

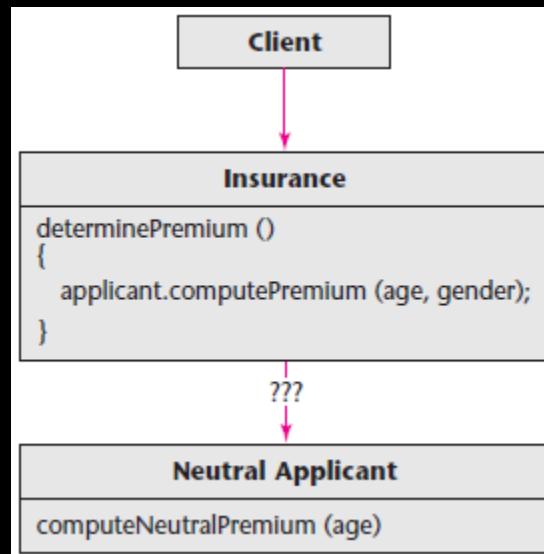
IFT 2255 - Genie Logiciel

Patrons de conception

QUESTION

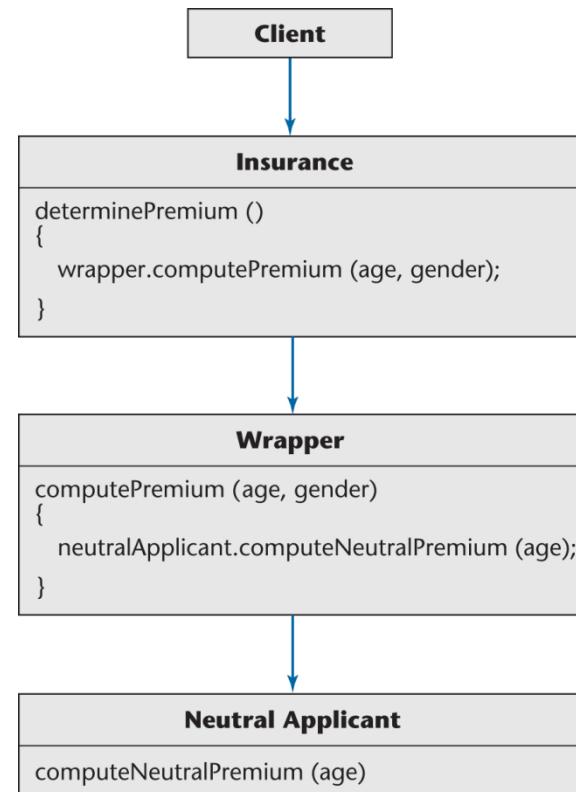
Quand la classe A envoie un message m à la classe B, A lui passe 2 paramètres. Mais pour B, m ne s'attend qu'à 1 paramètre. Modifier directement A et/ou B déclenchera des problèmes d'incompatibilité dans d'autres classes.

Comment résoudre ce problème d'incompatibilité d'interfaces ?



Wrapper (*Emballage*)

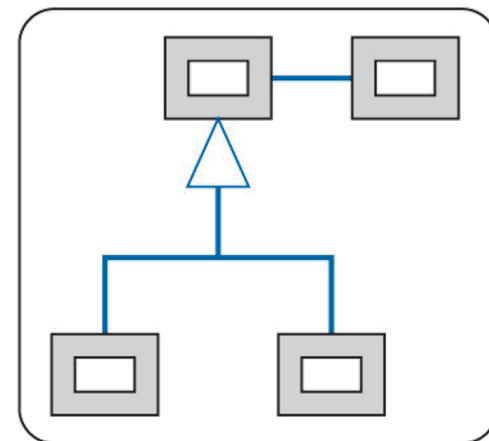
- Construire une classe C qui prend les 2 paramètres de A et n'en passe qu'un seul à B
- C est un **wrapper**



- L'emballeur est un cas particulier du patron **Adaptateur**

Patrons de conception

- Un patron décrit et nomme une **solution à un problème commun**
 - Offre une solution **abstraite** pour faciliter la **réutilisation**
 - Est formulé en termes de classes, d'objets et d'interactions entre eux
 - Peut être implémenté différemment en fonction du langage de programmation utilisé
- La solution proposée par un patron peut être modifiée ou adaptée selon les besoins du logiciel
- « Forcer » l'utilisation d'un patron dans un logiciel est une mauvaise pratique de développement

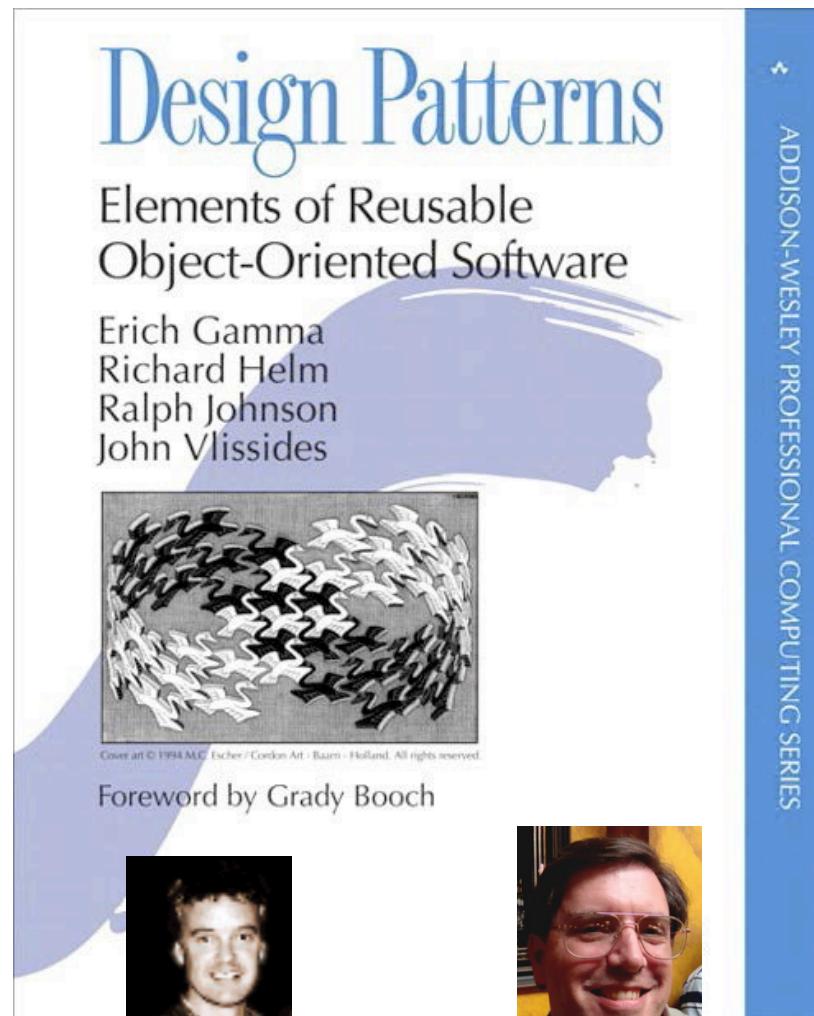


Objectifs des patrons

- Les patrons visent en général à accroître la **qualité** du code en visant un ou plusieurs des objectifs suivant:
 - **Flexibilité** accrue
 - Meilleure **compréhension/performance**
 - **Fiabilité** accrue
- Attention! L'utilisation des patrons peut aussi augmenter la **complexité** du code
 - Par exemple, ajout d'indirections
 - Il faut donc juger des avantages et inconvénients de l'ajout de patrons dans la conception d'un logiciel
 - Chaque patron vient avec des compromis à faire

Catalogue de 23 patrons de conception OO

Livre du “Gang of Four”



*Couvert en détail
dans IFT 3911*



Types de patrons

- **Patrons de construction**

- Abstraire le processus d'instanciation
- Rendre le système indépendant de comment les objets sont créés, composés et représentés

- **Patrons de structure**

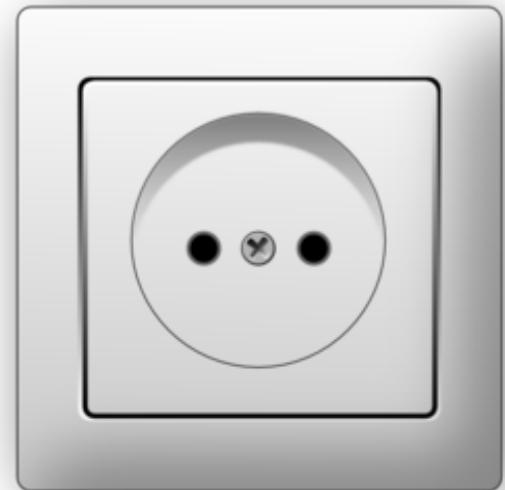
- Comment les classes et objets sont composés pour former de plus grandes structures
- Utilisés pour offrir de nouvelles fonctionnalités

- **Patrons de comportement**

- Algorithmes et assignation de responsabilité entre objets
- Communication entre objets et comportement interne de l'objet

Patron de l'adaptateur

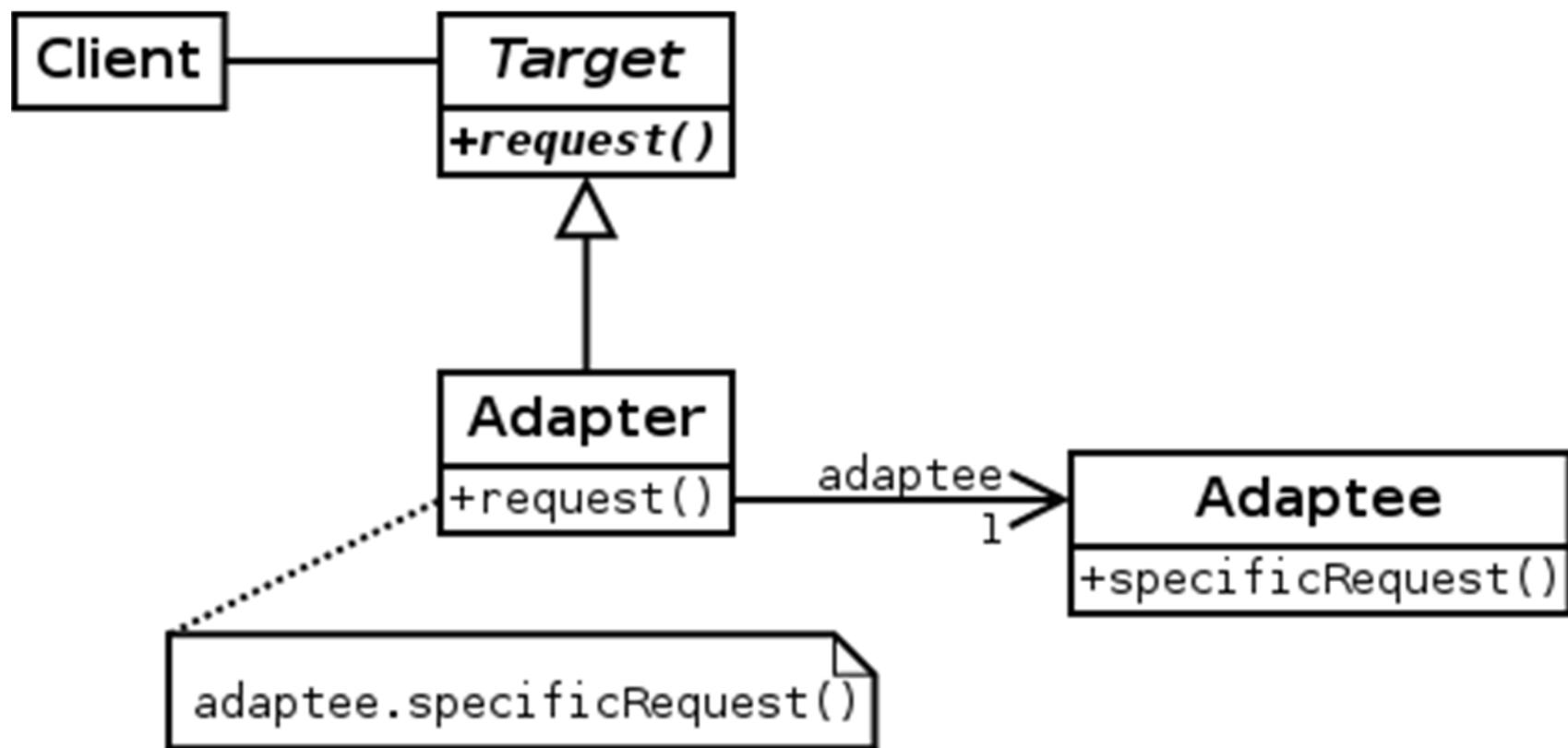
Analogie aux prises électriques dans le monde



Adaptateur

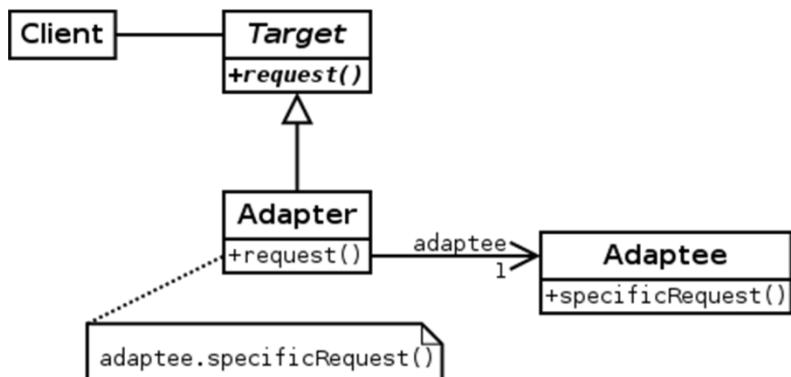
- Problème: Communication entre deux objets ayant leurs implémentations **incompatibles**
- Solution: **Convertir l'interface** d'une classe en une autre interface attendue par le client afin de leur permettre de travailler de concert
- Donne accès à son implémentation interne sans pour autant **coupler** les clients à son implémentation interne
- Adaptateur fournit tous les avantages de la **dissimulation de l'information** sans avoir à cacher ses détails d'implémentation

Structure de l'adaptateur

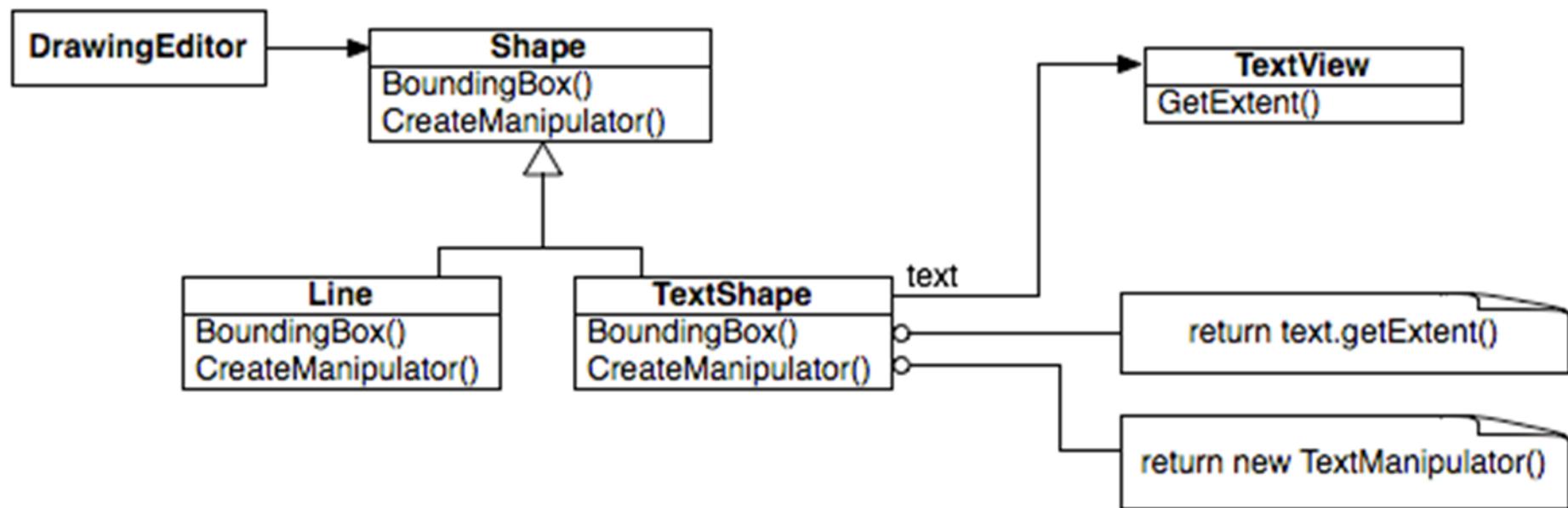


Participants

- Adapté (*Adaptee*)
 - Classe existante avec un comportement
- Cible (*Target*)
 - Définit l'interface attendue du Client
- Adaptateur (*Adaptor*)
 - Implémente l'interface de la Cible en utilisant la fonctionnalité de l'Adapté
- Client
 - Communique avec les classes qui implémentent l'interface de la Cible



Exemple



Interfaces incompatibles en Java

```
public class NorthAmericanPlug {  
    public String getNAVoltage() {  
        return "110V";  
    }  
}
```

```
public class EuropeanPlug {  
    public String getEUVoltage() {  
        return "220V";  
    }  
}
```

```
public class Socket {  
    public String outputToPlug(NorthAmericanPlug plug)  
    {  
        return "Delivering power to " + plug.getNAVoltage();  
    }  
}
```

```
public class Client {  
    public static void main(String[] args) {  
        Socket naSocket = new Socket();  
  
        NorthAmericanPlug naPlug = new NorthAmericanPlug();  
        System.out.println(naSocket.outputToPlug(naPlug));  
  
        EuropeanPlug euPlug = new EuropeanPlug();  
        System.out.println(naSocket.outputToPlug(euPlug));  
    }  
}
```

Adaptateur en Java

```
public class NorthAmericanPlug {  
    public String getNAVoltage() {  
        return "110V";  
    }  
}
```

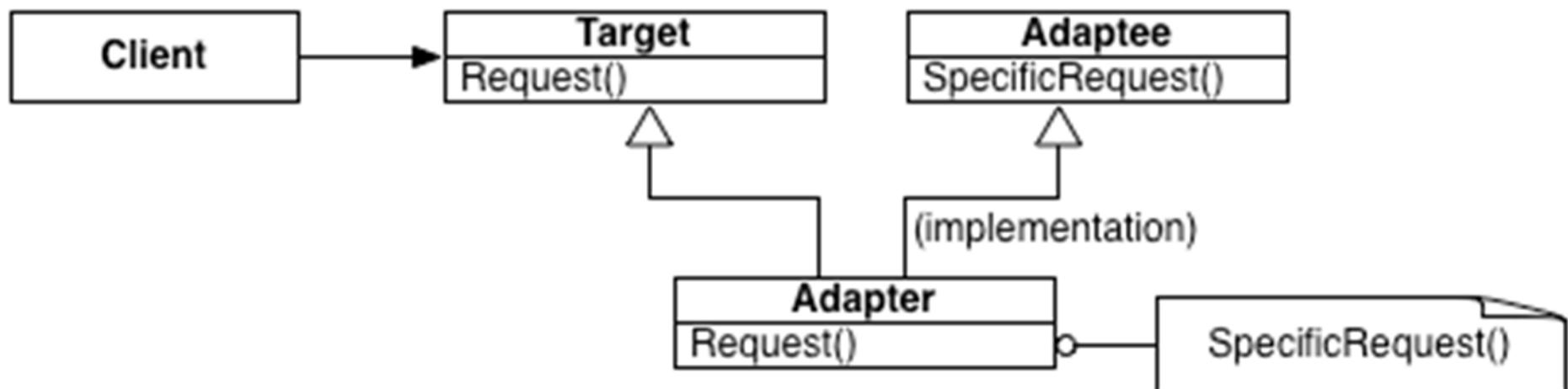
```
public class EuropeanPlug {  
    public String getEUVoltage() {  
        return "220V";  
    }  
}
```

```
public class Socket {  
    public String outputToPlug(NorthAmericanPlug plug)  
    {  
        return "Delivering power to " + plug.getNAVoltage();  
    }  
}
```

```
public class PlugAdapter extends NorthAmericanPlug {  
    EuropeanPlug euPlug;  
  
    public PlugAdapter()  
    {  
        euPlug = new EuropeanPlug();  
    }  
  
    @Override  
    public String getNAVoltage() {  
        return euPlug.getEUVoltage();  
    }  
}
```

```
public class Client {  
    public static void main(String[] args) {  
        Socket naSocket = new Socket();  
  
        NorthAmericanPlug naPlug = new NorthAmericanPlug();  
        System.out.println(naSocket.outputToPlug(naPlug));  
  
        PlugAdapter euPlugAdapter = new PlugAdapter();  
        System.out.println(naSocket.outputToPlug(euPlugAdapter));  
    }  
}
```

Variante par héritage multiple



Patron du singleton

QUESTION

Nommez des objets dont on a besoin en un seul exemplaire

- Boîte de dialogue
- Pour gérer les préférences d'une application
- Pour écrire des logs
- Cache
- Thread pool
- Pilote (*device driver*) pour imprimante, carte graphique

Contexte

- Jeu en ligne massivement multi-joueurs (MMOG)
- Monde contient un nombre fixe d'objets du jeu, dont certains sont contrôlés par des humains (joueurs)
 - Joueur peut se déplacer dans le jeu, examiner des items, les prendre et les laisser tomber
- Chaque objet (joueur, item, arbre, gazon, etc.) a un ID unique qui lui est associé
- Comment décerner les IDs en étant sûr de ne jamais donner deux fois le même?



Gestionnaire d'ID

- Utiliser des identifiants uniques pour tous les objets du jeu
- IDs doivent être décernés par un seul objet
 - Sinon, on peut assigner un ID en double
- L'application doit avoir un accès global à ce distributeur
 - Compliqué et lourd de passer la référence au distributeur partout dans l'application
- S'assurer qu'il n'y a qu'**une seule instance** du distributeur d'ID qui peut être créée
- L'instance doit être **accessible de partout** dans l'application

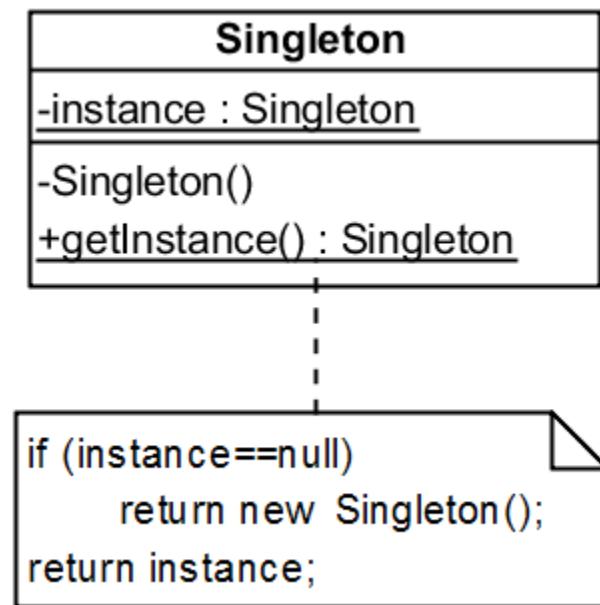
QUESTION

Pourquoi ne pas utiliser une variable globale à la place?

- Couplage clandestin
- Pas de contrôle d'accès
- Problème de concurrence
- Difficile de localiser une erreur lors des tests
- Doit créer l'objet au tout début de l'application, même s'il ne sera pas utilisé lors de l'exécution

Singleton

Garantir qu'une classe n'a qu'une seule instance
et offre un point d'accès global



En code

```
public class Singleton {  
    private static Singleton _instance;  
    private Singleton() {}  
    public static Singleton getInstance() {  
        if (_instance == null) {  
            _instance = new Singleton();  
        }  
        return _instance;  
    }  
}
```

Conséquences

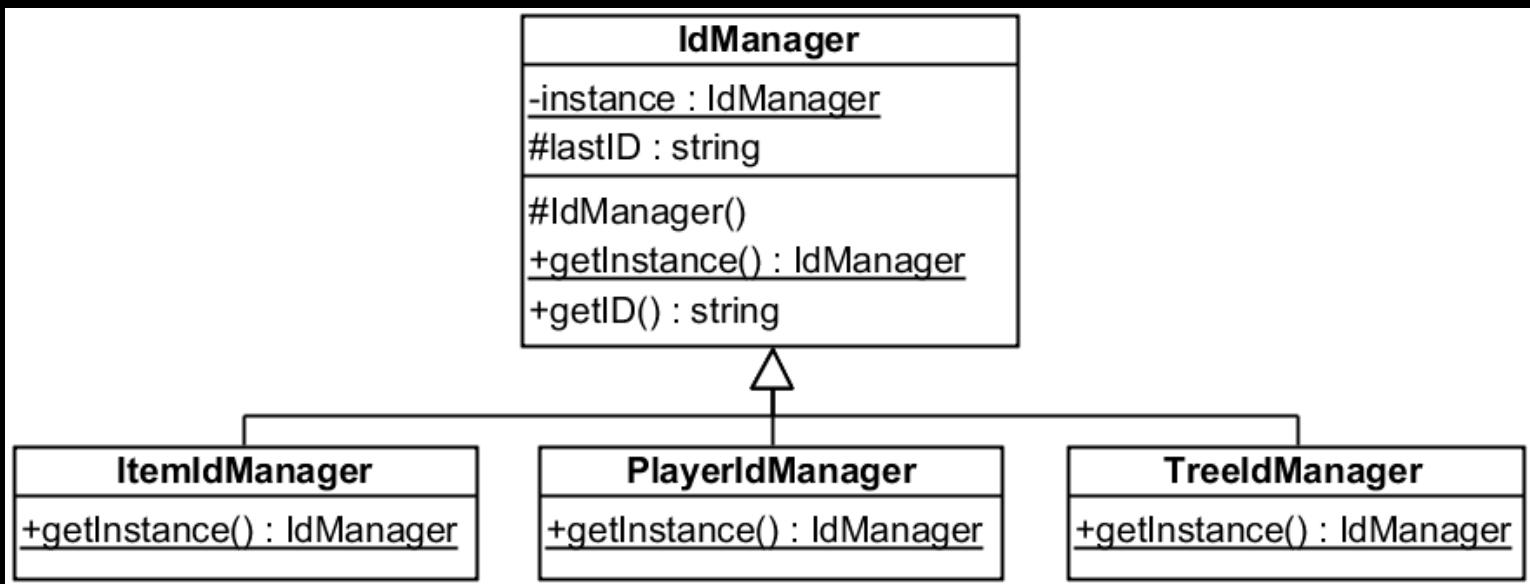
- Garantie qu'une seule instance peut être créée
- Contrôle l'accès à l'instance
- Accès global sans utiliser de variable globale
- Facilite le raffinement des opérations et représentations
 - Singleton peut-être sous-classé
- Peut permettre un nombre limité d'instances
 - Comment?

Implémentation du IdManager

```
public class IdManager {  
  
    private static IdManager _instance = new IdManager();  
  
    private string lastID;  
  
    private IdManager {  
        this.lastID = -1;  
    }  
  
    public static IdManager getInstance() {  
        return _instance  
    }  
  
    public string generateID() {  
        return ++this.lastID;  
    }  
}
```

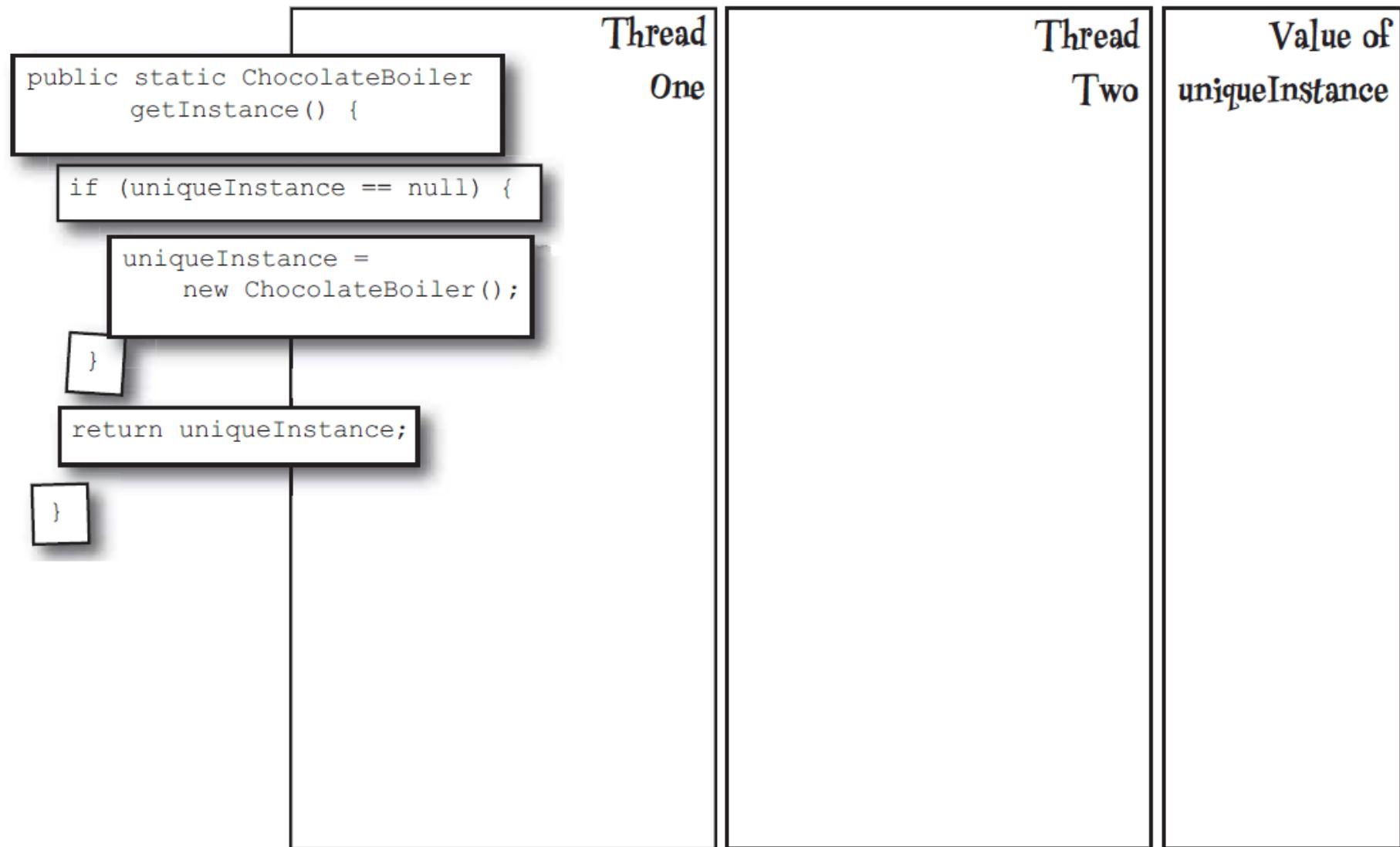
QUESTION

Est-ce que ceci gère correctement les Ids pour des types différents?

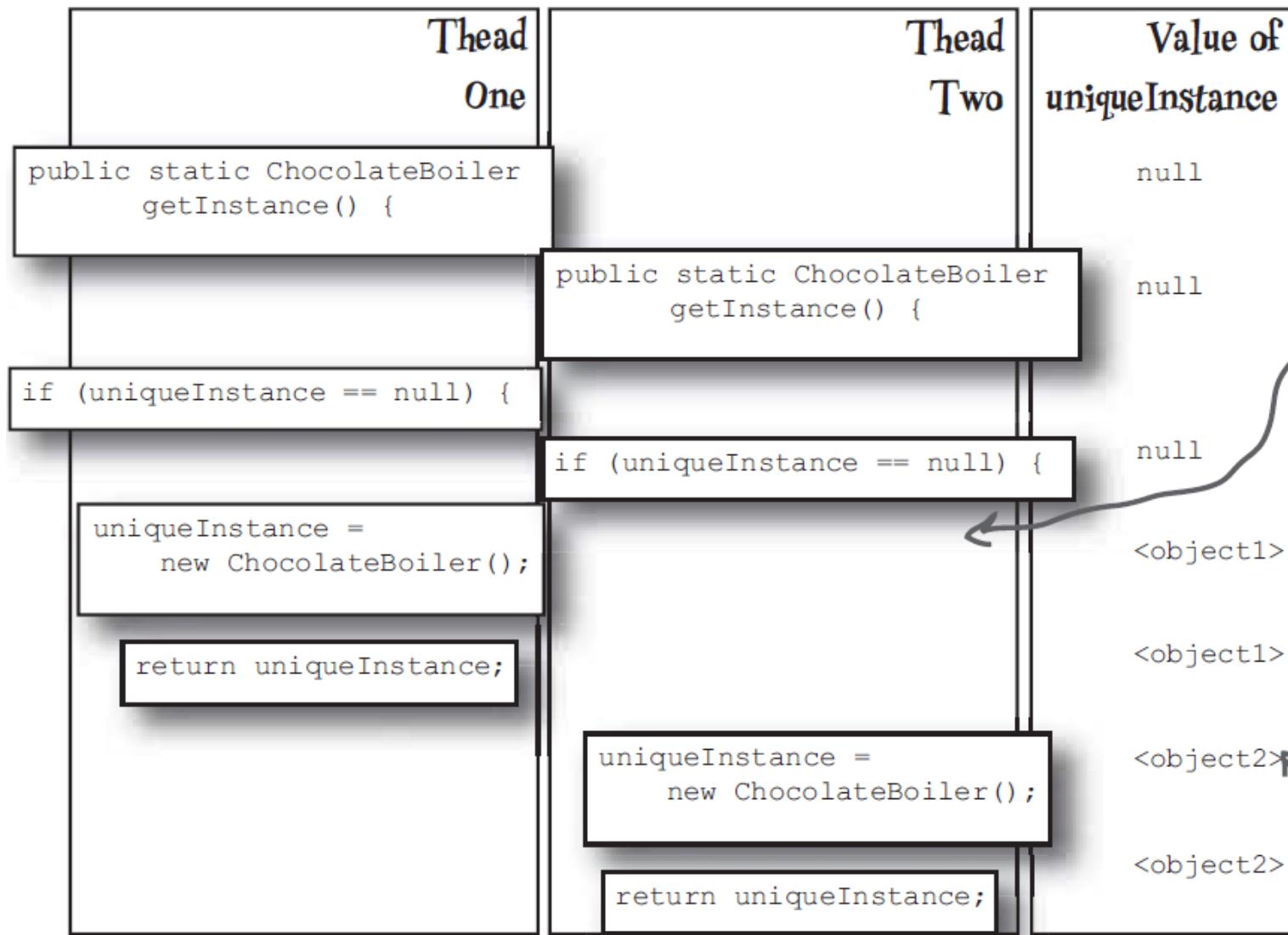


- Plusieurs singltons!
- Utiliser une classe générique
- Utiliser un dictionnaire

Que ce passe-t-il s'il y a 2 threads d'exécution?



Que ce passe-t-il s'il y a 2 threads d'exécution?



Alternative thread-safe

Méthode synchronized

```
public class Singleton {  
    private static Singleton _instance;  
    private Singleton() {}  
    public static synchronized Singleton getInstance() {  
        if (_instance == null) {  
            _instance = new Singleton();  
        }  
        return _instance;  
    }  
}
```

- Synchronisation est coûteuse
 - Doit être faite à chaque fois que l'on veut accéder à l'objet

Alternative thread-safe

Initialisation avide

```
public class Singleton {  
  
    private static final Singleton instance = new Singleton ();  
  
    //private constructor to avoid client applications to use constructor  
    private Singleton (){}  
  
    public static Singleton getInstance(){  
        return instance;  
    }  
}
```

- Si la classe n'utilise pas beaucoup de ressources

Alternatives

Initialisation statique

```
public class Singleton {  
    private static Singleton instance;  
    private Singleton() {}  
  
    //static block initialization for exception handling  
    static {  
        try {  
            instance = new Singleton();  
        } catch (Exception e) {  
            throw new RuntimeException("An error occurred.");  
        }  
    }  
  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

- Permet le traitement d'exception
- Avide et statique: crée l'instance avant de l'utiliser
 - Pas une bonne pratique

Alternatives

Verrouillage à double test

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

- Un seul thread peut appeler getInstance() à la fois
- Plus performant que si getInstance() est synchronized
 - (*Double checked locking*)

Alternatives

Énumération du Singleton

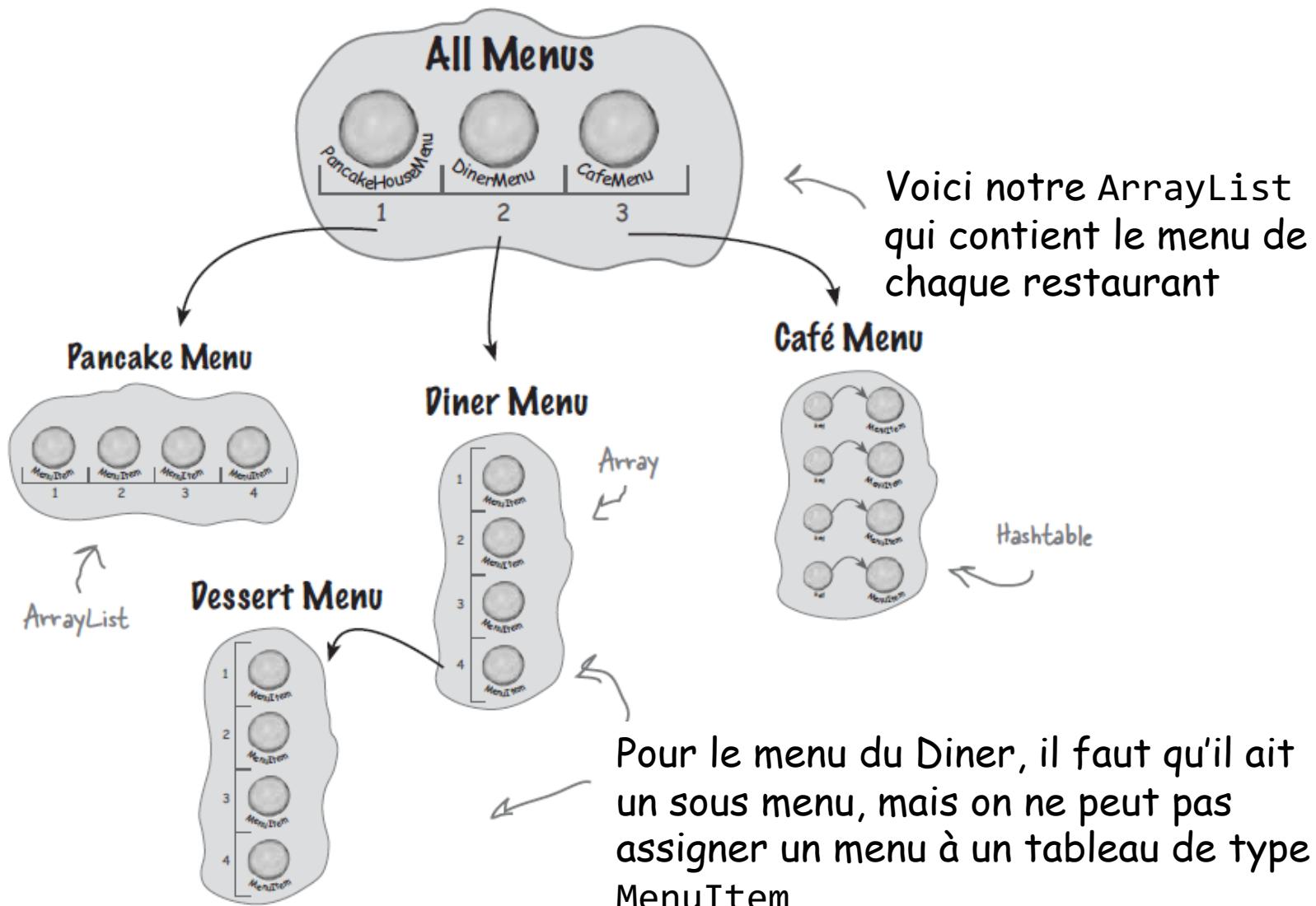
```
public enum Singleton {
    INSTANCE;
    public void do() {
        // perform something
    }
}

public static void main(String[] args) {
    Singleton.INSTANCE.do();
}
```

- Dans Java 5 et +
- Enum garantie qu'il n'y a qu'une seule instance même dans un environnement multi-threaded
- Accessible de partout

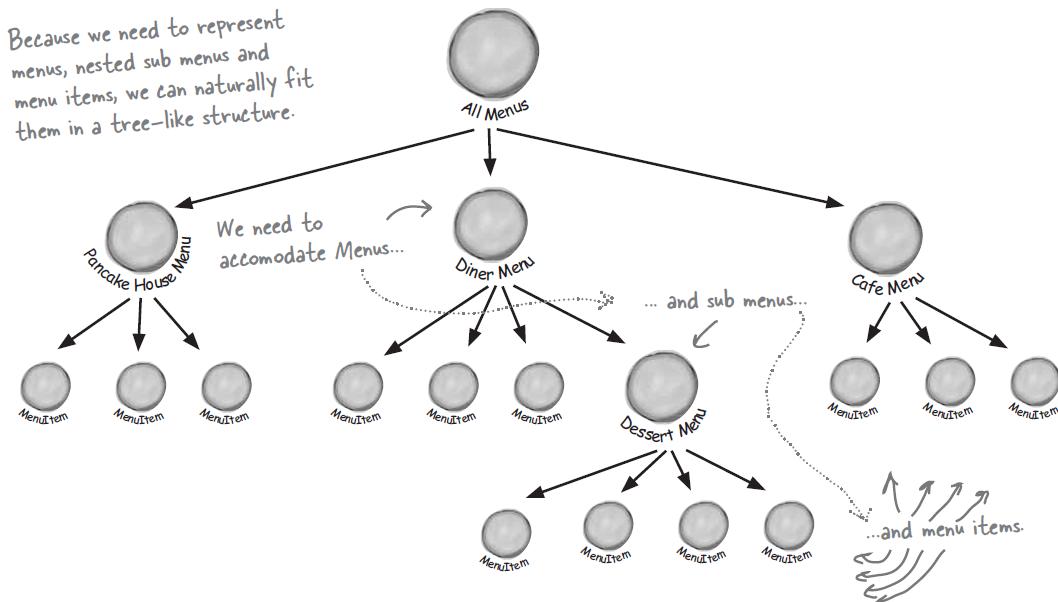
Patron de l'objet composite *(Composite pattern)*

Problème de hiérarchie compositionnelle

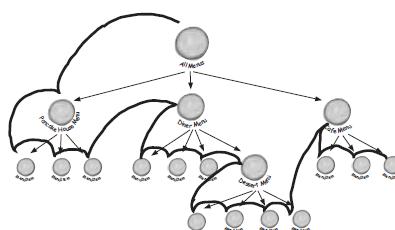


Besoins

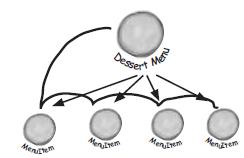
- Structure d'arborescence qui accommode les menus, sous menus et items de menu
- Maintenir une façon d'accéder à chaque item de chaque menu d'une manière transparente
 - Sans souci du type d'objet



We still need to be able to traverse the all the items in the tree.



We also need to be able to traverse more flexibly, for instance over one menu.

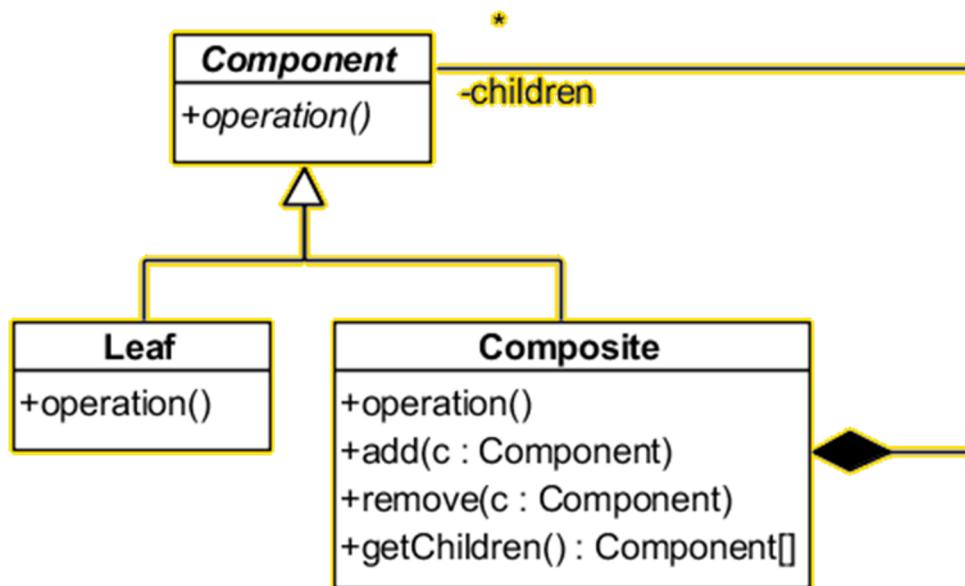


Objet Composite

Compose les objets en structures d'arborescence pour représenter des hiérarchies d'ensemble/partie

- Permet au client de traiter les objets et les compositions de façon uniforme
 - Objets primitifs, atomiques
 - Objets composites

Structure de l'objet composite



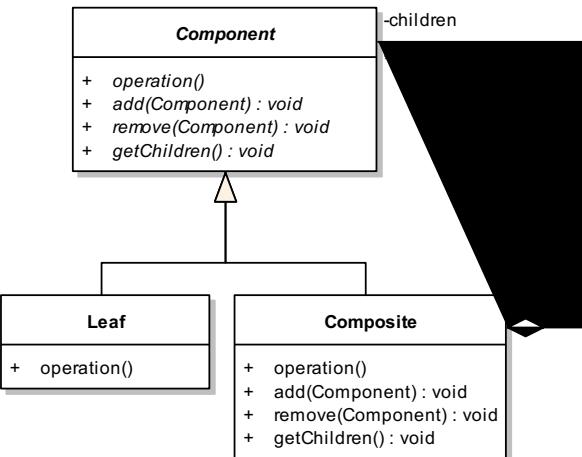
Participants

- Composant
 - Déclare l'interface pour les objets
 - Implémente le comportement par défaut
 - Déclare les interfaces de gestion des composants enfant

- Feuille
 - Représente les primitives: pas d'enfant

- Composite
 - Définit le comportement des composants qui ont des enfants
 - Implémente les opération de gestion des enfants

- Client
 - Manipule les objets via l'interface Composant



Objet composite en Java

```
public abstract class Employee {  
    protected String name;  
    protected double salary;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public double getSalary() {  
        return salary;  
    }  
    public void setSalary(double salary) {  
        this.salary = salary;  
    }  
  
    public abstract String parenthesize();  
}
```

```
public class Developer extends Employee {  
    public Developer(String name)  
    {  
        this.name = name;  
    }  
  
    @Override  
    public String parenthesize() {  
        return this.name;  
    }  
}
```

```
import java.util.ArrayList;  
import java.util.List;  
  
public class Manager extends Employee {  
    private List<Employee> children;  
  
    public List<Employee> getChildren() {  
        return children;  
    }  
  
    public void setChildren(List<Employee> children) {  
        this.children = children;  
    }  
  
    public Manager(String name)  
    {  
        this.name = name;  
        this.children = new ArrayList<Employee>();  
    }  
  
    public void add(Employee e) { this.children.add(e); }  
    public void remove(Employee e) { this.children.remove(e); }  
  
    @Override  
    public String parenthesize() {  
        String s = this.name + " (";  
        for (Employee e : this.children)  
            s += e.parenthesize() + " ";  
        return s + ")";  
    }  
}
```

Objet composite en Java

```
public class Client {
    public static void main(String[] args)
    {
        Manager john = new Manager("John"),
                  mike = new Manager("Mike"),
                  ursula = new Manager("Ursula");
        Developer ed = new Developer("Ed"),
                  jessie = new Developer("Jessie"),
                  jane = new Developer("Jane"),
                  brian = new Developer("Brian");

        john.add(mike);
        john.add(ursula);
        john.add(ed);
        mike.add(jessie);
        mike.add(jane);
        ursula.add(brian);

        System.out.println(john.parenthesize());
    }
}
```

Output: John (Mike (Jessie Jane) Ursula (Brian) Ed)

Conséquences

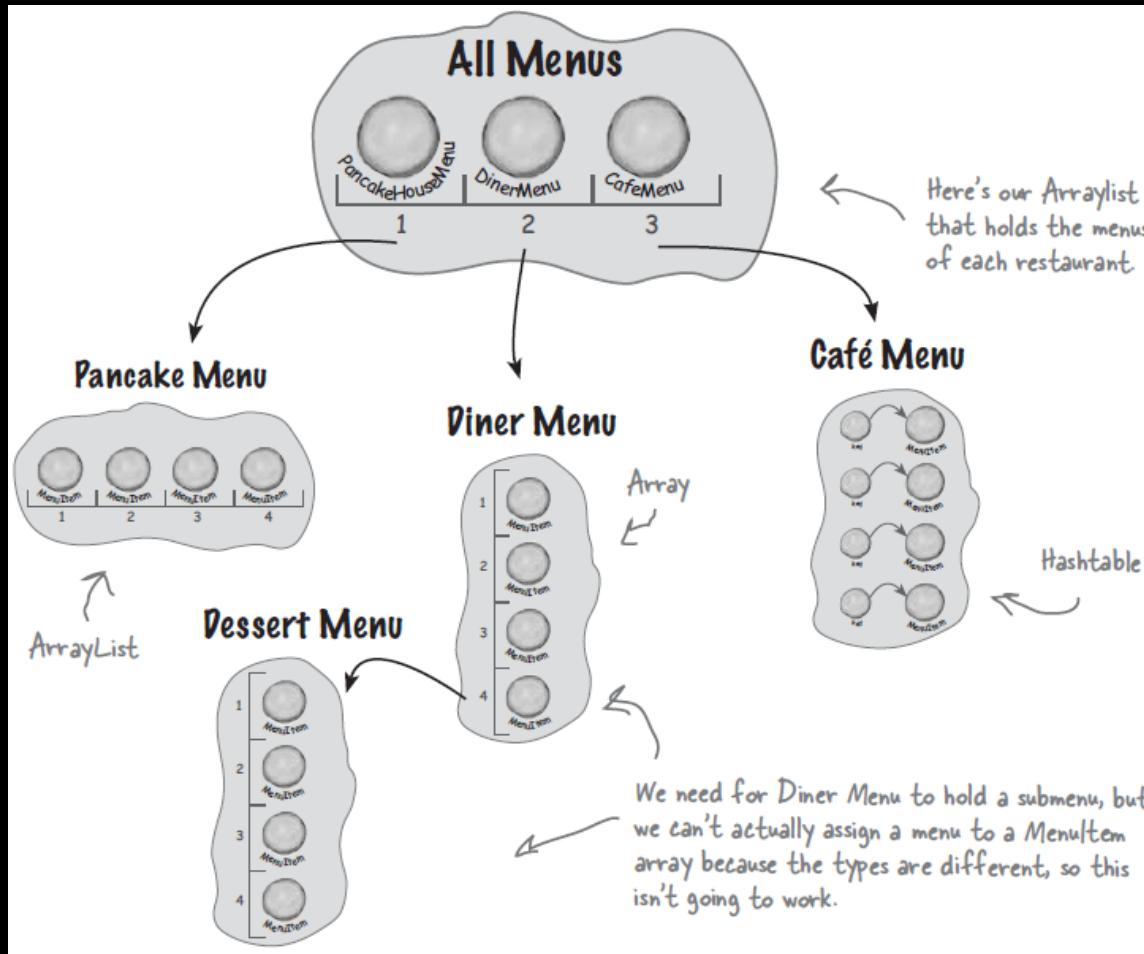
- Définit une **hiérarchie claire** des objets
 - **Composition récursive** d'objets simples et complexes
 - Pas besoin de vérifier le type d'objet (dissimulation d'info)
 - Pas besoin de répétition de switch/case pour chaque opération
- Ajout facile de nouveaux composants
 - Soit un composite soit une feuille: fonctionne directement avec la structure en place
 - Pas besoin de changer le code client: réutilisation
- Rend le design trop général
 - Difficile de restreindre le type de composants dans un composite
 - Besoin de vérifications lors de l'exécution

Principe de substitution de Liskov (LSP)

- Si B est une sous-classe de A, alors un objet b de type B peut être utilisé dans **n'importe quel contexte** où un objet a de type A est attendu
- Dans un programme, on peut remplacer a par b sans avoir à changer ce programme
- Patron d'objet composite est un bon exemple d'application de ce principe

QUESTION

Comment représenter le menu de Starbuzz ?



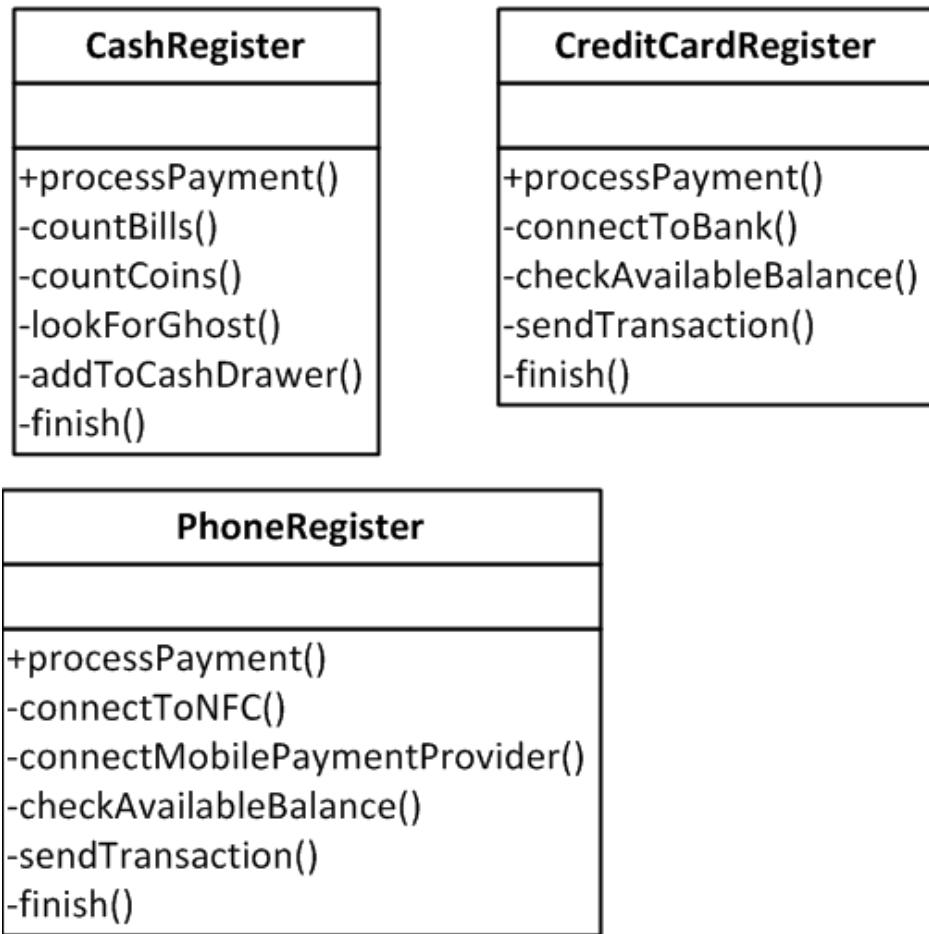
Patron de méthode *(Template method)*

Mode de paiement



Problème

- Implémentations différentes
 - Comptant, carte, téléphone
- Niveau de détail différent
 - Comment confirmer que le client a assez d'argent
- Même workflow
 - Mêmes étapes



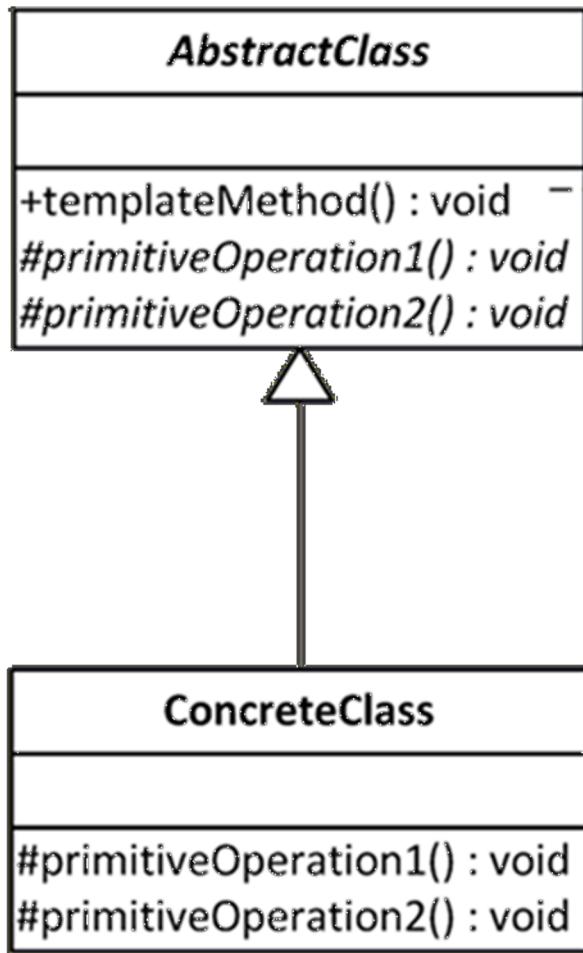
Comment factoriser un même algorithme général en tenant compte des différences dans les détails?

Patron de méthode (*Template Method*)

Définit le **squelette** d'un algorithme dans une opération en **reportant** certaines étapes à des **sous-classes**

- Permet d'**affiner** certaines étapes d'un algorithme sans changer sa structure

Structure



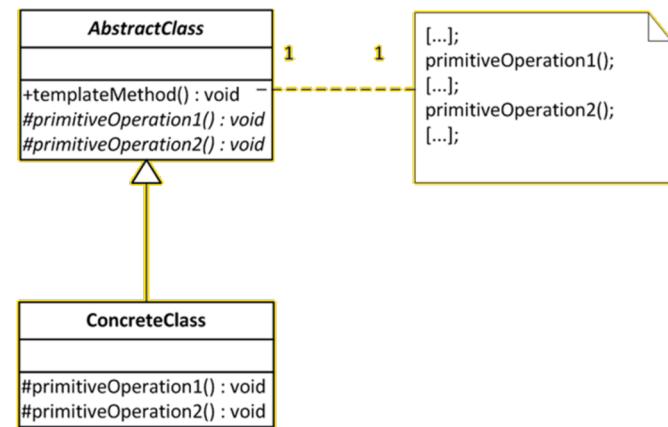
```
abstract class AbstractClass {

    public void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
    }

    protected abstract void primitiveOperation1();

    protected abstract void primitiveOperation2();
}
```

Participants



- **Classe Abstraite**

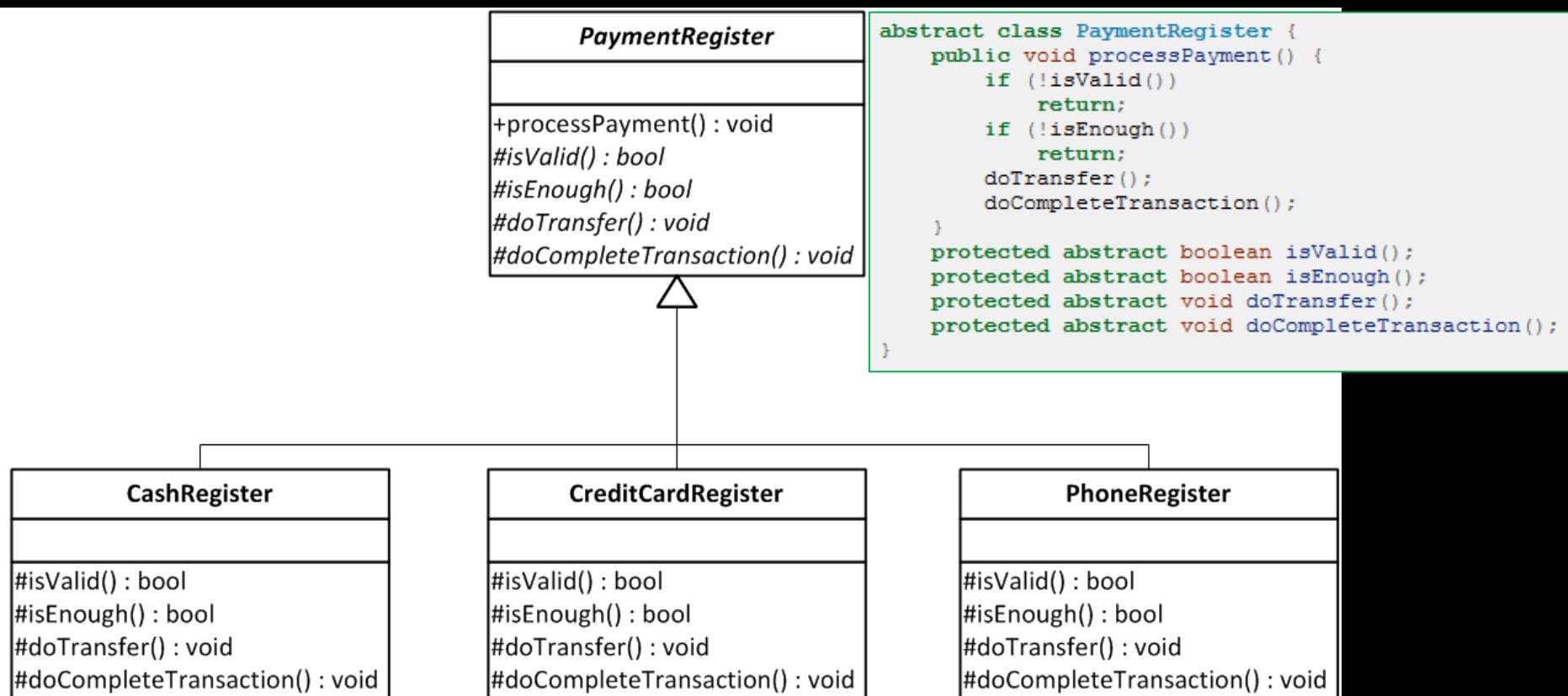
- Définit l'ordre des tâches qui vont être utilisées dans toutes les classes concrètes
- Choisit quelle tâche primitive doit être définie dans chaque classe concrète

- **Classe Concète**

- Implémente les tâches primitives

QUESTION

*Comment utiliser le patron de méthode pour le système de paiement?
Quelles sont les méthodes primitives?*



Très utile!

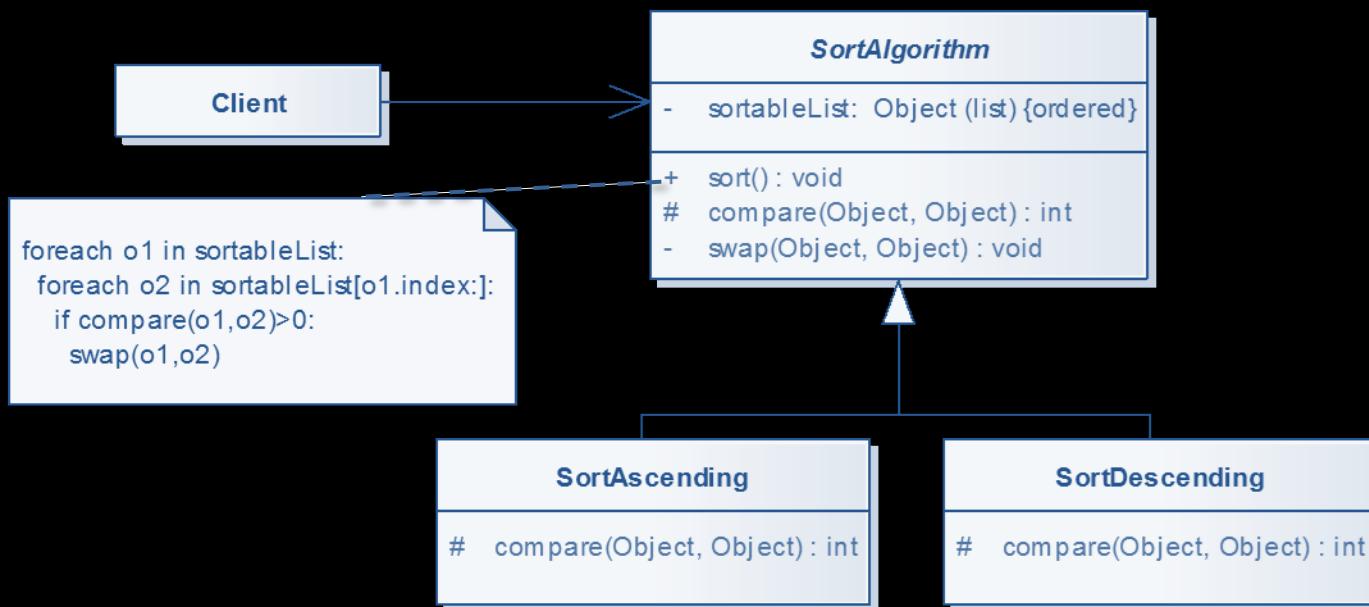
- Implémente les **parties invariantes** d'un algorithme et laisse les sous-classes implémenter le comportement qui va **varier**
 - Polymorphisme
- Évite la **duplication de code** en factorisant le comportement commun des sous-classes
 - Réutilisation
- Contrôle ses **extensions** en définissant des hameçons (*hooks*) à des endroits spécifiques
- Permet qu'une superclasse appelle des opérations dans les sous-classes

Conséquences

- Offrir un comportement par défaut, ainsi que des comportements alternatifs spécifiques à un intérêt
- Imposer l'appel à une méthode
 - abstract, virtual, final, sealed
- Important de minimiser le nombre d'opérations primitives
- Bonne application du LSP via polymorphisme

QUESTION

Implémenter un algorithme de triage croissant et décroissant d'une liste d'objets



Force des patrons de conception

- Promeut la **réutilisation** en résolvant un problème de conception général
- Fournit une documentation de la conception en spécifiant des **abstractions**
- **Implémentations** déjà existantes
 - Pas besoin de programmer et documenter cette partie du code
 - Mais, besoin d'être testées et adaptées à notre contexte
- Lors de la maintenance, il est plus facile de **comprendre** un programme qui utilise des patrons de conception
 - Même sans avoir vu ce programme avant!

Faiblesse des patrons de conception

- Pas de manière **systématique** de déterminer où et quand **utiliser** un patron de conception
- Programmes plus complexes emploient **plusieurs patrons** qui interagissent entre eux pour maximiser leurs avantages (et leurs inconvénients...)
 - Problème de gestion des dépendances entre patrons peut être très complexe et perdre leurs avantages
- Le fait qu'on ait besoin de 23+ patrons de conception peut indiquer que notre **langage/paradigme** n'est pas assez puissant, ou trop générique