Iterator Pattern

Contents

The Iterator Pattern

An *aggregate object* is an object that contains other objects for the purpose of grouping those objects as a unit.
It is also called a *container* or a *collection*.
Examples are linked list and hash table.

- Also Known As
  - Cursor
- Purpose
  - Allows for access to the elements of an aggregate object without allowing access to its underlying representation.
- Use When
  - Access to elements is needed without access to the entire representation.
  - Multiple or concurrent traversals of the elements are needed.
  - A uniform interface for traversal is needed.
  - Subtle differences exist between the implementation details of various iterators.

Breaking news…..

- The Objectville Diner and Pancake House have merged!
- Menus must be merged. Owners agree on the implementation of the MenuItems.

```
public class MenuItem {
  String name;
  String description;
  boolean vegetarian;
  double price;
  public MenuItem (String name, String description, boolean vegetarian,
                   double price ) {
     // code here
  }
  // set of getter methods to get access to the fields.
}
```

Implementation of PancakeHouseMenu

```
public class PancakeHouseMenu {
   ArrayList menuItems;
   public PancakeHouseMenu ( ) {
        menuItems = new ArrayList ();
        addItem (“Regular Pancake Breakfast”,
                          “Pancakes with fried eggs, sausage”, false, 2.99);
        addItem (“Blueberry pancakes”,
                       “Pancakes made with fresh blueberries, true, 3.49);
       //other items
   }
   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 methods
}
```

Implementation of DinerMenu

```
public class DinerMenu {
   static final int MAX_ITEMS = 6; int numberOfItems = 0;
   MenuItem[] menuItems;
   public DinerMenu  ( ) {
        menuItems = new MenuItem[MAX_ITEMS];
        addItem (“Vegetarian BLT”, “Bacon with Lettuce & tomato on whole wheat”, true, 2.99);
        addItem (“Soup of the Day”, “Soup of the Day with potato salad”, false, 3.29);
        // other items
   }
   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 any more items”);
        else {
           menuItems[numerOfItems] = menuItem;
           numberOfItems = numberOfItems + 1;
        }
   }
   public MenuItem[] getMenuItems ( ) {
        return menuItems;
   }
   // other methods
}
```

Implementing printMenu()

- It prints every item on the menu

```
PencakeHouseMenu pancakeHouseMenu = new PancackHouseMenu();
ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();
DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();

for (int i = 0; i < breadfirstItems.size(); i++){
   MenuItem menuItem = (MenuItem)breakfastItems.get(i);
   System.out.print (menuItem.getName() + " ");
   System.out.println(menuItem.getPrice() + " ");
   System.out.println(menuItem.getDecription() + " ");
}
for (int i = 0; i < lunchItems.size(); i++){
   MenuItem menuItem = lunchItems[i];
   System.out.print (menuItem.getName() + " ");
   System.out.println(menuItem.getPrice() + " ");
   System.out.println(menuItem.getDecription() + " ");
}
```

Problems

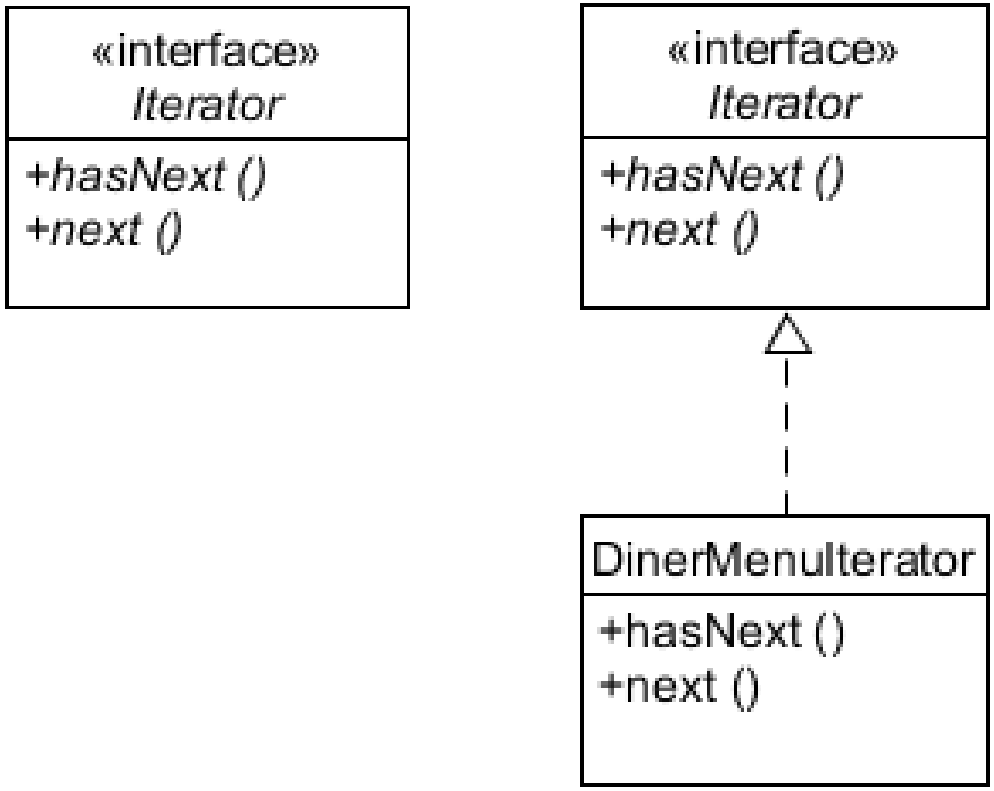
1. We are coding to the concrete implementations, not to an interface
2. If we decided to switch from using DinerMenu to another type of menu implemented with a Hashtable, we'd have to modify a lot of code
3. The waitress needs to know how each menu represents its internal collection of menu items.
4. We have duplicate code: the printMenu()

Two Approaches

```
// The Original Approach
for (int i = 0; i < breakfastItems.size(); i++)
  MenuItem menuItem = (MenuItem) breakfastItems.get(i);

for (int i = 0; i < lunchItems.length; i++)
  MenuItem menuItem = lunchItems[i];
```

```
// The Second Approach using Iterator
Iterator iterator = breakfastMenu.createIterator();
while (iterator.hasNext())
    MenuItem menuItem = (MenuItem) iterator.next();

iterator = lunchMenu.createIterator();
while (iterator.hasNext())
    MenuItem menuItem = (MenuItem) iterator.next();
```

Implementing Iterator Interface



Defining Iterator and DinerMenuIterator

```
public interface Iterator {
  boolean hasNext();
  Object next();
}
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 ++;
        return menuItem;
  }
  public boolean hasNext() {
        if (position >= items.length) return false;
        else return true;
  }
}
```

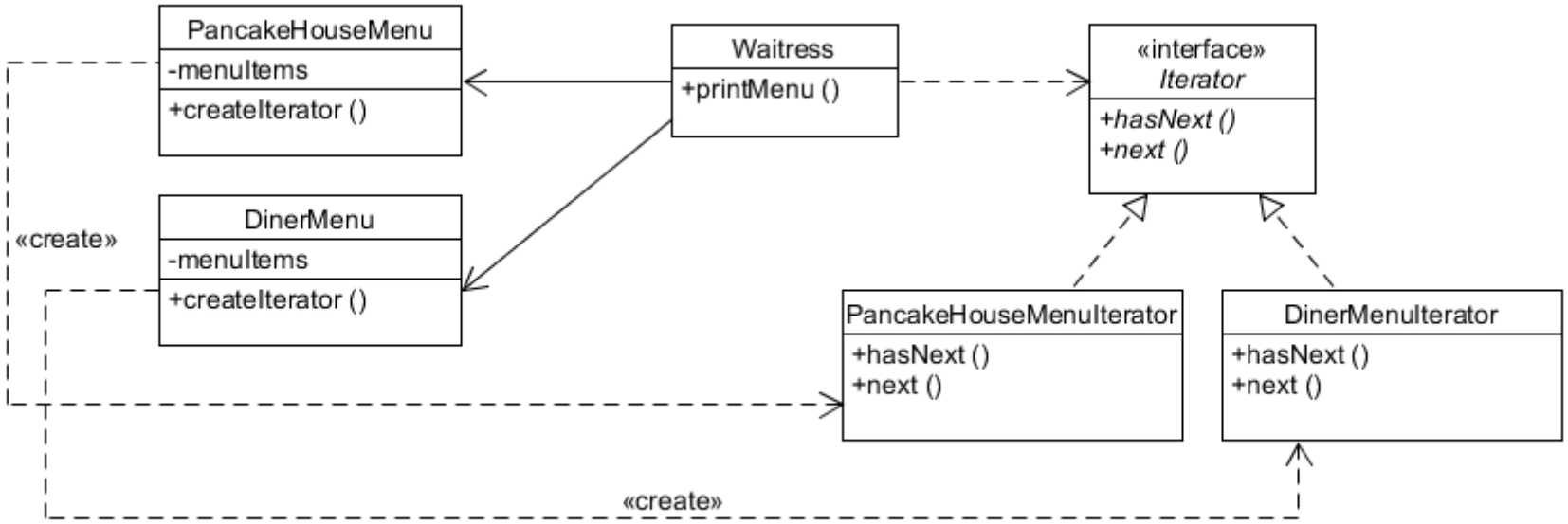
Reworking the DinerMenu with Iterator

```
public class DinerMenu {
  // constructor
  // addItem
  MenuItem[] menuItems;

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

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

Fixing up the Waitress code

```
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 iterator) {
          while (iterator.hasNext()) {
                    MenuItem menuItem = (MenuItem) iterator.next();
                    System.out.println(menuItem.getName() + ", ");
                    ..
          }
   }
}
```

Current Design



Using Java's iterator

- The java.util.Iterator interface:

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

Rewriting DinerMenuIterator

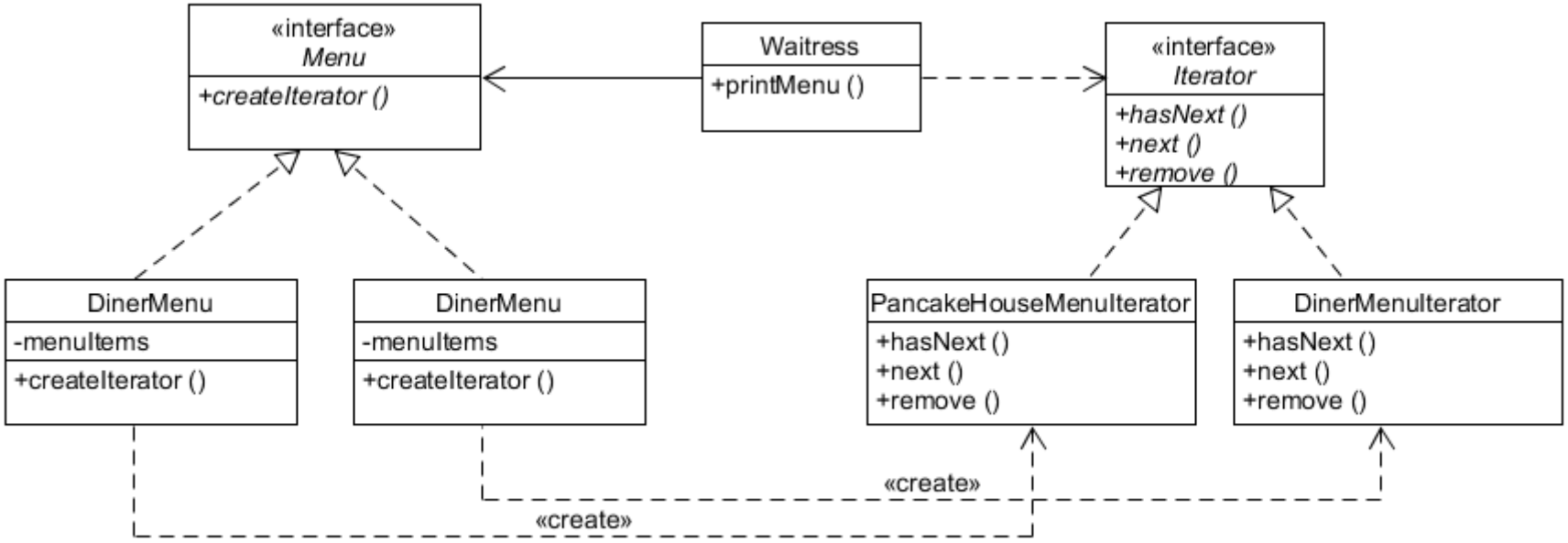
```
import java.util.Iterator;
public class DinerMenuIterator implements Iterator {
  MenuItem[] list;
  public Object next() {
      // the same
  }
  public boolean hasNext() {
      // the same
  }
  public void remove() {
      if (position <=0 )
        throw new IllegalStateException("You can't remove blah blah");

      // code for removing an item and shifting the rest
  }
}
```

Using Menu Interface

```
public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;
    public Waitress(Menu pancakeHouseMenu, Menu 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() {
          while (iterator.hasNext()) {
                  MenuItem menuItem = (MenuItem) iterator.next();
                  System.out.println(menuItem.getName() + ", ");
                  ..
          }
    }
}
```

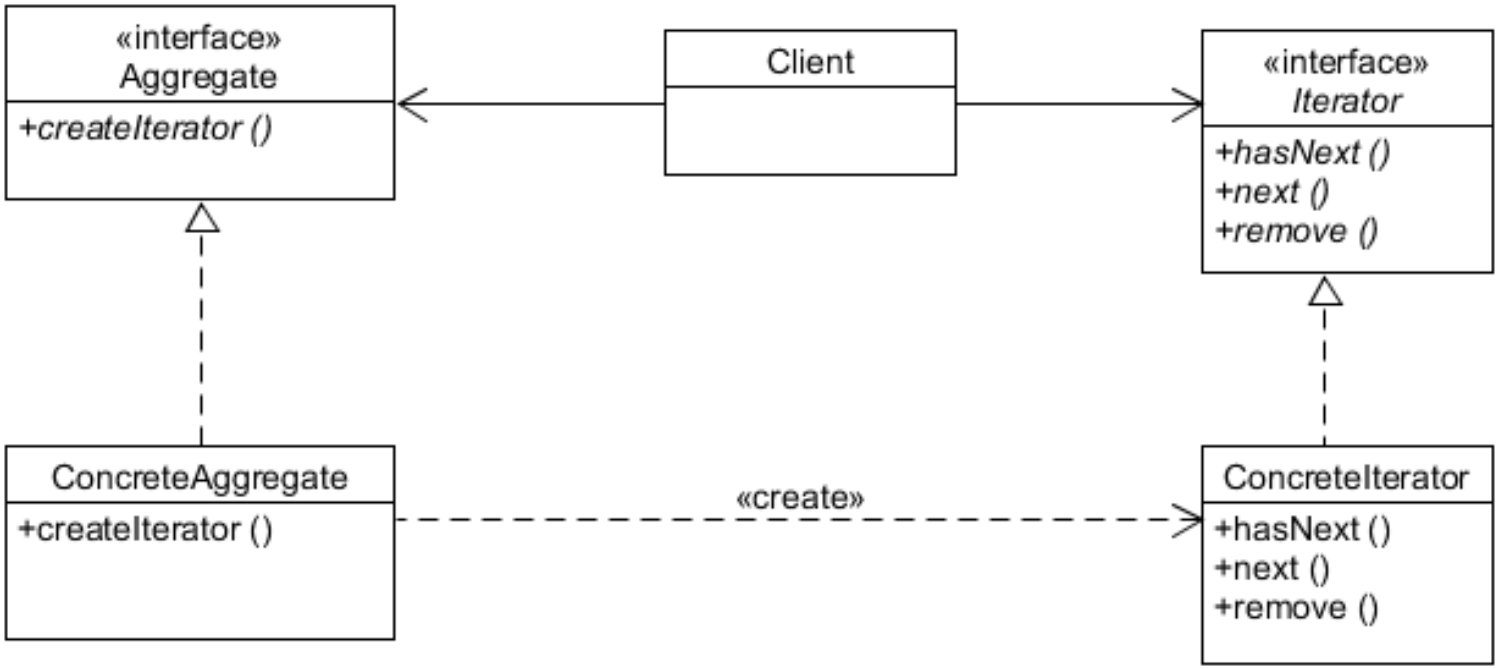
```
public interface Menu {
    public Iterator createIterator();
}
```

New Design


«interface»
Menu
+createIterator ()
Waitress
+printMenu ()
«interface»
Iterator
+hasNext ()
+next ()
+remove ()
DinerMenu
-menuItems
+createIterator ()
DinerMenu
-menuItems
+createIterator ()
PancakeHouseMenuIterator
+hasNext ()
+next ()
+remove ()
DinerMenuIterator
+hasNext ()
+next ()
+remove ()
«create»
«create»

Iterator Pattern

- Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Single Responsibility

- A class should have only one reason to change
  - Aggregate and Iteration – two different jobs
  - Cohesion
    - a measure of how closely a class or a module supports a single purpose or responsibility
    - High cohesion – designed around a set of related functions
    - Low cohesion – designed around a set of unrelated functions

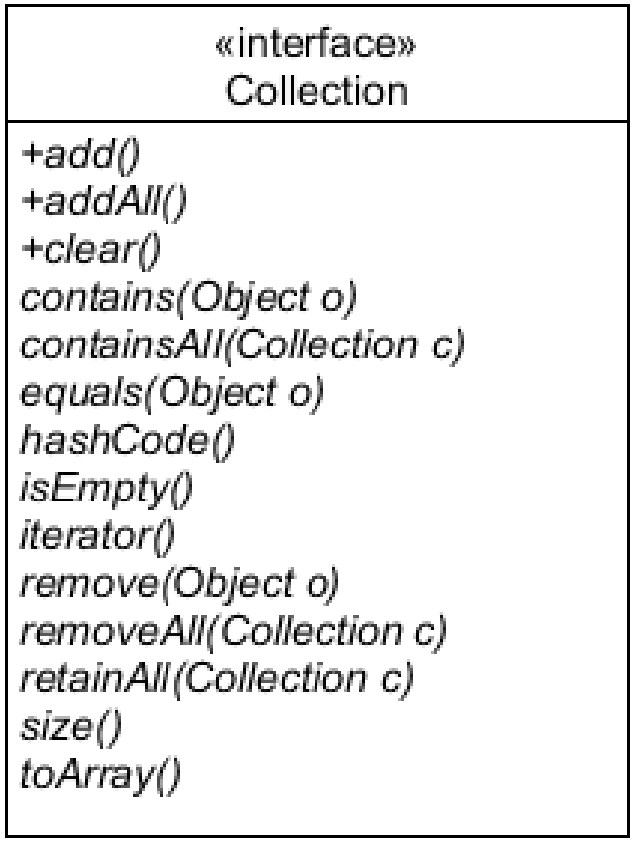
Menu with Hashtable

```
public class CafeMenu implements Menu {
    Hashtable menuItems = new Hashtable();
    public CafeMenu() {
        // constructor
    }
    public void addItem(String name, String desc, boolean vegetarian,
         double price) {
        ..
    }
    public Iterator createIterator() {
         return menuItems.values().iterator();
    }
}
```

Data Structures

- Collections
  - ArrayList, Hashtable, Vector, LinkedList
  - HashTable.values().iterator()
- Built-in Array
  - We implemented iterator for it

Collection Interface



List and ListIterator Interface

```
public interface List extends Collection {
      Object get(int index);
      Object set(int index, Object element);
      void add(int index, Object element);
      Object remove(int index);
      boolean addAll(int index, Collection c);
      int indexOf(Object o);
      int lastIndexOf(Object);
      ListIterator listIterator();
      ListIterator listIterator(int index);
      List subList(int from, int to);
}
```

```
 public interface ListIterator extends Iterator {
      boolean hasNext();
      Object next();
      void remove();
      boolean hasPrevious();
      Object previous();
      int nextIndex();
      int previousIndex();
      void set(Object o);
      void add(Object o);
 }
```

Using ListIterator

```
// Forward traverse
List list = new LinkedList();
...
ListIterator iterator = list.listIterator();
while (iterator.hasNext()) {
     Object item = iterator.next();
     // Code here to process item.
}


// Backward traverse
List list = new LinkedList();
ListIterator iterator = list.listIterator(list.size());
...
while (iterator.hasPrevious()) {
     Object item = iterator.previous();
     // Code here to process item.
}
```

Iterators and Collections in Java5

- For/In (similar to foreach in C#)

```
for (Object obj: collection ) {…}
```

- Example

```
ArrayList items = new ArrayList();
  …
 for (MenuItem item: items)
   System.out.println(“item = “ + item);
```

Related Patterns

- Iterator can traverse a Composite. Visitor can apply an operation over a Composite.
- Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass.
- Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.

Summary

- Single Responsibility Principle
  - A class should have only one reason to change
- Iterator Pattern
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation