이터레이터와 컴포지트 패턴

이관우

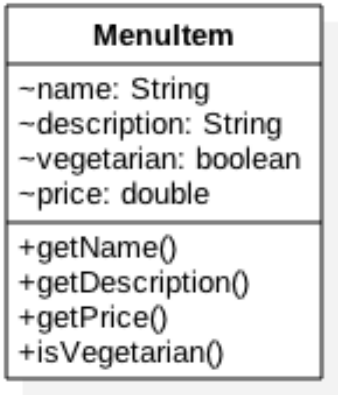
kwlee@hansung.ac.kr

학습 목표

- 이터레이터 패턴이 다루는 문제를 이해.
- 이터레이터 적용 및 개선.
- 이터레이터 패턴의 정의 및 디자인 원칙
- 컴포지트 패턴이 필요한 상황 이해
- 컴포지트 패턴 정의 및 적용

이터레이터 패턴이 다루는 문제를 이해

- *디너* 와 *팬케이크 하우스*의 합병...
  - 아침에는 *팬케이크 하우스* 메뉴를, 점심에는 *디너* 메뉴를 사용
  - 두 식당의 메뉴 항목 구현에는 합의됨



  - 두 식당의 메뉴 구현 방식에는 합의가 되지 않음
    - PancakeHouseMenu → ArrayList 사용
    - DinerMenu → 배열 사용

DinerMenu 구현

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];

        addItem("Vegetarian BLT",
                "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 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;
        }
    }

    public MenuItem[] getMenuItems() {
        return menuItems;
    }
….
```

PancakeHouseMenu 구현

```
public class PancakeHouseMenu {
    ArrayList<MenuItem> menuItems;

    public PancakeHouseMenu() {
            menuItems = new ArrayList<MenuItem>();
            addItem("K&B's Pancake Breakfast",
                                "Pancakes with scrambled eggs, and toast",
                                true,
                                2.99);
            ....
    }

    public void addItem(String name, String description, boolean vegetarian, double price) {
            MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
            menuItems.add(menuItem);
    }

    public ArrayList<MenuItem> getMenuItems() {
            return menuItems;
    }
....
```

메뉴를 사용하는 클라이언트: Waitress

- printMenu()
  - 메뉴에 있는 모든 항목을 출력
- printBreakfastMenu()
  - 아침 식사 항목만 출력
- printLunchMenu()
  - 점심 식사 항목만 출력
- printVegetarianMenu()
  - 채식주의자용 메뉴 항목만 출력
- isItemVergetarian(name)
  - 해당 항목이 채식주의자용이면 true를 리턴하고 그렇지 않으면 false를 리턴

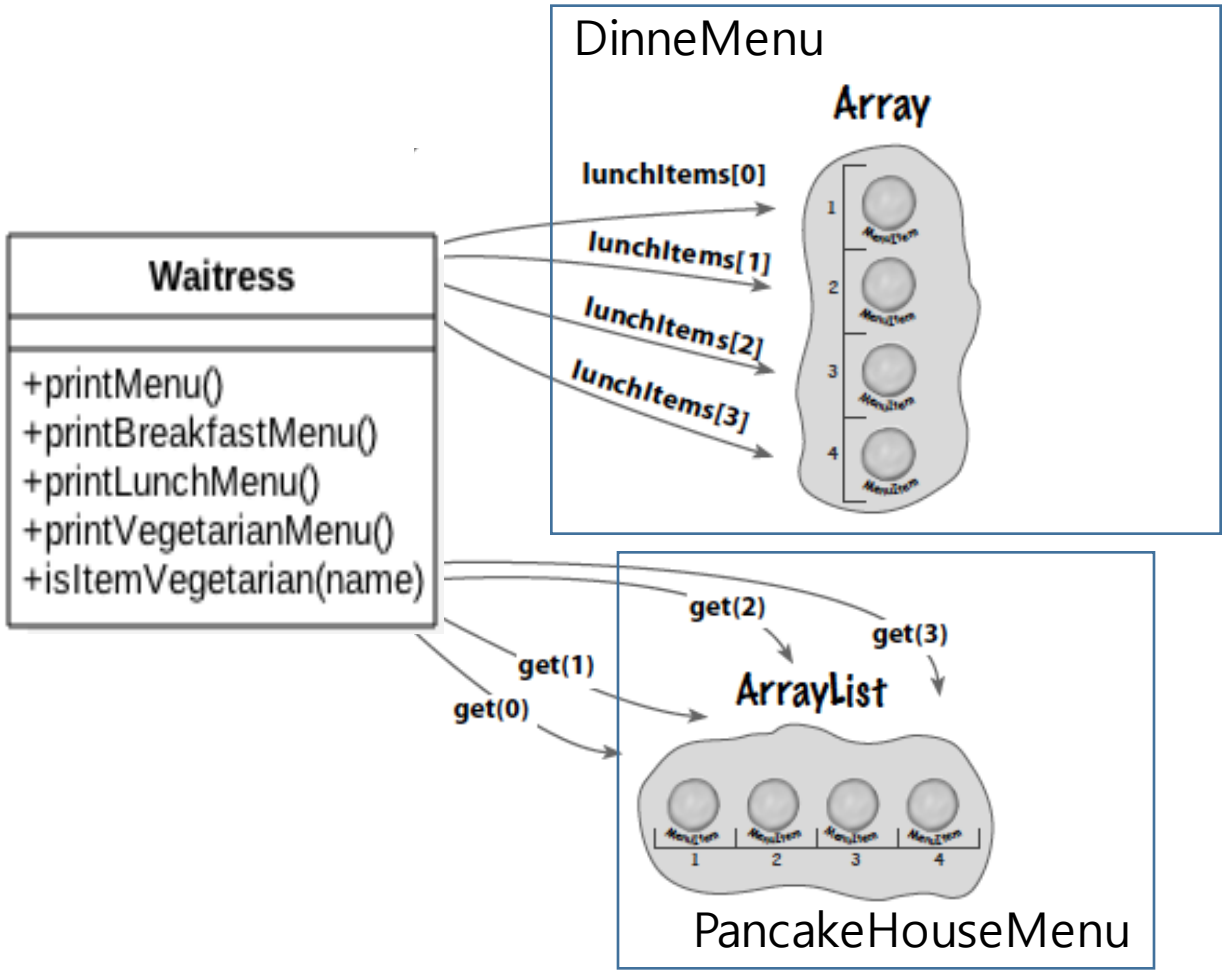
printMenu() 메소드

```
public void printMenu() {
        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.print(menuItem.getName()+"  ");
                System.out.println(menuItem.getPrice()+"  ");
                System.out.println(menuItem.getDescription());
        }

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

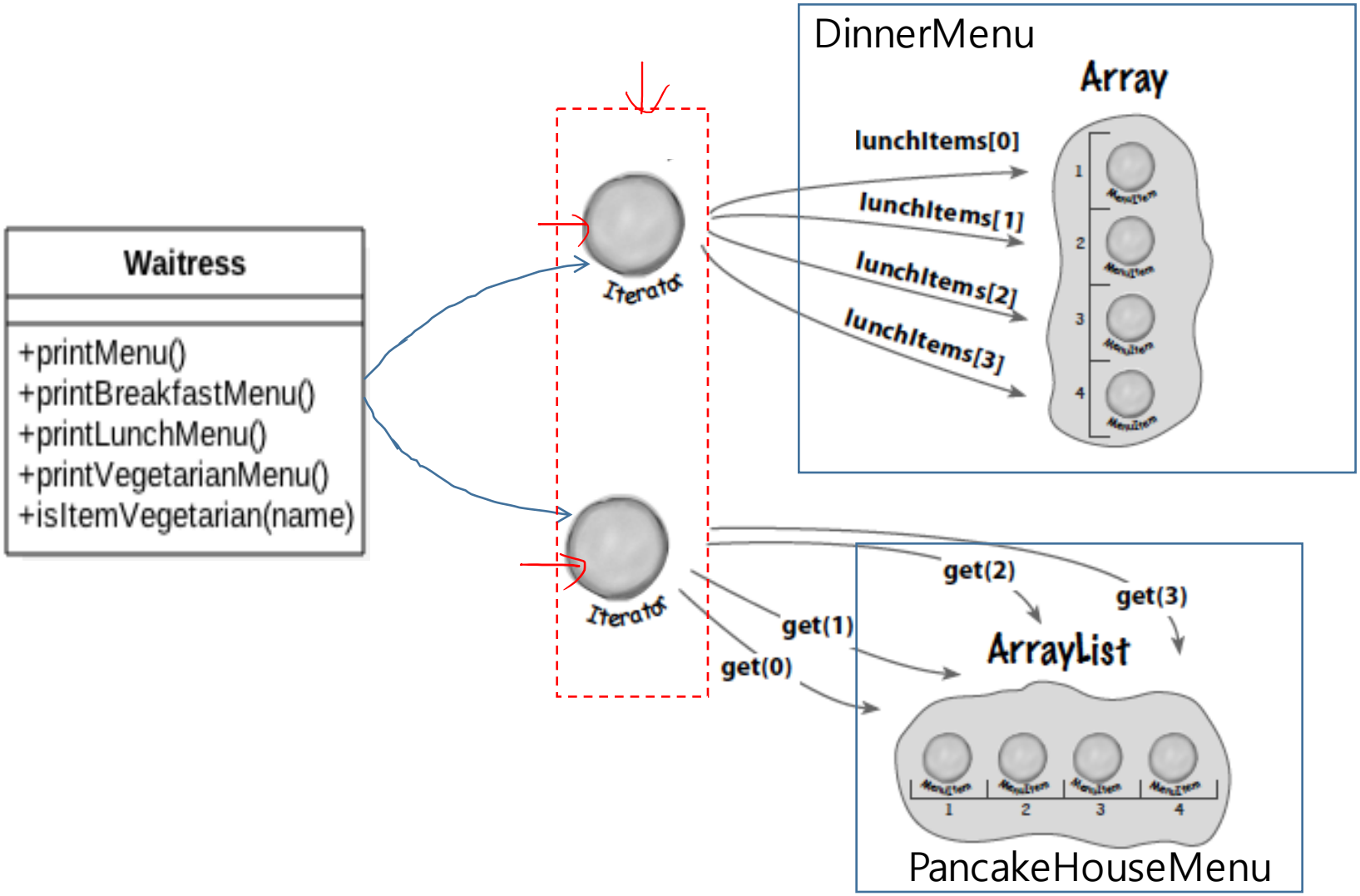
메소드 이름은 동일 하지만, 반환되는 컬렉션이 다름

컬렉션의 차이로 인해, 반복작업을 하는 구현이 다름

어떻게 컬렉션에 대한 반복작업을 처리하는 방법을 캡슐화할 수 있을까요?



반복작업 구현의 캡슐화


DinnerMenu
Array
lunchItems[0]
lunchItems[1]
lunchItems[2]
lunchItems[3]
1
2
3
4
MenuItem
Waitress
+printMenu()
+printBreakfastMenu()
+printLunchMenu()
+printVegetarianMenu()
+isItemVegetarian(name)
Iterator
Iterator
get(2)
get(3)
get(1)
get(0)
ArrayList
1
2
3
4
PancakeHouseMenu

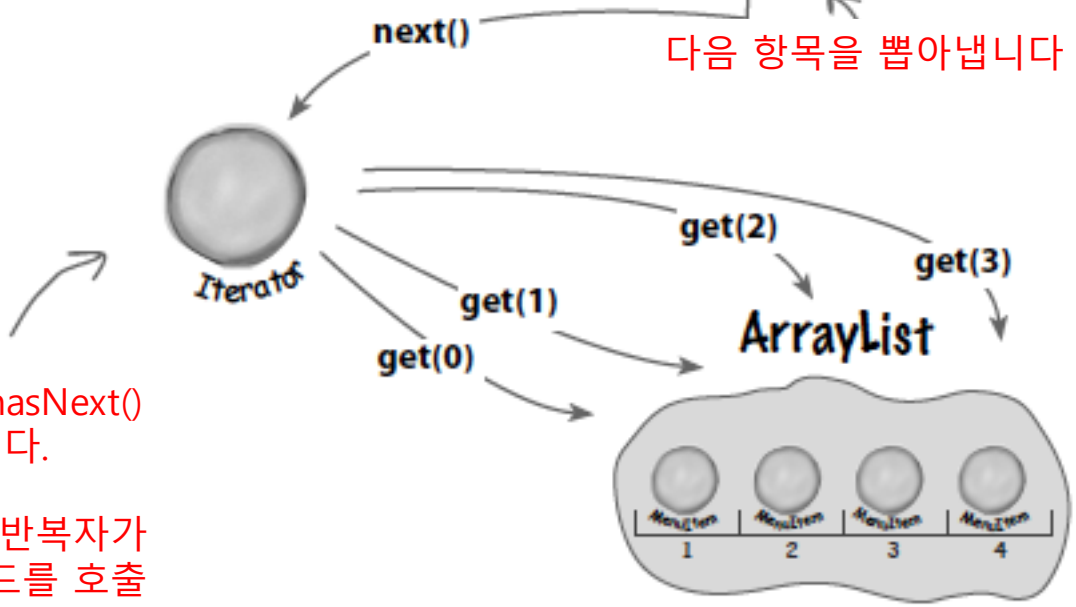
ArrayList에 대한 반복작업 구현의 캡슐화

```
Iterator iterator = breakfastMenu.createIterator()

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

breakfastMenu에 들어있는 MenuItem들에 대한 반복자(Iterator 객체)요청.

아직 항목이 남아 있는 동안...

다음 항목을 뽑아냅니다



클라이언트에서 방금 hasNext()와 next()를 호출했습니다.

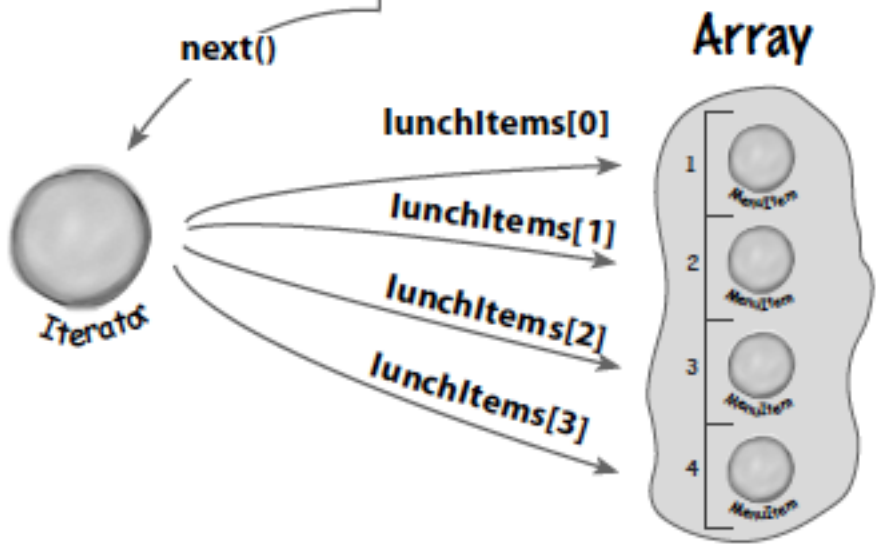
그러면 무대 뒤에서는 반복자가 ArrayList의 get() 메소드를 호출하죠.

배열에 대한 반복작업 구현의 캡슐화

```
Iterator iterator = lunchMenu.createIterator();

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

이 코드는 breakfastMenu 코드와 완전히 동일하군요

여기서도 마찬가지로, 클라이언트에서 hasNext()와 next()를 호출하면,
무대 뒤에서는 반복자가 배열의 인덱스를 써서 MenuItem 항목을 꺼내죠.



DinerMenu에서 Iterator 인터페이스 구현

```
public class DinerMenuIterator implements Iterator {
        MenuItem[] items;
        int position = 0;

        public DinerMenuIterator(MenuItem[] items) {
                this.items = items;
        }

        public MenuItem 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;
                }
        }
}
```

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

DinerMenu에서 DinerMenuIterator 사용

```
public class DinerMenu {
        static final int MAX_ITEMS = 6;
        int numberOfItems = 0;
        MenuItem[] menuItems;

        public DinerMenu() {  ...          }

        public void addItem(String name, String description,
                        boolean vegetarian, double price) {
                        ...
        }

        public MenuItem[] getMenuItems() {
                return menuItems;
        }

        public Iterator createIterator() {
                return new DinerMenuIterator(menuItems);
        }
        ...
}
```

Waitress 코드 고치기

```
public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();

        System.out.println("MENU ₩n----₩n BREAKFAST");
        printMenu(pancakeIterator);

        System.out.println("₩n  LUNCH");
        printMenu(dinerIterator);

}

private void printMenu(Iterator iterator) {
        while ( iterator.hasNext() ) {
                MenuItem menuItem = iterator.next();
                System.out.print(menuItem.getName()  + ", ");
                System.out.print(menuItem.getPrice()  + " -- ");
                System.out.println(menuItem.getDescription());
        }
}
```

정리...

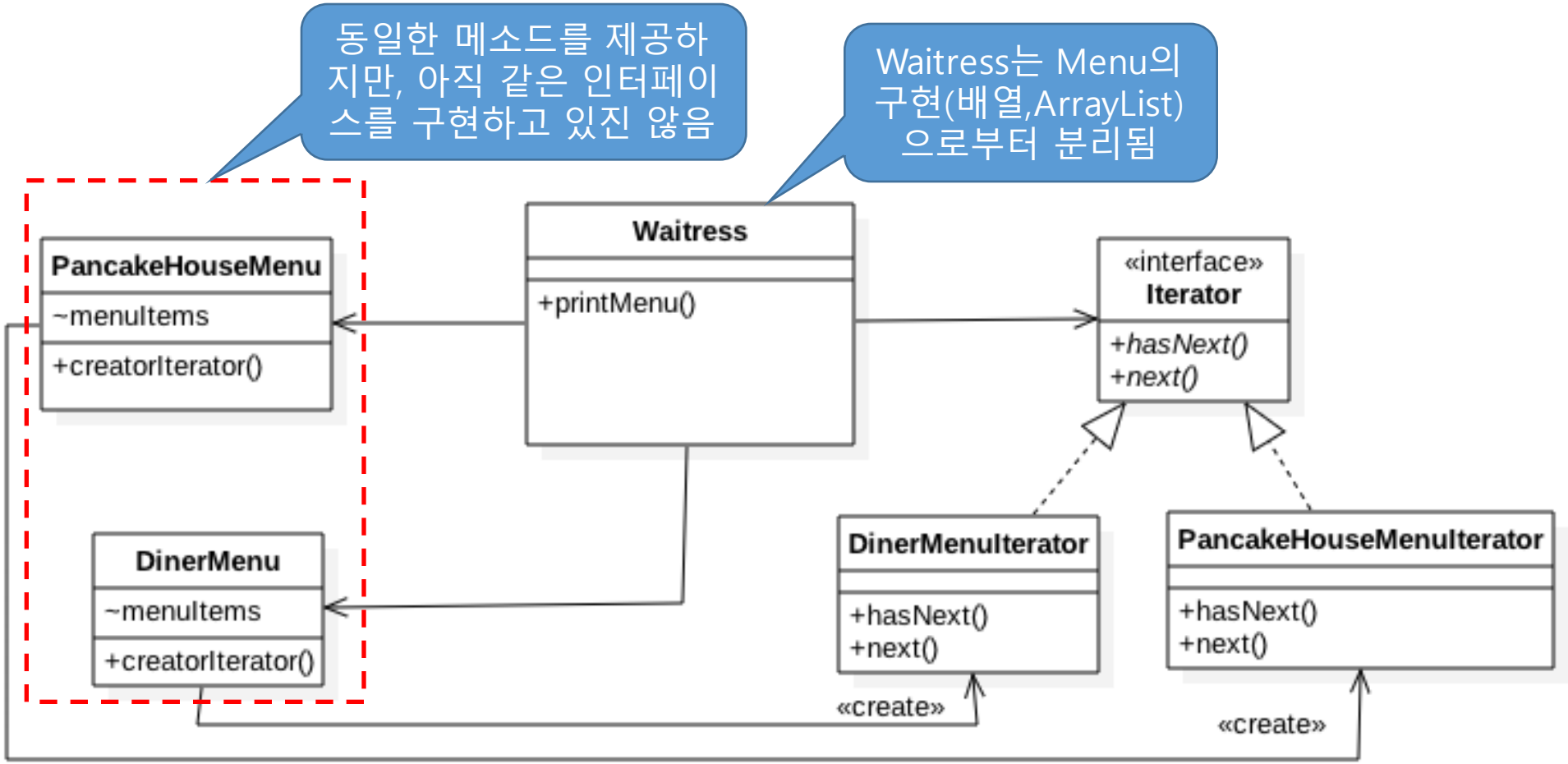
관리하기 힘든 Waitress 코드

- 메뉴 구현이 캡슐화 안됨
  - DinerMenu 구현에서 사용된 배열이 다른 클래스에 노출됨
  - Waitress가 구상 클래스 (MenuItem[] 과 ArrayList)에 직접 연결
- Waitress 클래스에서 MenuItems에 대한 반복작업을 처기하기 위해 순환문이 두개 필요
- Waitress는 인터페이스가 거의 똑같은 두 개의 서로 다른 구상 메뉴 클래스에 의존함.

Iterator가 장착된 Waitress 코드

- 메뉴 구현이 캡슐화됨
  - Waitress에서는 인터페이스(Iterator)만 알고 있으면 되고, 메뉴에서 메뉴항목을 어떤 식으로 저장하는지 전혀 알필요 없음
- Iterator만 구현한다면 어떤 컬렉션이든 다형성을 활용하여 한 개의 순환문으로 처리할 수 있음
- Waitress는 여전히 두 개의 구상 메뉴 클래스에 의존합니다. 이는 수정될 필요가 있습니다.

현재 디자인



https://github.com/kwanulee/DesignPattern/tree/master/iterator/dinermerger

현재 디자인

Waitress 클래스는 구상 클래스 (PancakeHouseMenu, DinerMenu)에 의존적임

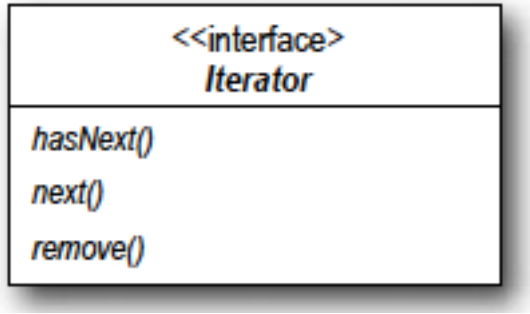
```
public class Waitress {
        PancakeHouseMenu pancakeHouseMenu;
        DinerMenu dinerMenu;

        public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
                this.pancakeHouseMenu = pancakeHouseMenu;
                this.dinerMenu = dinerMenu;
        }

        public void printMenu() {
                Iterator pancakeIterator = pancakeHouseMenu.createIterator();
                Iterator dinerIterator = dinerMenu.createIterator();

                System.out.println("MENU₩n----₩nBREAKFAST");
                printMenu(pancakeIterator);
                System.out.println("₩nLUNCH");
                printMenu(dinerIterator);
        }

        private void printMenu(Iterator<MenuItem> iterator) {
                while (iterator.hasNext()) {
                        MenuItem menuItem = iterator.next();
                        System.out.print(menuItem.getName() + ", ");
                        …
                }
        }
```

동일함

https://github.com/kwanulee/PatternExample/blob/master/iterator/dinermerger/src/hansung/designpatterns/iterator/dinermerger/Waitress.java

Java의 Iterator 인터페이스 사용

- ArrayList는 반복자를 리턴하는 iterator() 메소드 제공
  → PancakeHouseMenuIterator는 구현할 필요 없음




```
import java.util.Iterator;

public class DinerMenuIterator implements Iterator<MenuItem> {
    ...
    public MenuItem next() { ... }
    public boolean hasNext() { ... }

    public void remove() {
            if (position <= 0)
                    throw new IllegalStateException ("You can't remove an item ...)");
            if (list[position-1] != null) {
                    for (int i = position-1; i < (list.length-1); i++)
                            list[i] = list[i+1];
                    list[list.length-1]  = null;
            }
    }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/iterator/dinermergeri/src/hansung/designpatterns/iterator/dinermergeri/DinerMenuIterator.java

현재 디자인 개선...

- PancakeHouseMenu와 DinerMenu 인터페이스 통일

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

```
public class PancakeHouseMenu implements Menu {
...
    public Iterator<MenuItem> createIterator() {
        return menuItems.iterator();
    }
}
```

```
public class DinerMenu implements Menu {
    ...
    public Iterator<MenuItem> createIterator() {
        return new DinerMenuIterator(menuItems);
    }
}
```

현재 디자인 개선...

Waitress 클래스는 더이상 메뉴 구상 클래스 (PancakeHouseMenu, DinerMenu)에 의존적이지 않음

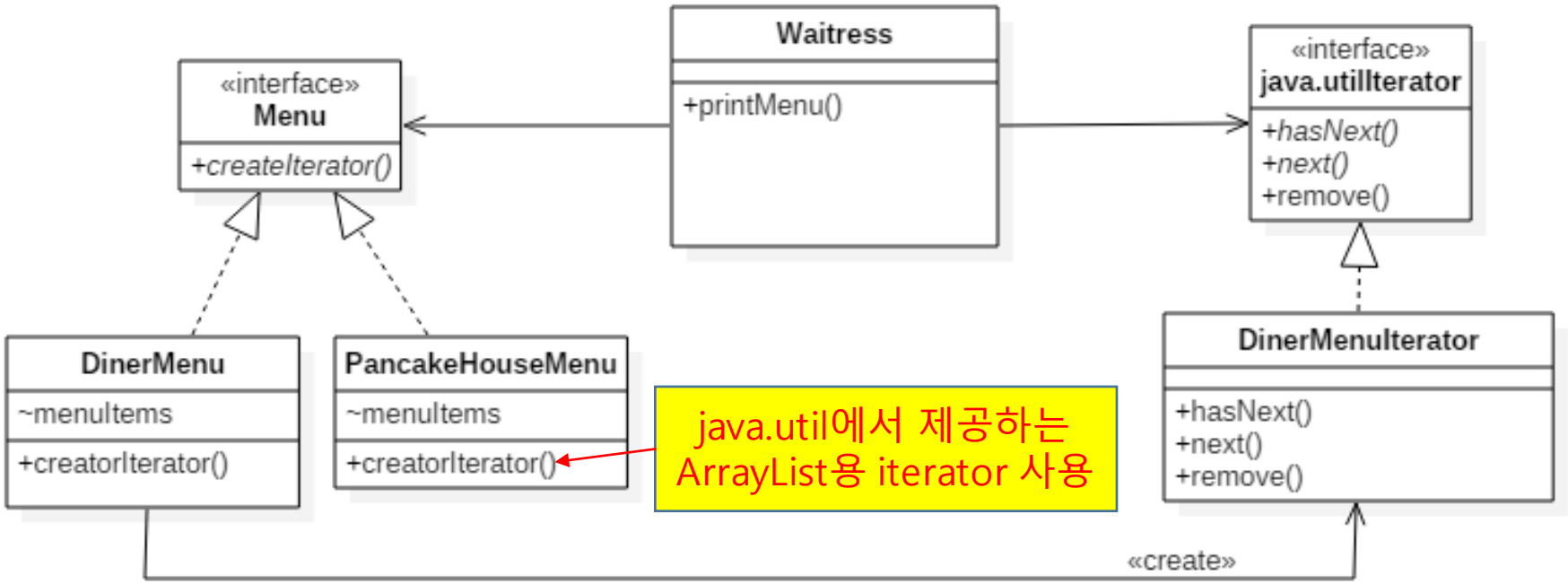
```
import java.util.Iterator;

public class Waitress {
        Menu pancakeHouseMenu;
        Menu dinerMenu;

        public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {
                this.pancakeHouseMenu = pancakeHouseMenu;
                this.dinerMenu = dinerMenu;
        }

        public void printMenu() {
                Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
                Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();

                System.out.println("MENU₩n----₩nBREAKFAST");
                printMenu(pancakeIterator);
                System.out.println("₩nLUNCH");
                printMenu(dinerIterator);
        }

        private void printMenu(Iterator<MenuItem> iterator) {
                while (iterator.hasNext()) {
                        MenuItem menuItem = iterator.next();
                        System.out.print(menuItem.getName() + ", ");
                        ...
                }
        }
```

동일함

https://github.com/kwanulee/PatternExample/blob/master/iterator/dinermergeri/src/hansung/designpatterns/iterator/dinermergeri/Waitress.java

정리..

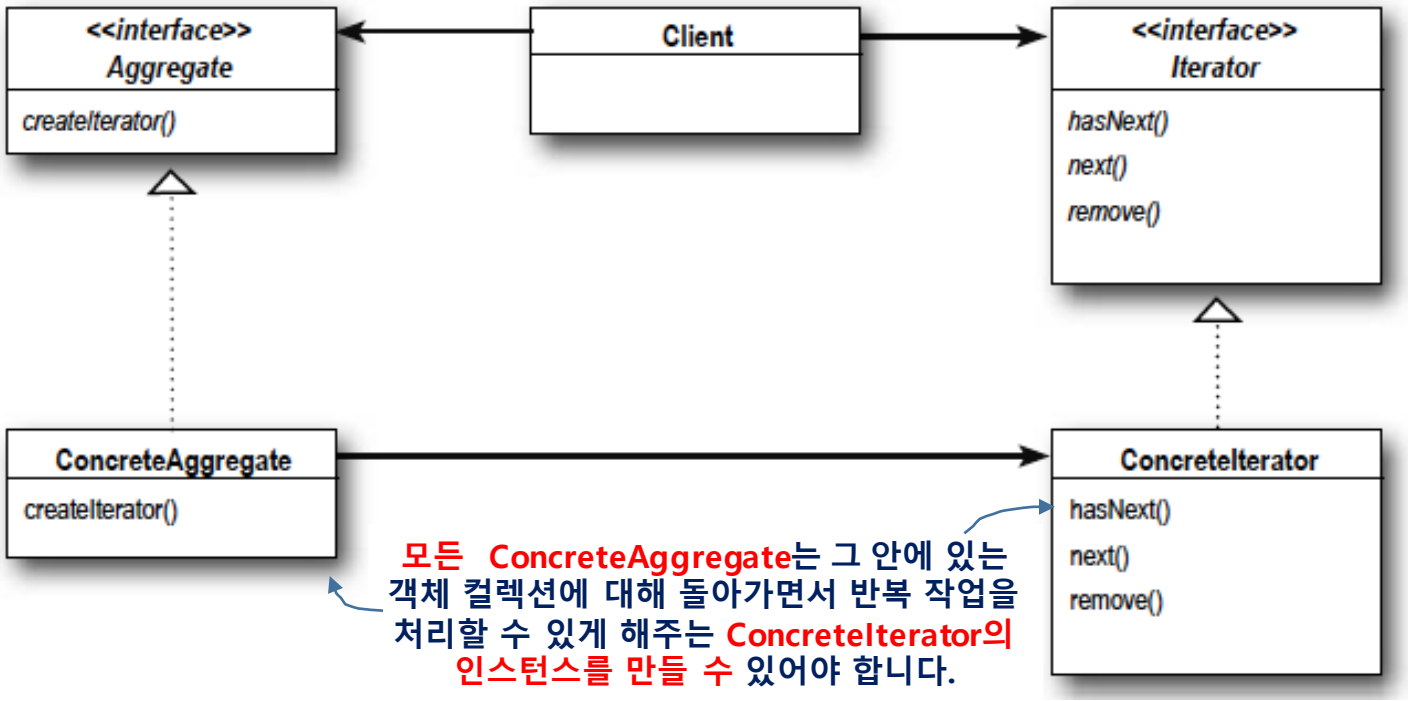
"특정 구현이 아닌 인터페이스에 맞춰서 프로그램"



https://github.com/kwanulee/DesignPattern/tree/master/iterator/dinermergeri

이터레이터 패턴 (반복 작업의 캡슐화)

- 정의
  - 이터레이터 패턴은 컬랙션 구현 방법을 노출시키지 않으면서도 그 집합체 안에 들어 있는 모든 항목에 접근할 수 있게 해 주는 방법을 제공해 줍니다.



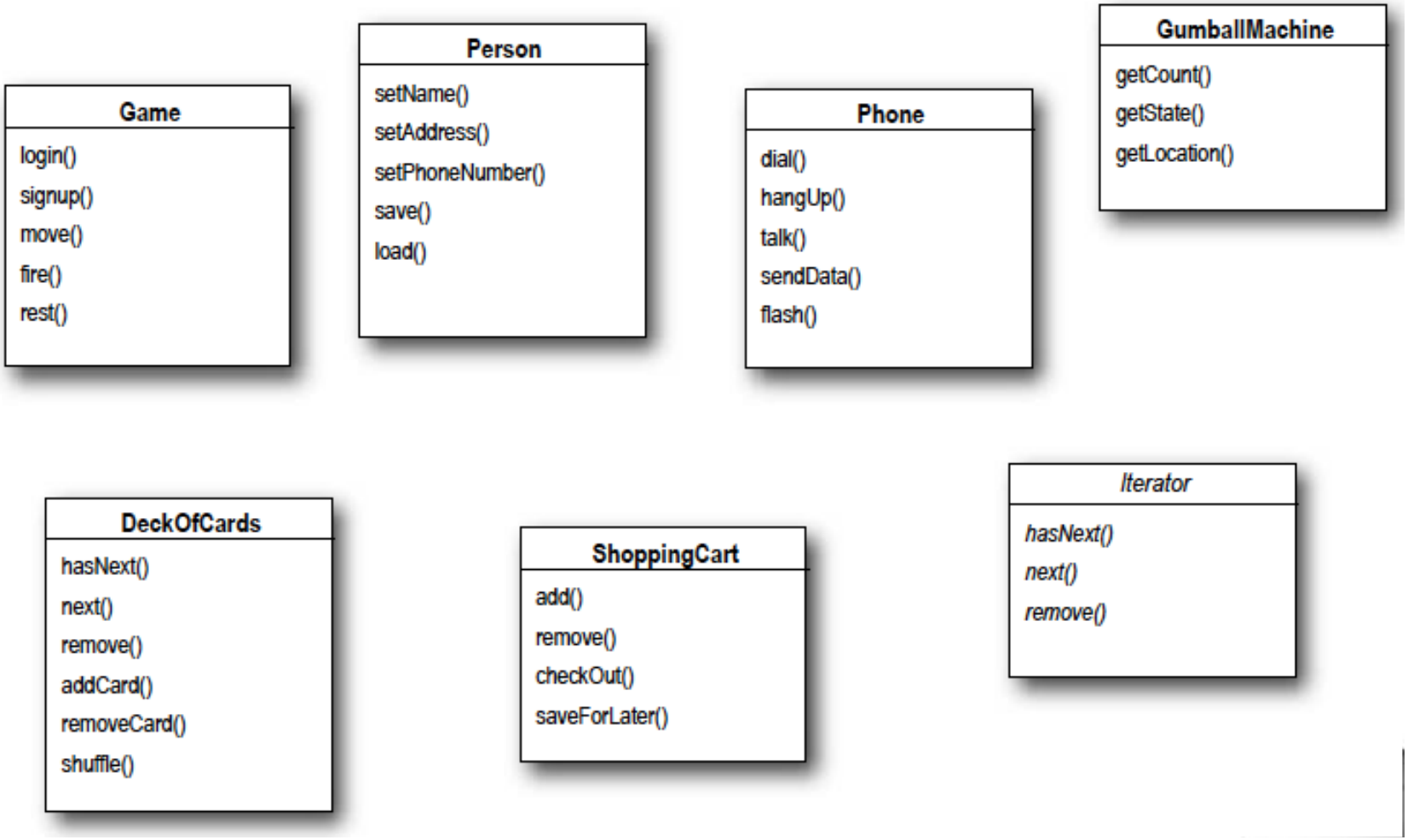
디자인 원칙 : 단일 책임 (Single Responsibility)

> 클래스를 변경시키는 이유는 한 가지 뿐이어야 한다.

- 클래스에서 맡고 있는 모든 책임 (역할)은 후에 코드 변화를 야기시키는 요인이 될 수 있음
  - 책임 (역할) 이 두 개 이상이면 변경 가능한 요인이 두개 이상임

- 응집도 (Cohesion)
  - 한 클래스 또는 모듈이 특정 목적 또는 역할을 얼마나 일관되게 지원하는 지를 나타내는 척도

다음 클래스들을 살펴보고 여러 역할을 맡고 있는 클래스를 찾아봅시다



새로운 뉴스...

- 객체마을 카페도 합병되어, 카페의 메뉴가 저녁 메뉴로 추가 될 예정입니다...

```
public class CafeMenu {
    HashMap<String, MenuItem> menuItems = new HashMap<String, MenuItem>();

    public CafeMenu() {
            addItem("Veggie Burger and Air Fries",
                    "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
                            true, 3.99);
                    ...
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price) {
            MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
            menuItems.put(menuItem.getName(), menuItem);
    }

    public Map<String, MenuItem> getItems() { return menuItems; }
    ....
```

https://github.com/kwanulee/DesignPattern/blob/master/iterator/dinermergercafe/src/hansung/designpatterns/iterator/dinermergercafe/CafeMenu.java

CafeMenu 고치기

```
public class CafeMenu implements Menu {
    HashMap<String, MenuItem> menuItems = new HashMap<String, MenuItem>();

    public CafeMenu() {
            addItem("Veggie Burger and Air Fries",
                    "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
                            true, 3.99);
                    …
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price) {
            MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
            menuItems.put(menuItem.getName(), menuItem);
    }

    public Map<String, MenuItem> getItems() { return menuItems; }

    public Iterator<MenuItem> createIterator() {
                        return menuItems.values().iterator();
    }
    ….
```

HashMap 객체인 menuItems는 왜 직접 iterator()를 지원하지 않을 까요?

Waitress에 카페 메뉴 추가하기

이슈: 메뉴를 추가할 때마다, Waitress 클래스에 코드가 추가되어야 함

```
public class Waitress {
        Menu pancakeHouseMenu;
        Menu dinerMenu;
        Menu cafeMenu;

        public Waitress(Menu pancakeHouseMenu, Menu dinerMenu, Menu cafeMenu) {
                this.pancakeHouseMenu = pancakeHouseMenu;
                this.dinerMenu = dinerMenu;
                this.cafeMenu = cafeMenu;
        }

        public void printMenu() {
                Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
                Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
                Iterator<MenuItem> cafeIterator = cafeMenu.createIterator();

                System.out.println("MENU₩n----₩nBREAKFAST");
                printMenu(pancakeIterator);
                System.out.println("₩nLUNCH");
                printMenu(dinerIterator);
                System.out.println("₩nDINNER");
                printMenu(cafeIterator);
        }
        ....
```

Waitress 코드 개선

```
public class Waitress {
        ArrayList<Menu>  menus;

        public Waitress(ArrayList<Menu> menus) {
                this.menus  =  menus;
        }

        public void printMenu() {
                Iterator<Menu>  menuIterator=  menus.iterator();
                while  (menuIterator.hasNext())  {
                        Menu  menu  =  (Menu)menuIterator.next();
                        printMenu(menu.createIterator());
                }
        }

        private void printMenu(Iterator<MenuItem> iterator) {
                while  (iterator.hasNext()) {
                        MenuItem  menuItem  =  iterator.next();
                        System.out.print(menuItem.getName()  +  ", ");
                        System.out.print(menuItem.getPrice()  +  " -- ");
                        System.out.println(menuItem.getDescription());
                }
        }
        ....
```

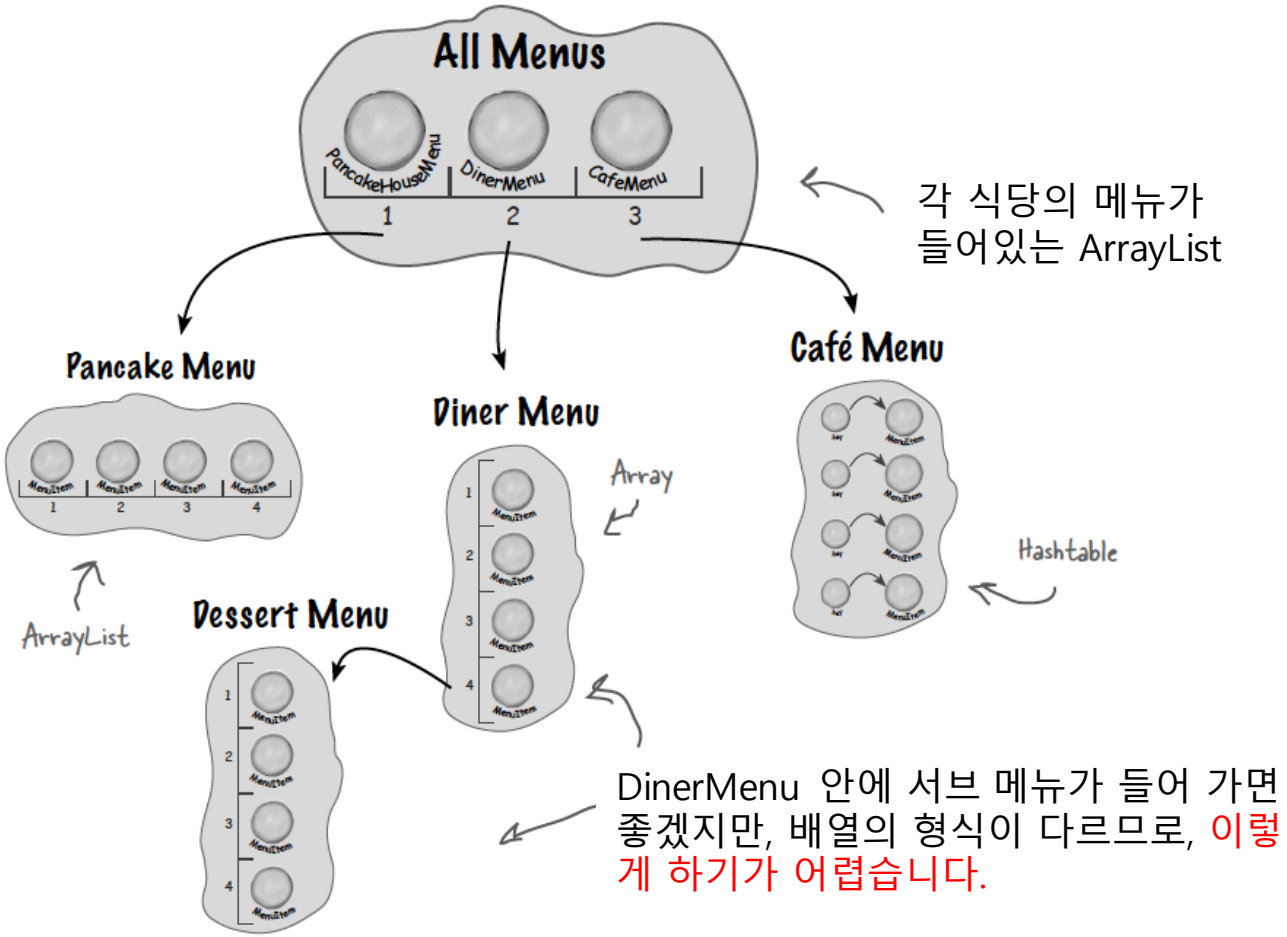
https://github.com/kwanulee/DesignPattern/tree/master/iterator/dinermergercafe

학습 목표

- 이터레이터 패턴이 다루는 문제를 이해.
- 이터레이터 적용 및 개선.
- 이터레이터 패턴의 정의 및 디자인 원칙
- 컴포지트 패턴이 필요한 상황 이해
- 컴포지트 패턴 정의 및 적용
- 컴포지트 패턴과 이터레이터 패턴의 결합-복합 반복자

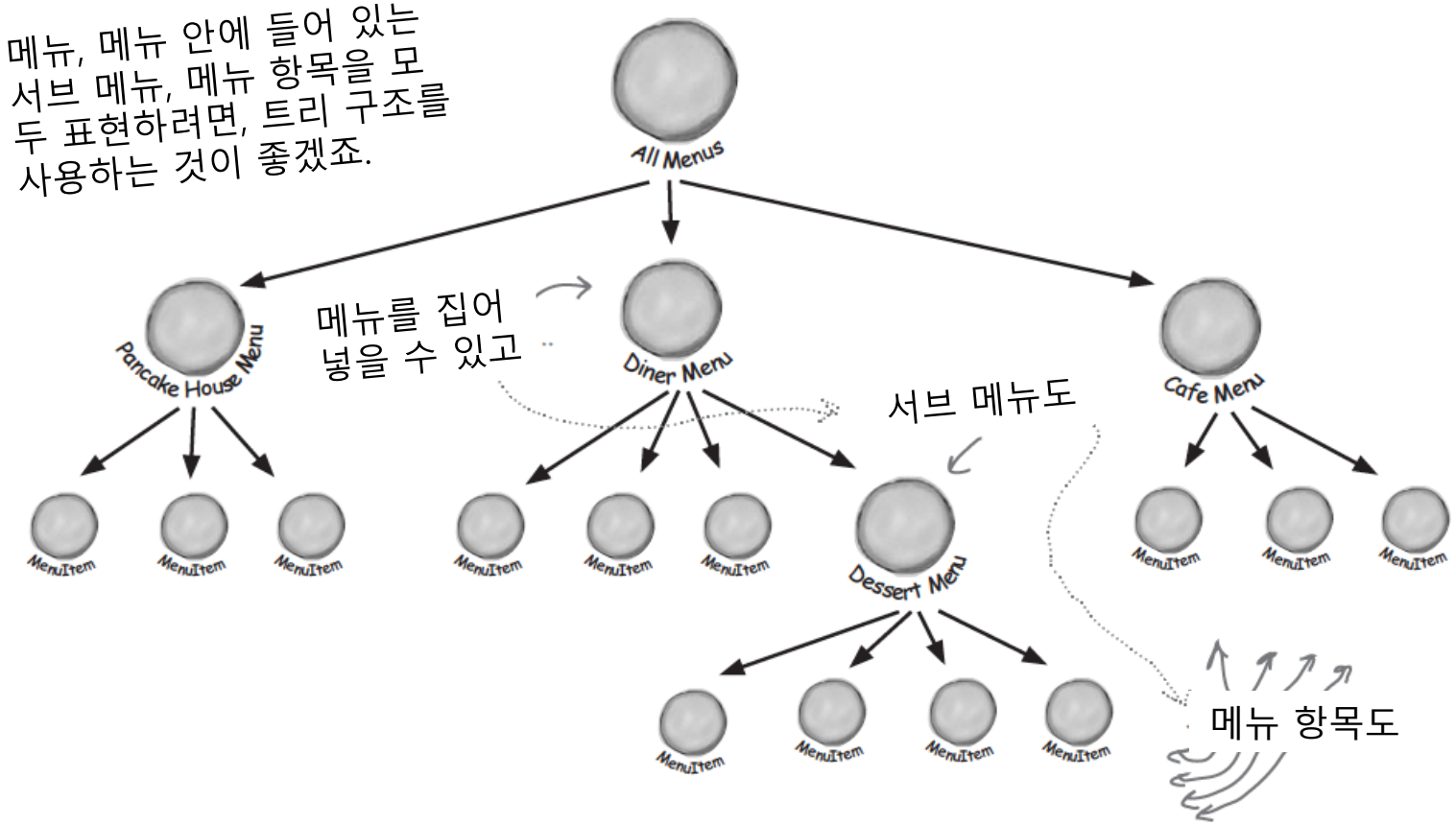
새로운 요구사항

- 디저트 서브 메뉴를 추가



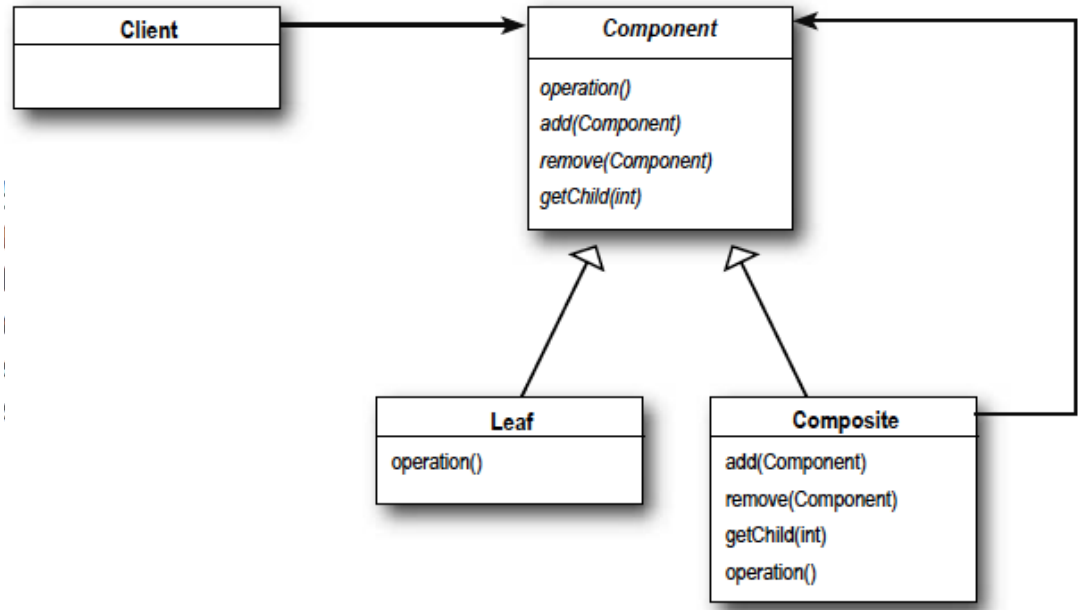
새로운 디자인에서 필요한 것

- 메뉴, 서브메뉴, 메뉴항목 등을 모두 넣을 수 있는 트리 형태의 구조가 필요

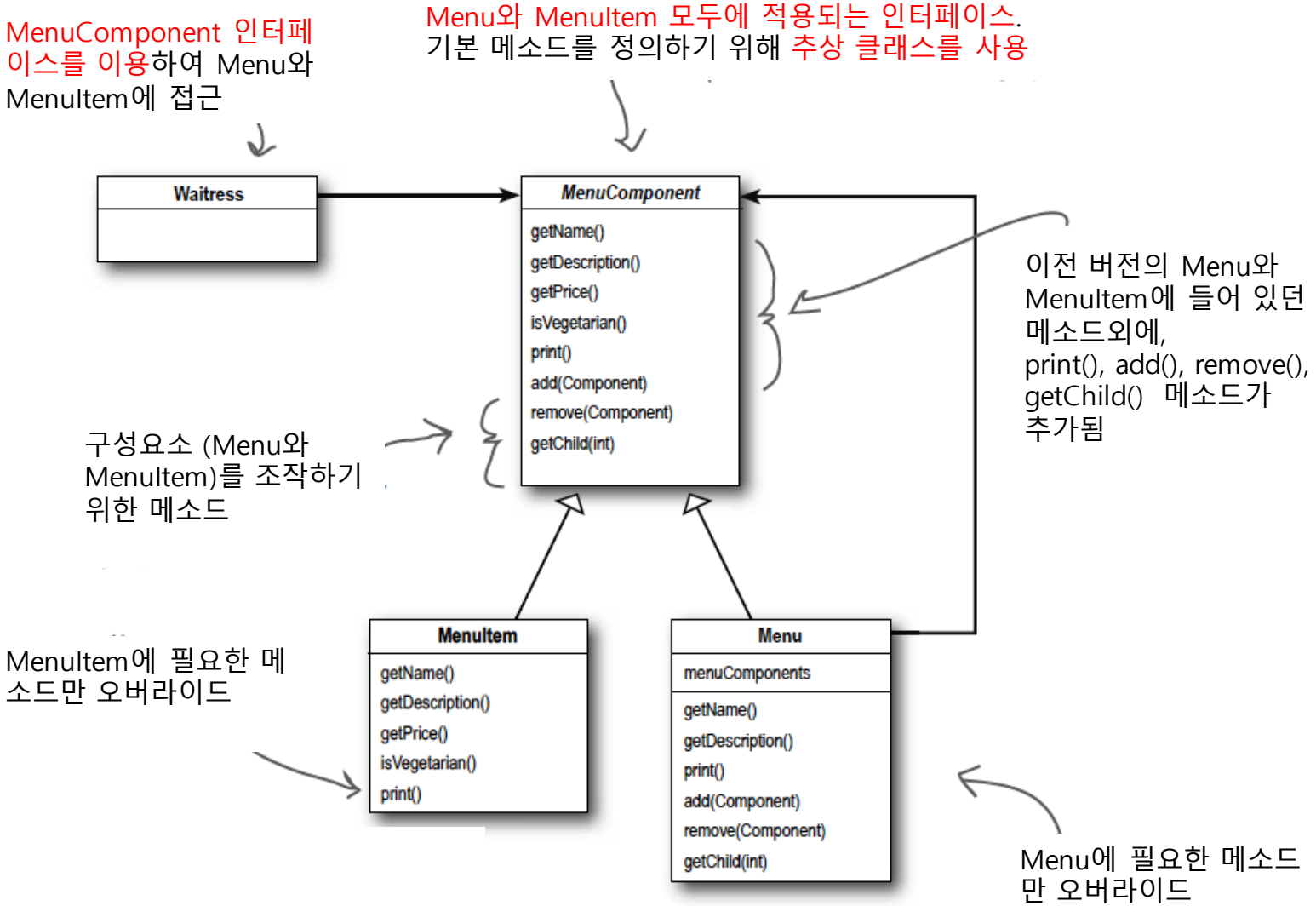


컴포지트 패턴

- 정의
  - 컴포지트 패턴을 이용하면 객체들을 트리 구조로 구성하여, 부분과 전체를 나타내는 계층구조로 만들 수 있습니다.
  - 이 패턴을 이용하면 클라이언트에서 개별 객체와 다른 객체들로 구성된 복합 객체 (composite)를 "동일한 방법"으로 다룰 수 있습니다.



컴포지트 패턴을 이용한 메뉴 디자인



https://github.com/kwanulee/DesignPattern/tree/master/composite/menu

```
public abstract class MenuComponent {
    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }
    public String getName() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrice() {
        throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }
    public void print() {
        throw new UnsupportedOperationException();
    }
}
```

MenuComponent 추가/삭제/가져오기 메소드

MenuItem에서 작업을 처리하기 위해서 사용하는 메소드

Menu와 MenuItem에서 모두 구현하는 메소드

```
public class MenuItem extends MenuComponent {
        String name;
        String description;
        boolean vegetarian;
        double price;

        public MenuItem(String name, String description,
                                boolean vegetarian, double price) {
                this.name = name;
                this.description = description;
                this.vegetarian = vegetarian;
                this.price = price;
        }

        public String getName() { return name; }
        public String getDescription() { return description; }
        public double getPrice() { return price; }
        public boolean isVegetarian() { return vegetarian; }

        public void print() {
                System.out.print("  " + getName());
                if (isVegetarian()) {
                        System.out.print("(v)");
                }
                System.out.println(", " + getPrice());
                System.out.println("     -- " + getDescription());
        }
}
```

기존 MenuItem 구현과 동일

새로운 부분

MenuComponent의 print() 메소드 재정의

메뉴항목 내용 출력

```
public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    String name;
    String description;

    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    public MenuComponent getChild(int i) {
        return (MenuComponent)menuComponents.get(i);
    }

    ...
}
```

MenuComponent 형식의 자식을 몇 개든지 저장할 수 있음

전에는 메뉴마다 다른 클래스를 사용했지만, 메뉴 객체마다 다른 이름과 설명을 설정

MenuItem이나 Menu 모두 MenuComponent이므로 동일하게 추가/삭제/가져오기 를 수행

```
public class Menu extends MenuComponent {
    ...
    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }
```

이름과 설명 리턴

다른 메소드 (getPrice(), isVegitarian())은 메뉴에 어울리지 않는 메소드 이므로, 구현하지 않음

```
    public void print() {
        System.out.print("₩n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("--------------------");

        Iterator<MenuComponent> iterator = menuComponents.iterator();
        while ( iterator.hasNext() ) {
            MenuComponent menuComponent =
                (MenuComponent) iterator.next();
            menuComponent.print();
        }
    }
}
```

print() 메소드에서 Menu에 대한 정보 뿐만아니라 Menu에 들어 있는 모든 MenuItem에 대한 정보까지 출력

Waitress 클래스

```
public class Waitress {
        MenuComponent allMenus;

        public Waitress(MenuComponent allMenus) {
                this.allMenus = allMenus;
        }

        public void printMenu() {
                allMenus.print();
        }
}
```

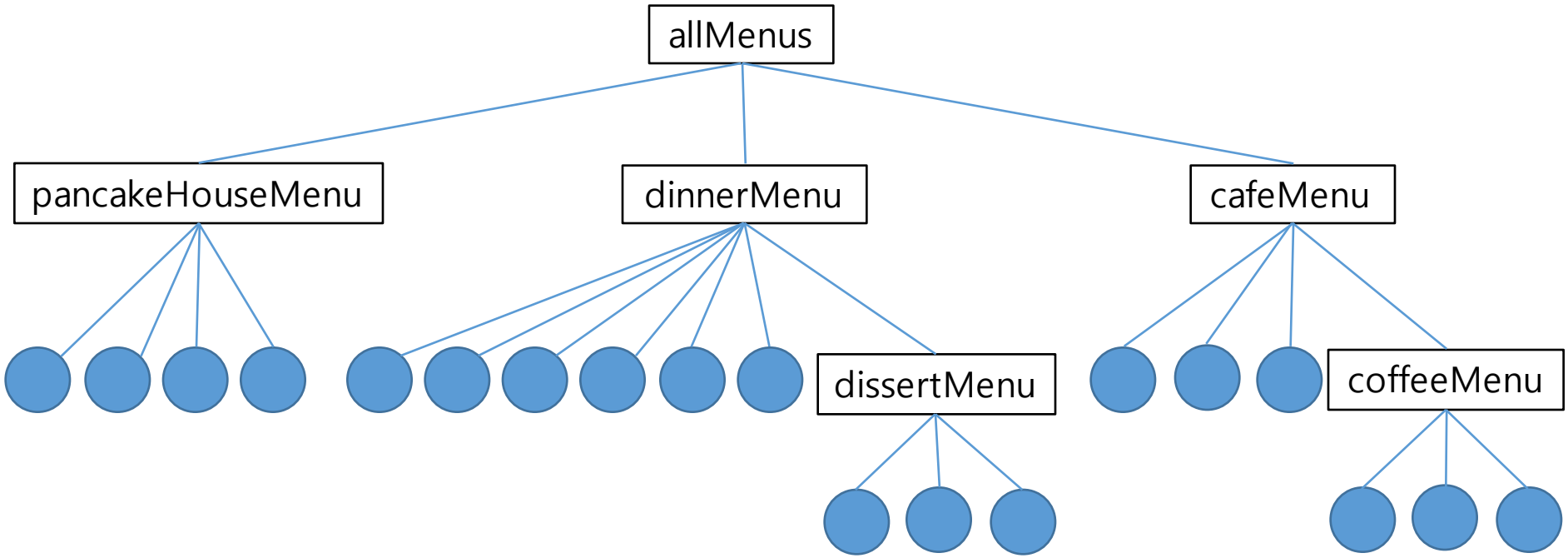
최상위 메뉴 구성요소

메뉴 전체의 계층구조를 출력하고 싶다면,
최상위 메뉴 구성요소의 print() 호출

Composite Pattern 예제 프로젝트

- 프로젝트 코드
  - https://github.com/kwanulee/DesignPattern/tree/master/composite/menu

- 메뉴 구조



핵심정리

- 이터레이터 패턴은 집합체 내부의 구조를 외부로 노출시키지 않고도, 집합체 구성요소들을 접근할 수 있는 방법을 제공
- 이터레이터는 집합체의 항목을 탐색하기 위한 공통의 인터페이스를 제공
- 가능한한 하나의 클래스는 하나의 책임을 담당하도록 설계

- 컴포지트 패턴은 복합 객체와 개별 객체를 클라이언트에서 동일하게 다룰 수 있도록 해줌
- 컴포지트 패턴은 한 클래스(Component)에 두 가지 책임(역할)을 집어 넣고 있습니다. 이러한 설계는 컴포넌트 사용의 투명성을 제공해 주는 반면에 안전성에는 문제가 발생할 수 있습니다.