ÉTS
Le génie pour l'industrie

LOG121

Conception orientée objet

Patron Itérateur et introduction aux patrons de conception

Enseignante: Souad Hadjres

Plan

- Introduction aux patrons de conception par l'application du patron Itérateur
- Les patrons de conception

Exemple de problème de conception

- Deux restaurants (PancakeHouse et Diner) ont été fusionnés et on aimerait fusionner leurs menus
- Pour stocker les items du menu:
  - PanckakeHouse utilise un ArrayList
  - Diner utilise un Array

- Aucun des deux restaurants ne veut changer son implémentation
- Les items sont implémentés de la même façon

Exemple de problème de conception

Implémentation d'un item du menu

```
public class MenuItem{
    String name;
    String description;
    boolean vegetarian;
    double price;
    public String getName (){ return name; }
    public String getDescription(){ return description; }
    public boolean IsVegetarian(){ return vegetarian; }
    public double getPrice(){ return price; }
    public MenuItem(String name, String description, boolean vegetarian,
                    double price){
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }
}
```

Exemple de problème de conception


```
public class PancakeHouseMenu{
    ArrayList menuItems;
    public PancakeHouseMenu() {
        menuItems = new ArrayList();
        addItem("K&B's Pancake Breakfast","Pancakes with scrambled eggs,and
                toast",true,2.99);
        addItem("Blueberry Pancakes","Pancakes made with fresh blueberries",true,3.49);
    }
    public void addItem(String name, String description, boolean vegetarian, double price){
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }
    public ArrayList getMenuItems() {
        return menuItems;
    }
    // other menu methods here
}
```

Exemple de problème de conception


```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;
    public DinerMenu() {
            menuItems = new MenuItem[MAX_ITEMS];
            addItem("Vegetarian BLT", "lettuce & tomato", true, 2.99);
            addItem("BLT","Bacon with lettuce & tomato on whole wheat", false, 2.99);  }
    public void addItem(String name, String description,boolean vegetarian, double price) {
            MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
            if (numberOfItems >= MAX_ITEMS) {
                    System.err.println("Sorry, menu is full! Can't add item to menu");
            } else {
                    menuItems[numberOfItems] = menuItem;
                    numberOfItems = numberOfItems + 1; }      } // end of addItem method here
    public MenuItem[] getMenuItems() {  return menuItems; }
    // other menu methods here
}
```

Exemple de problème de conception

- Pour imprimer les items des deux menus, on a besoin de deux boucles

Extrait d'une classe « Waitress » utilisant les deux menus

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();

for(int i=0; i<breakfastItems.size(); i++){
        MenuItem menuItem = (MenuItem)breakfastItems.get(i);
        System.out.println(menuItem.getName() + " ");
        System.out.println(menuItem.getPrice() + " ");
        System.out.println(menuItem.getDescription() + " ");
}

for(int i=0; i<lunchItems.lenght; i++){
        MenuItem menuItem = lunchItems[i];
        System.out.println(menuItem.getName() + " ");
        System.out.println(menuItem.getPrice() + " ");
        System.out.println(menuItem.getDescription() + " ");
}
```

Parties différentes

Exemple de problème de conception

- Deux boucles sont aussi nécessaires pour d'autres opérations utilisant les deux menus
  - Exemple: printVegetarianMenu()
- Le programme manipulant ces deux menus doit connaître leurs structures internes
- L'ajout d'un troisième menu nécessite une autre boucle
- Pas de réutilisation du code commun

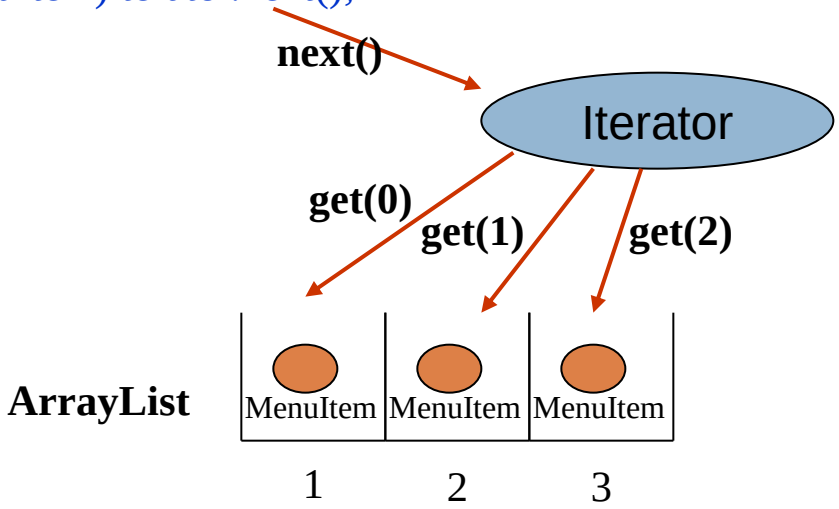
Exemple de problème de conception

- Comment améliorer cette conception?

Solution au problème

- Encapsuler la variation dans un objet: l'itération sur une collection
- Un itérateur sur le ArrayList

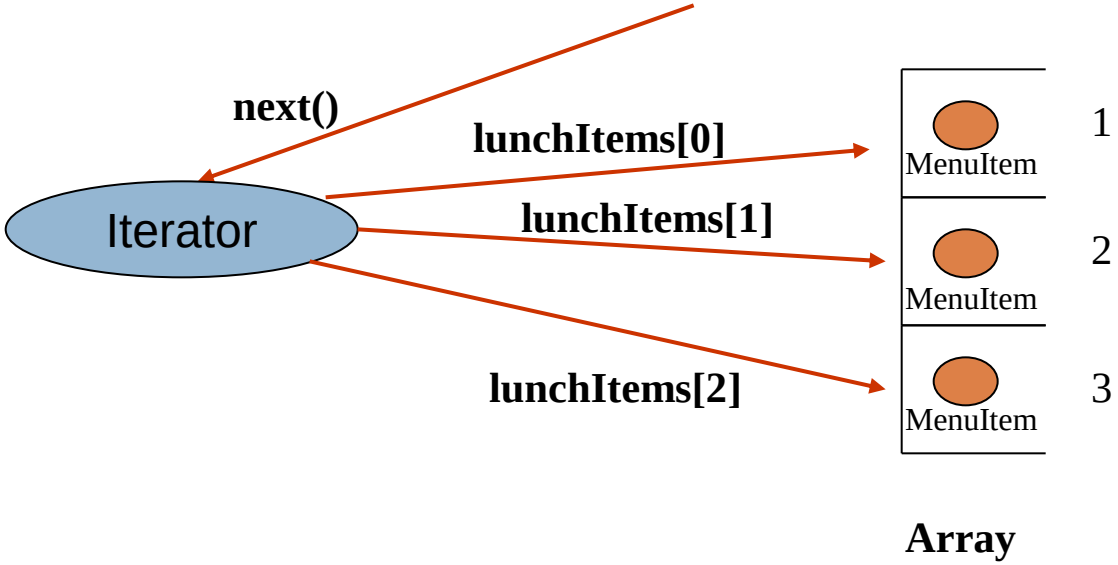
```
Iterator iterator = pankaceHouseMenu.createIterator();
while(iterator.hasNext())
{
    MenuItem menuItem = (MenuItem)iterator.next();
}
```



Solution au problème

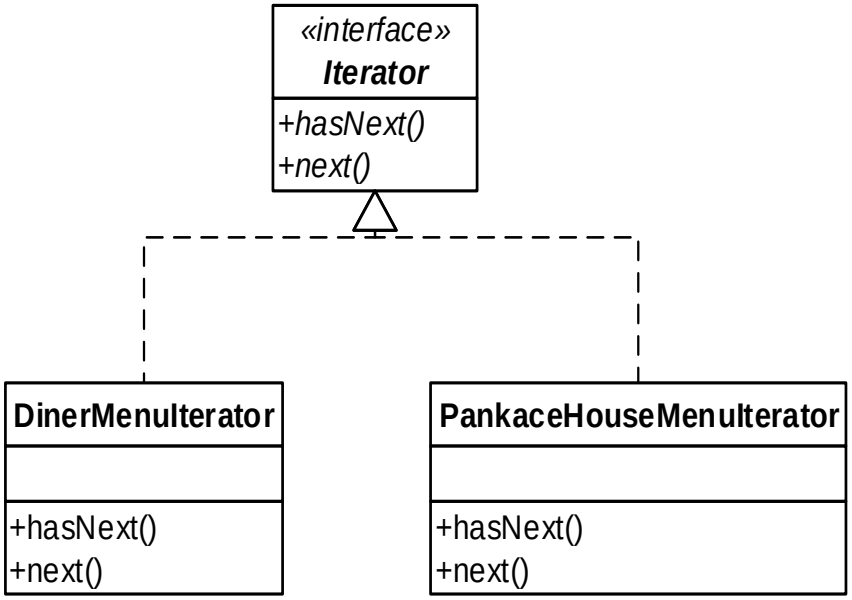
- Un itérateur sur le Array

```
Iterator iterator = dinerMenu.createIterator();
while(iterator.hasNext())
{
    MenuItem menuItem = (MenuItem)iterator.next();
}
```



Solution au problème

- Les deux itérateurs ont la même interface




```
public interface Iterator{
    boolean hasNext();
    Object next();
}
```

Encapsule l'itération sur un array

Encapsule l'itération sur un arrayList

Solution au problème


```
public class DinerMenuIterator implements Iterator {
    MenuItem[] items;
    int position = 0;
    public DinerMenuIterator(MenuItem[] items) {
        this.items = items;
    }
    public Object next() {
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
    }
    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
            } else {
                return true;
            }
    }
}
```

Solution au problème


```
public class PancakeHouseMenuIterator implements Iterator {
    ArrayList <MenuItem> items;
    int position = 0;
    public PancakeHouseMenuIterator(ArrayList <MenuItem> items){
        this.items = items;
    }
    public Object next() {
        MenuItem menuItem = (MenuItem)items.get(position);
        position = position + 1;
        return menuItem;
    }
    public boolean hasNext() {
    if (position >= items.size() {
        return false;
        } else {
            return true;
        }
    }
}
```

Solution au problème


```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;
    public DinerMenu() {
            menuItems = new MenuItem[MAX_ITEMS];
            … }
    public void addItem(String name, String description,boolean
    vegetarian, double price) {
            ….. }
    public MenuItem[] getMenuItems() { return menuItems; }
    public Iterator createIterator(){
            return new DinerMenuIterator(menuItems);
    }
// other menu methods here
}
```

Solution au p

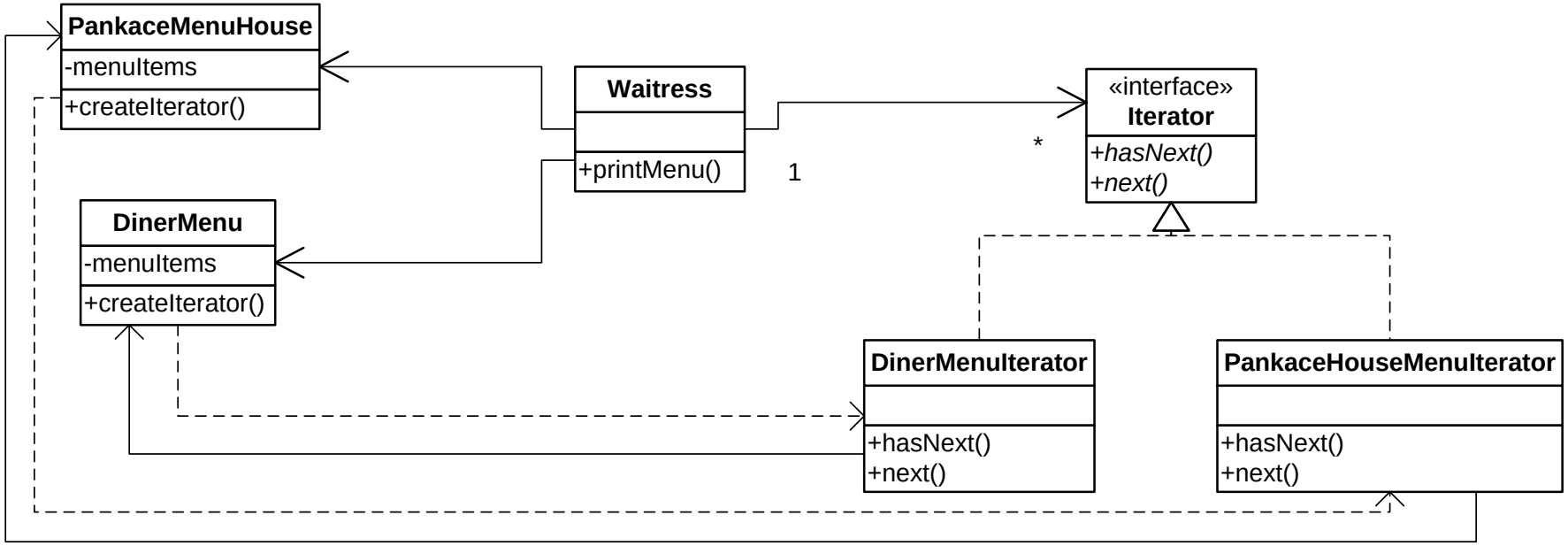

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu);
        waitress.printMenu();
    }
}
```

La classe Waitress révisée

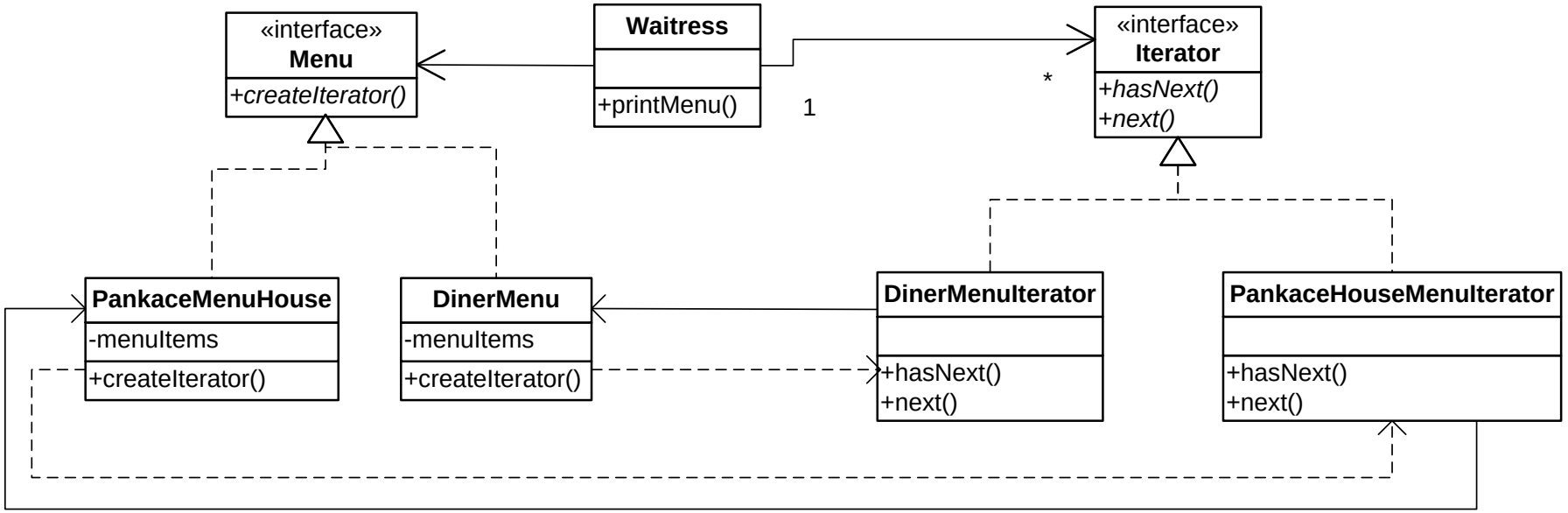
```
public class Waitress {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu dinerMenu;
    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {…}
    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        printMenu(pancakeIterator);
        printMenu(dinerIterator);
    }
    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " --");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

Solution au problème

- Nouvelle conception avec des itérateurs



Solution au problème



Solution au problème


```
public interface Menu {
    public Iterator<MenuItem> createIterator();
}
```

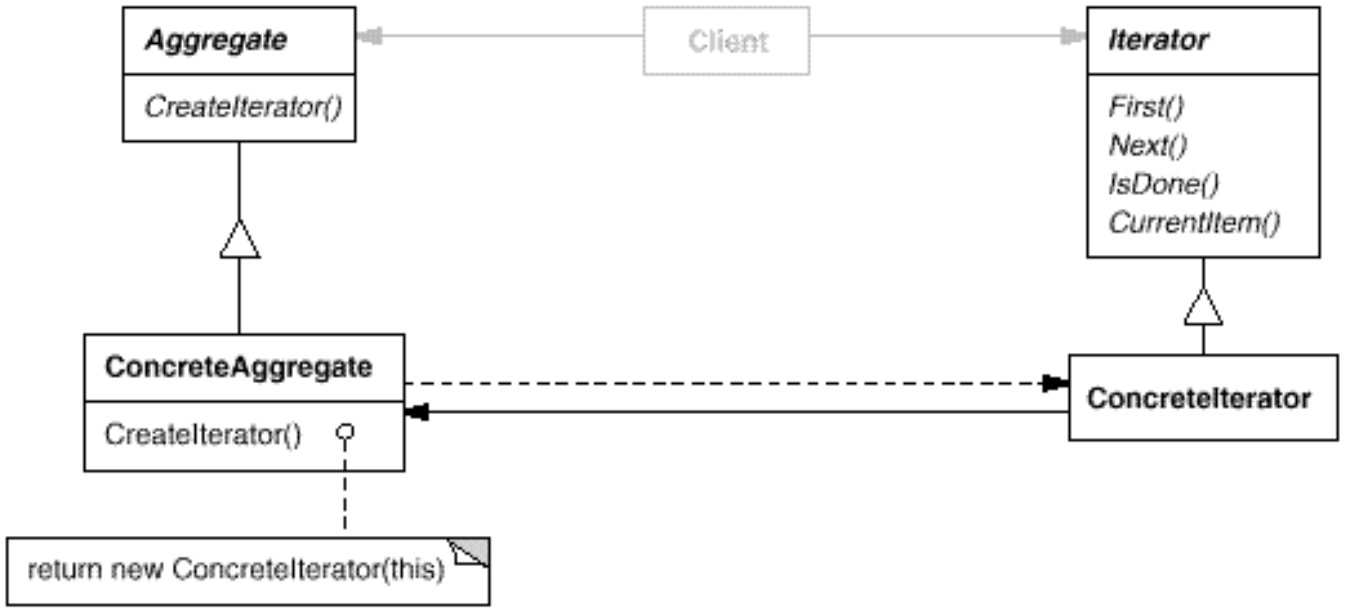
```
import java.util.Iterator;
public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;
    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {…}
    public void printMenu() {
            Iterator<MenuItem>  pancakeIterator = pancakeHouseMenu.createIterator();
            Iterator <MenuItem> dinerIterator = dinerMenu.createIterator();
            printMenu(pancakeIterator);
            printMenu(dinerIterator);
    }
    private void printMenu(Iterator iterator) {
            while (iterator.hasNext()) {
                    MenuItem menuItem = (MenuItem)iterator.next();
                    System.out.print(menuItem.getName() + ", ");
                    System.out.print(menuItem.getPrice() + " --");
                    System.out.println(menuItem.getDescription());
            }
}
```

La patron Itérateur

- Nous venons d'appliquer le patron Itérateur
- Intention du patron
  - « *Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.* » (Gamma et al., 95)
- Bénéfices
  - Permet l'itération polymorphique: fournit la même interface pour traverser différentes collections
  - On peut créer différents itérateurs pour la même collection d'objets: traverser la même collection de différentes façons
    - Plus puissant et flexible que d'utiliser un curseur dans une liste

La patron Itérateur

- La structure générique du patron



La patron Itérateur

- Question:
  - Énumérez les principes orientés objets que nous avons utilisé dans l'application du patron Itérateur.

La patron Itérateur

- Principes orientés objet
  - Encapsulation
  - Utilisation du polymorphisme
  - Séparation des responsabilités
    - Minimiser les raisons de changer une classe
      - Gérer une collection
      - Traverser une collection
    - Principe relié à la notion de cohésion

La patron Itérateur dans l'API Java

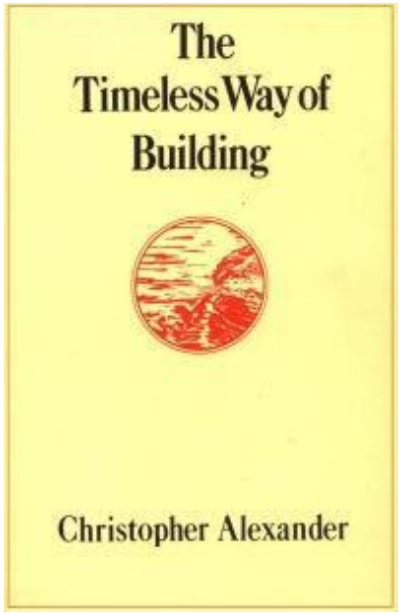
- Les structures de données utilisées font partie de **Java collections framework**
  - Ces classes implémentent l'interface java.util.Collection qui définit une méthode iterator()
- Pour changer notre exemple pour utiliser l'API Java
  - Modifier DinerMenuIterator pour implémenter java.util.Iterator
  - Supprimer PancakeHouseIterator car ArrayList a déjà une méthode qui retourne un itérateur java.
- Connaissez-vous d'autres classes de Java qui appliquent l'Itérateur?

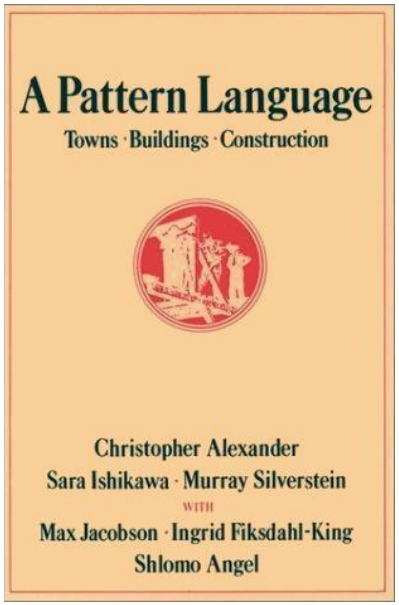
Plan

- Introduction aux patrons de conception par l'application du patron Itérateur
- Les patrons de conception

Les patrons de conception

- Inspirés des patrons introduits en 1977 par C.Alexander architecte en bâtiment





Les patrons de conception

- Les patrons Orientés Objet ont été introduits par K. Beck et W. Cunningham (1987);
- Au début des années 1990, des approches de patrons avancés de conception ont pris de la vitesse (Weinand et al.88 et 89, Helm et al.90, Gamma 92);
- La première conférence sur les patrons de conception fut organisée en 1994;
- Le livre « Design patterns » de Gamma, Helm, Johnson et Vlissides (GoF: Gang of four) décrit plusieurs patrons de conception.

Les patrons de conception

- Famille de solutions éprouvées visant à
  - Guider le concepteur à travers des micro-architectures utiles permettant de
    - Supporter la flexibilité, la maintenabilité, la réutilisation
    - Minimiser l'impact des changements
  - Capitaliser la connaissance d'experts sur la façon de concevoir
  - Fournir un vocabulaire commun aux concepteurs

Les patrons de conception

- «Un patron décrit un problème devant être résolu, une solution, et le contexte dans lequel cette solution est considérée. Il nomme une technique et décrit ses coûts et ses avantages. Il permet, à une équipe, d'utiliser un vocabulaire commun pour décrire leurs modèles » Johnson 97

Les patrons de conception

- Un patron est décrit par un ensemble de rubriques
  - Nom: un nom significatif
  - Contexte : ensemble de situations récurrentes dans lesquelles le patron est appliqué.
  - Problème : ensemble de forces (objectifs et contraintes) qui ont lieu dans ce contexte.
  - Solution : modèle de conception que l'on peut appliquer pour résoudre ces forces.
  - Conséquences: les résultats d'utilisation du patron.

Les patrons de conception

- La GoF (Gang of Four) distingue trois familles de patrons
  - Les patrons créateurs: concernent la création de classes ou d'objets;
  - Les patrons structurels: s'intéressent à la composition d'objets ou classes pour réaliser de nouvelles fonctionnalités;
  - Les patrons comportementaux: concernent les interactions entre classes et l'affectation des responsabilités;

Les patrons de conception

- Connaissez vous d'autres patrons?
- L'encapsulation peut-elle être vue comme un patron?