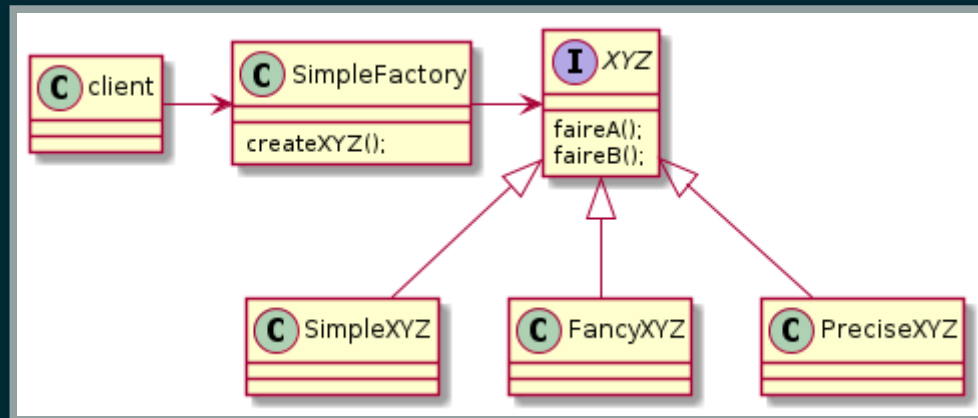
- Savoir décrire les patterns Abstract factory et Factory method.
- Utiliser la factoryMethod pour présenter le résultat d'un traitement dans un afficheur Swing ou Console
- Utiliser l'Abstract Factory pour créer un affichage graphique ou une interface console sans connaissance de l'objet qui va l'utiliser.

# LE PSEUDO-PATTERN STATIC FACTORY



- Le plus simple pour encapsuler l'instanciation hors de notre classe.
- Méthode `static`. Pas de hiérarchie, pas de souplesse.

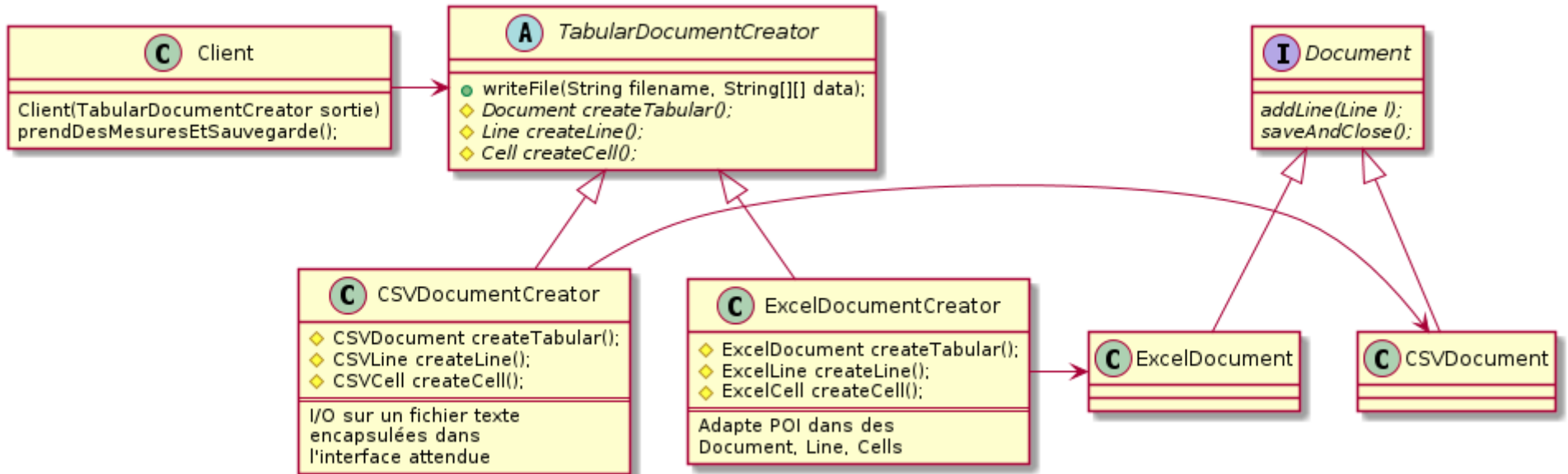
# LE PSEUDO-PATTERN SIMPLEFACTORY



- Sortir la création de l'objet de notre classe et pouvoir la substituer.



# FACTORY METHOD

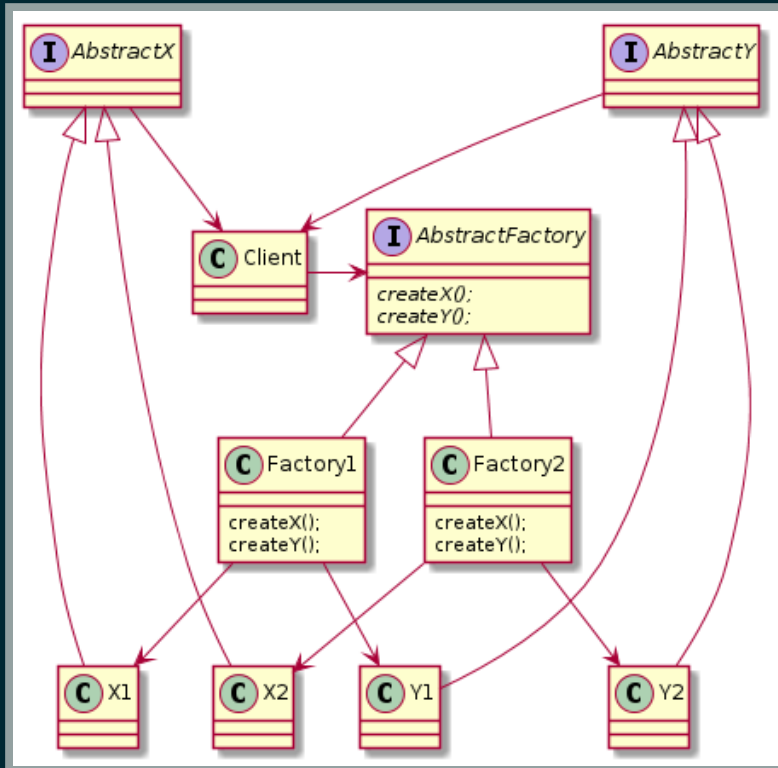


But: Laisser le choix de l'implémentation pour plus tard (runtime)

```
//dans new Client(CSVDocumentCreator).prendDesMesuresEtSauvegard  
table = factory.writeFile("resultats", [[1,2,3],[2,1,4]])
```

```
//Dans factory.writefile  
new File(filename+".csv");
```

# ABSTRACT FACTORY



- Produit des familles d'objets cohérentes et pousse ses clients à utiliser cette cohérence.

Factory	Quand?	Pattern officiel?
static	"Single responsibility principle" quand l'instanciation est compliquée.	Non
simple	static mais avec la possibilité de substituer. (en TDD, aide pour les dates)	Non

Factory	Quand?	Pattern officiel?
method	Déléguer le choix de la classe concrète utilisée dans un traitement aux sous classes (donc au code qui instancie)	Oui
abstract	Substituer des grappes de classes cohérentes	Oui

## **EXAMPLES:**

- `java.util.Calendar#getInstance();`  
//static factory
- `javax.xml.parsers.DocumentBuilderFactory`  
`#newInstance()`//static factory d'une  
abstract factory



# CATÉGORIES ET FORMALISME

# PATTERNS: CATÉGORIES ET ID: #17 FORMALISME

En tant que P.O.

Je veux Définir ce qu'est un pattern et grouper ceux que je connais déjà en catégories

Afin de fournir une structure et identifier des besoins généraux.

Priorité: Haute

Valeur: 1000



# CRITÈRES D'ACCEPTATION

ID: #

- Identifier et décrire 2 façons de catégoriser les patterns
- Décrire au moins un pattern avec son formalisme



**SINGLETON**

# PATTERNS: SINGLETON

ID: #8

En tant que développeur

Je veux n'instancier qu'un objet d'une classe donnée par exécution et au dernier moment.

Afin de conserver des ressources, éviter des glitches (tels que 2 fenêtres modales) ou offrir un service commun.

Priorité: Haute

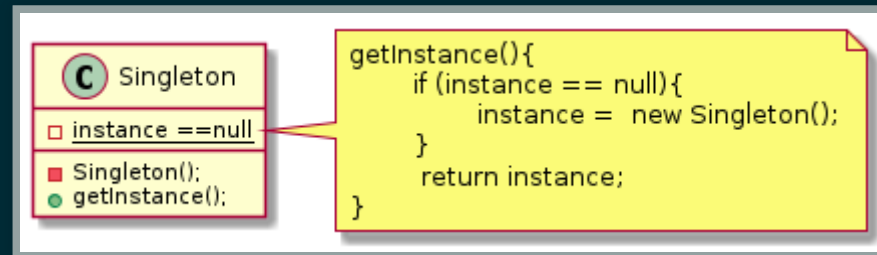
Valeur: 600

# CRITÈRES D'ACCEPTATION

ID: #8

- Connaître le principe du singleton
- Connaître la version sans concurrence du singleton
- Connaître les impacts du multithreading sur un singleton.

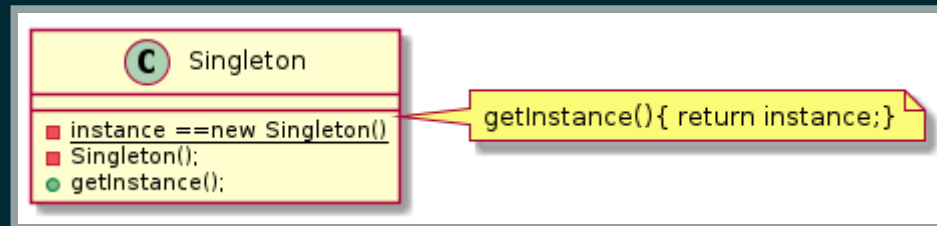
# LA VERSION SIMPLE



S'il n'y a pas multithreading on s'arrête là.

S'il y a multithreading sans recherche de performance:  
`synchronized getInstance()` et c'est tout.

# LA VERSION EAGER



Si multithreads, problèmes de performances et certitude d'en avoir besoin.

La JVM initialise la classe *avant* qu'un thread y accède.

# DOUBLE CHECK LOCKING

```
private volatile static Singleton instance;  
public static Singleton getInstance(){  
    if (instance == null){  
        synchronized (Singleton.class){  
            if (instance == null){  
                instance = new Singleton();  
            }  
        }  
    }  
}
```

À partir de java 5 "volatile" a changé de signification.

Notez que la JVM peut :

- inline des fonctions (constructeurs inclus)
- changer l'ordre des instructions

# THE BILL PUGH APPROACH

```
public class BillPughSingleton {  
    private BillPughSingleton() {...}  
  
    private static class SingletonHolder {  
        private static BillPughSingleton instance  
            = new BillPughSingleton();  
    }  
  
    public static BillPughSingleton getInstance() {  
        return SingletonHolder.instance;  
    }  
    ...  
}
```

Subtilité: Java ne charge les classes en mémoire que lorsqu'elles sont référencées. Le premier appel à `getInstance` initialise le helper.



# NOTES & BUGS ÉTRANGES

- Avant Java3, le singleton était collecté lorsque `Singleton.instance` était la seule référence.
- Plus facile de raisonner avec un objet "habituel" qu'avec beaucoup de code statique.
- Un singleton gère sa propre unicité & le service qu'il fournit. C'est une violation du Single-responsibility Principle.

## **EXEMPLES:**

- `java.lang.Runtime#getRuntime()`
- Spring se base sur des singletons dans la conf par défaut



**COMMAND**

# PATTERNS: COMMAND

ID: #9

En tant que développeur

Je veux être capable de déclencher une action sans dépendre des détails de l'opération, de(s) l'objet(s) cibles ou des paramètres.

Afin de planifier une action indépendamment de son déclenchement. Pouvoir les logger, les mettre en attente ou supporter l'annulation (Ctrl+z).

Priorité: Moyenne

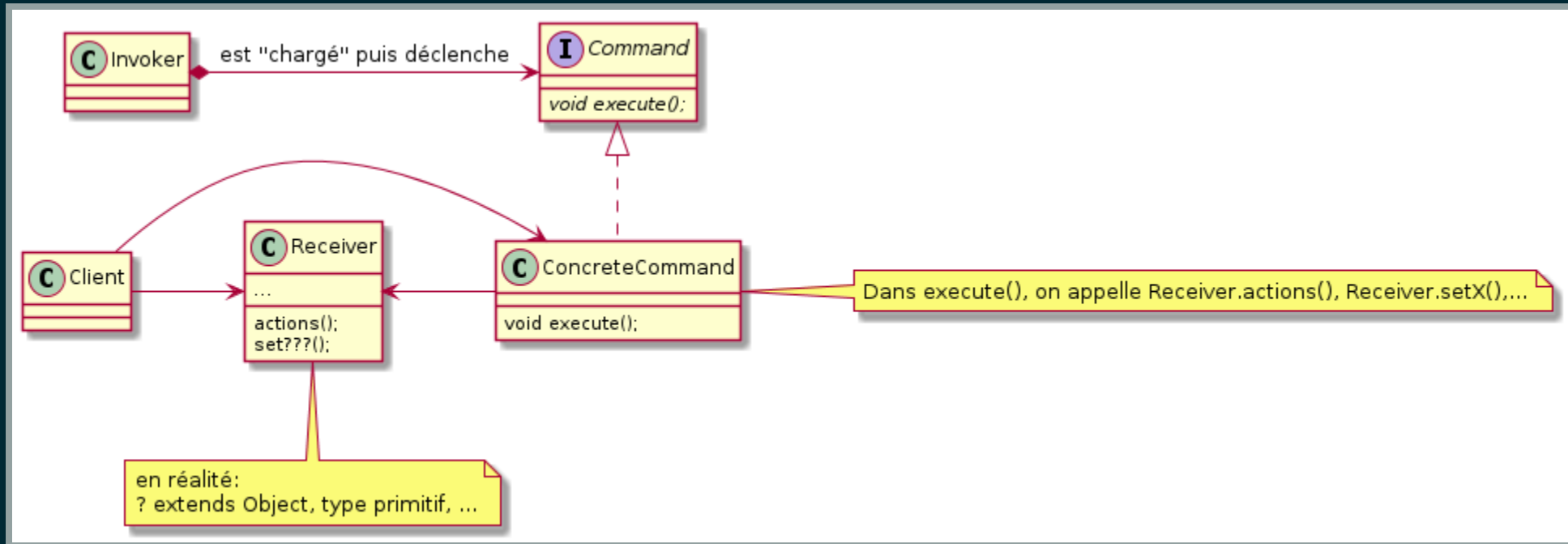
Valeur: 600

# CRITÈRES D'ACCEPTATION

ID: #9

- Décrire le pattern command
- Implémenter le pattern command dans une fausse application domotique.
- Implémenter l'annulation avec état.

# UML



En option :

# LES OPTIONS

## Une méthode `undo()`

Il s'agit des actions "miroir" de `execute()`. Si besoin, `execute()` stocke dans la commande l'état précédent du receiver pour remettre en état (vitesse précédente si la commande est stop, par exemple).

# LES OPTIONS

## Logging

Si les commandes sont sérialisables, on peut les stocker pour faire de la reprise sur crash

## Macros

Il est possible d'écrire une commande composite qui lance les sous commandes en série et dans l'ordre inverse lors du undo.



# **EXEMPLES:**

- `java.lang.Runnable` où l'invoker est `Thread` par exemple.

# **TEMPLATE METHOD**

# PATTERNS: TEMPLATE METHOD ID: #10

En tant que développeur

Je veux spécialiser pour mon besoin un algorithme (ou classe) tout en en préservant la structure.

Afin de réutiliser les éléments invariants et permettre d'étendre la portée de l'algorithme lorsque le besoin se présente.

Priorité: Haute

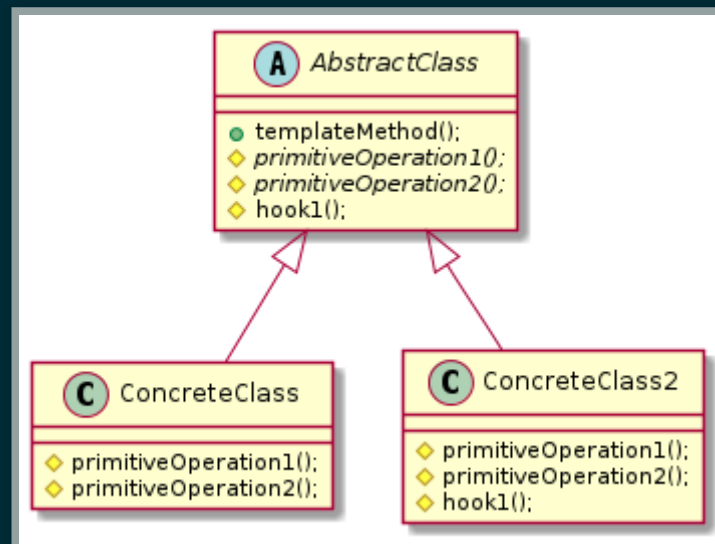
Valeur: 800

# CRITÈRES D'ACCEPTATION

ID: #10

- Décrire le pattern template method
- Décrire ce qu'est un hook dans le contexte
- Utiliser le pattern pour implémenter algorithme simple.
- Puis inclure un hook dans l'algorithme.

# UML



*Ne m'appellez pas, on vous appelle.*

## **EXAMPLE:**

- `java.io.InputStream.read(byte[], int, int)` qui utilise l'implémentation de `read()` et l'appelle en conséquence.

# UN PEU DE TRI

Pattern	Description
Template méthod	Les sous-classes décident de l'implémentation d'une étape.
Strategy	Encapsule des comportements interchangeables et utilisent la délégation.
Factory Method	Les sous-classes décident de quelle classe concrète instancier.

# ITERATOR



# PATTERNS: ITERATOR

ID: #11

En tant que développeur

Je veux parcourir un agrégat de façon séquentielle sans être impacté par la façon dont il est implémenté

Afin de laisser la possibilité de changer l'implémentation de l'agrégat ou permettre à mon algorithme de fonctionner sur plusieurs type d'agrégats.

Priorité: Moyenne

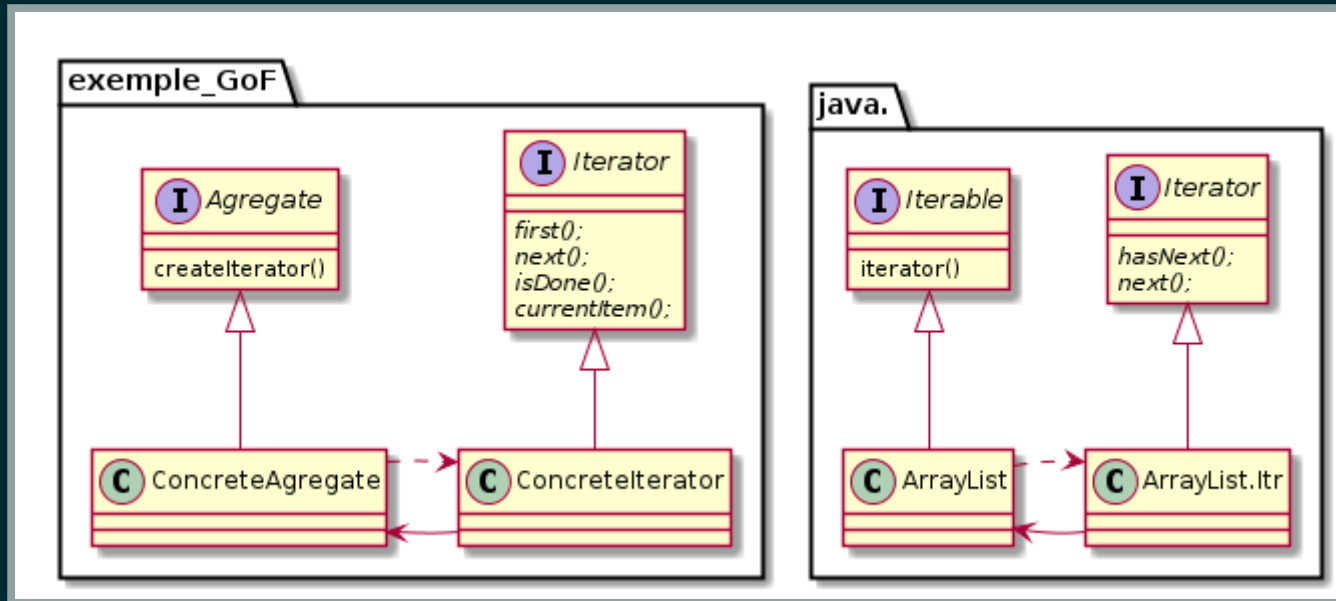
Valeur: 800

# CRITÈRES D'ACCEPTATION

ID: #11

- Décrire le pattern iterator
- Ecrire son propre aggrégat et utiliser le pattern pour le parcourir.

# UML



```
Iterator iter = list.iterator();
while (iter.hasNext()) {
    System.out.print(iter.next());
}
```

# **EXAMPLES:**

- `java.util.Iterator`
- `java.util.Enumeration`



# NULL OBJECT

# PATTERNS: NULL OBJECT

ID: #12

En tant que développeur

Je veux , lorsque je dois fournir un objet et que mon état fait que cet objet n'a rien à fournir, éviter à mon client de tester pour null et continuer à utiliser son algorithme.

Afin de simplifier l'écriture des clients en garantissant l'interface du service rendu.

Priorité: Basse

Valeur: 200

# CRITÈRES D'ACCEPTATION

ID: #12

- Démontrer l'utilisation d'un Null Object.

# **EXEMPLES:**

- Un itérateur de collection vide dont `next() == null` et `hasNext() == false`
- `new Add(0)`, `new Times(1)` dans un programme de calcul





# COMPOSITE

# PATTERNS: COMPOSITE

ID: #13

En tant que développeur

Je veux manipuler ou consulter les propriétés d'un  
groupe d'objet ou un objet individuel  
indifféremment

Priorité: Moyenne

Valeur: 400Valeur:

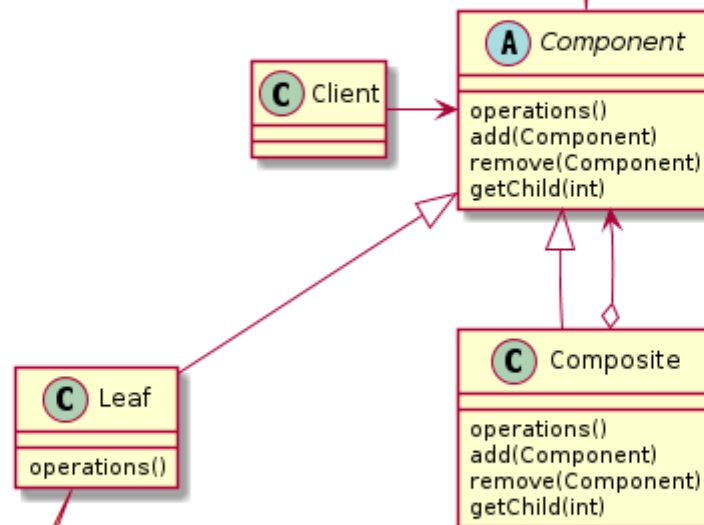
# ⊕ CRITÈRES D'ACCEPTATION

ID: #13

- Être capable de décrire le fonctionnement du pattern Composite
- Utiliser le pattern pour représenter une hiérarchie de noeuds HTML (HTML, head, title, body, h1, p, ul, li) et implémenter `textToUppercase`, `getText` et `print`

# UML

Des corps par défaut de méthodes  
afin d'éviter de surcharger le code de leaf ou Composite.  
Beaucoup lanceront RuntimeException ou retourneront des Null Object



Les opérations add, remove et getChild n'ont pas de sens.

Si une operation n'a pas de sens.

# **EXEMPLES:**

- Les composants graphiques dans la plupart des langages



# PATTERNS: STATE

ID: #14

En tant que développeur

Je veux faire changer le comportement d'un objet  
en fonction de son état interne

Priorité: Haute

Valeur: 600

# CRITÈRES D'ACCEPTATION

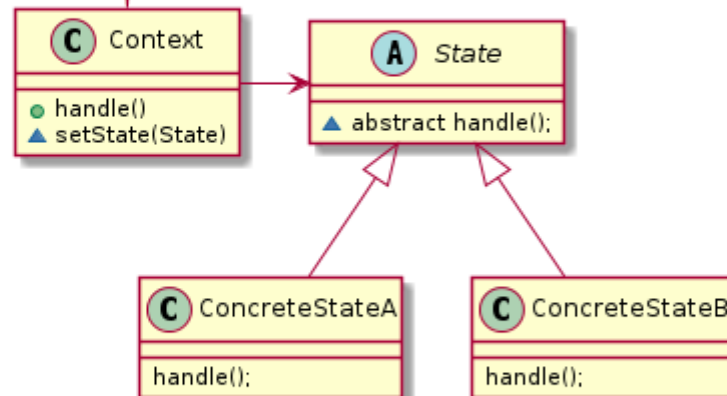
ID: #14

- Être capable de décrire le fonctionnement du pattern State
- Comprendre pourquoi la plupart des définitions indiquent que l'objet semble changer de classe.
- Implémenter un exemple d'utilisation de State



# UML

handle est en fait une ou plusieurs méthodes  
elles représentent les différents événements  
`request(){ state.handle()}`



handle appelle Contexte.setState lorsque l'événement nécessite une transition.

A white icon on a dark blue background. It features a vertical ruler with horizontal tick marks on the left side. A pencil is positioned diagonally across the ruler, with its tip pointing towards the bottom right.

**PROXY**

# PATTERNS: PROXY

ID: #15

En tant que développeur

Je veux isoler la question du contrôle de l'accès à un objet donnée (instanciation, appels sur une machine distante, sécurisation des appels)

Afin de faire un appel distant, économiser des ressources, interdire l'appel de certaines méthodes à certaines références.

Priorité: Moyenne

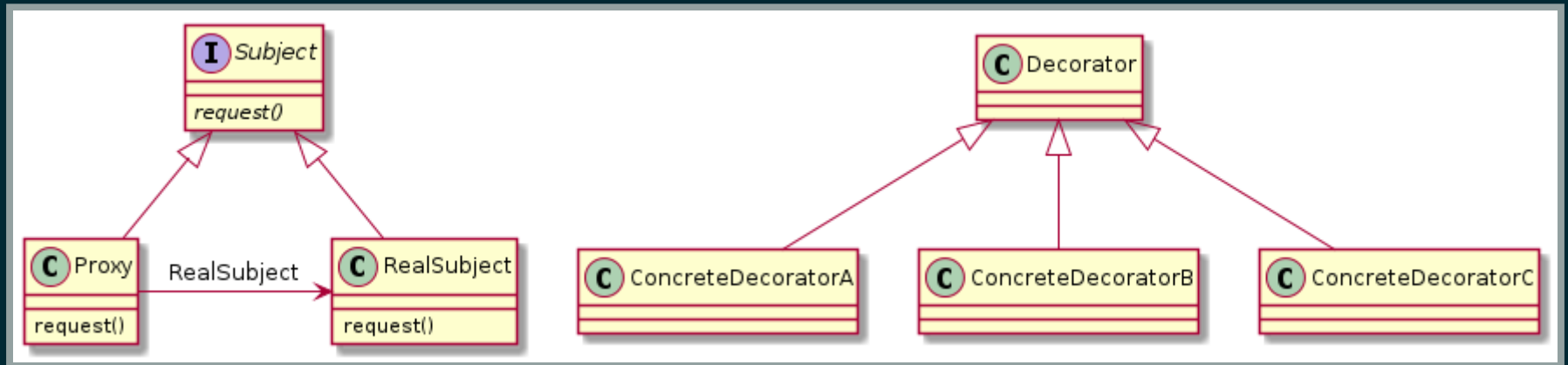
Valeur: 400

# CRITÈRES D'ACCEPTATION

ID: #15

- Être capable de décrire le fonctionnement du pattern proxy
- Implémenter un proxy d'accès remote avec rmi ou un virtualproxy (avec cache)

# UML



# UN DECORATOR? NON


- Un decorator enveloppe une instance existante

# UN STATE?

- Parfois, dans le cas du cachingproxy par exemple

# **EXEMPLES:**

- Les icones en attendant des miniatures dans les OS
- `inetd`
- `java.lang.reflect.Proxy`
- `java.rmi.*`



# **PATTERNS COMPOSÉS**



# PATTERNS COMPOSÉS

ID: #16

En tant que développeur

Je veux identifier des moyens de composer des patterns

Afin de résoudre des problèmes de plus en plus complexes

Priorité: Moyenne

Valeur: 800

# CRITÈRES D'ACCEPTATION

ID: #16

- Citer et décrire une version d'un pattern composé qui est lui même considéré comme un pattern.
- Implémenter un pattern composé à partir d'autres patterns du cours.

# MVC

## Modèle

La logique, les données, l'état actuel.

## Vue

Ce qu'on présente à l'utilisateur

## Controlleur

Fait le chef d'orchestre

# PATTERNS (SOUVENT) EN JEU

## Observer

La vue observe le modèle (en pull ou en push)

## Strategy et Command

Le controller fourni à la vue des stratégies (doit-on afficher ce bloc?), et des commandes à appeller lorsqu'une action est effectuée.

## Composite

La plupart du temps les librairies de vues sont implémentées comme composites

# BIBLIOGRAPHIE

## Head First Design Patterns

de Elisabeth Robson, Bert Bates, Eric Freeman:

<http://amzn.eu/8Fe610f>

## Design Patterns: Elements of Reusable Object-Oriented Software

Erich Gamma, Richard Helm, Ralph Johnson, John

Vlissides: <http://amzn.eu/cFzZ3C5>

## Exemples dans le JDK

<https://stackoverflow.com/a/2707195>

