Iterator Pattern

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

Outline

After today’s lecture you will be able to:

- Understand the use of the Iterator Design Pattern
- Use and implement the Iterator Pattern

Inspiration

“When someone says, ‘I want a programming language in which I need only say what I want done,’ give him a lollipop.” – Alan Perlis

Collections-Related Design Patterns

- Collections are ubiquitous in programming
  - Lists (Array, Stack, Queue), Hash Table, Trees, ...
- When you use these collections within classes, you are making implementation decisions that need to be encapsulated
  - You don’t want to expose the details of the collections that your class uses to get its job done
    - You are increasing the coupling of your system (perhaps unnecessarily)
    - Since clients are tied to your class and the class of the collection(s)
- Chapter 9 provides details on patterns you can use to encapsulate the details of internal collections; this gives you the freedom to change the collections you use as your requirements change with only minimal impact

The Merging of Two Diners

Two restaurants in Objectville are merging

- One is Lou's breakfast place, the other is Mel's diner
- They want to merge their menus BUT
  - Lou used an `ArrayList` to store menu items
  - Mel used an array to store menu items
    - i.e. `MenuItem[] items`
- Neither person wants to change their implementations, as they have too much existing code that depends on these data structures

Menu Item Implementation - Data Holder

```
public class MenuItem {

  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 String toString() {
    return (name + ", $" + price + "\n   " + descrip
  }
}
```

Lou's Menu (ArrayList)

```
import java.util.ArrayList;

public class PancakeHouseMenu implements Menu {
  ArrayList menuItems;

  public PancakeHouseMenu() {
    menuItems = new ArrayList();

    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 getMenuItems() { // Yuck! Implemen
    return menuItems;
  }

  public String toString() {
    return "Objectville Pancake House Menu";
  }

  // other menu methods here
}
```

Mel’s Menu (Array)

```
public class DinerMenu implements Menu {
  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() { // Yuck! Impleme
    return menuItems;
  }

  // other menu methods here
}
```

Pros and Cons

- Use of ArrayList
  - Easy to add and remove items
  - No management of the “size” of the list
  - Can use a lot of memory if menu items allowed to grow unchecked
  - (Ignoring Java 1.5 generics) Items in `ArrayList` are untyped, need to cast returned objects when retrieving them from the collection
- Use of plain array
  - Fixed size of array provides control over memory usage
  - Items are stored with their types intact, no need to perform a cast on retrieval
  - Need to add code that manages the bounds of the array (kind of silly for programmers living in the 21st century!)

But... Implementation Details Exposed

- Both menu classes reveal the type of their collection via the `getMenuItems()` method
  - All client code is now forced to bind to the menu class and the collection class being used
  - If you needed to change the internal collection, you wouldn't be able to do it without impacting all of your clients

Example Client: Server

- Implement a client of these two menus with the following specification
  - `printMenu()`: print all menu items from both menus
  - `printBreakfastMenu()`: print all breakfast items
  - `printLunchMenu()`: print all lunch items
  - `printVegetarianMenu()`: print all vegetarian menu items
- All of these methods will be implemented in a similar fashion, lets look at `printMenu()`

First: Retrieve the Menus

```
...

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

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

...
```

Simple, but its annoying to have to use two different types to sore the menu items; imagine if we needed to add a third or fourth menu?

Second: Loop Through Items and Print

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

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

Note: differences in checking size and retrieving items; having to cast items retrieved from the `ArrayList`

Again, think of the impact of having to add a third or fourth list!

Design-Related Problems

- Just to emphasize, the current approach has the following design-related problems
  - Coding to an implementation rather than an interface
  - Vulnerable to changes in the collections used
  - `Server` knows the internal details of each `Menu` class
  - We have (in essence) duplicated code, one distinct loop per menu
- Lets solve this problem with our design principle “Encapsulate What Varies”
  - In this case, the details of the data structures and iteration over them

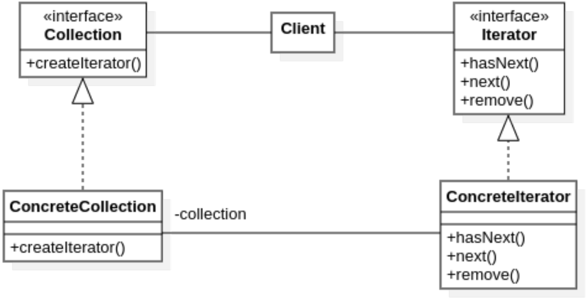
Design of Iterator

- In order to encapsulate the iteration over the data structures, we need to abstract two jobs
  - "Determine if we need to keep iterating" and "get the next item"
- Create Iterator interface with these methods
  - `hasNext()` (return `boolean`) and `next()` (return `Object`)
  - Since `next()` returns an `Object`, we'll have to cast retrieved objects
- Create two implementations of the `Iterator` interface, one for each menu
- Update menu classes to return an iterator
- Update client code (`Server`) to retrieve iterators and loop over them

Iterator Pattern: Definition

- The Iterator Pattern provides a way to access the elements of a collection sequentially without exposing the collection's underlying representation
  - It also places the task of traversal on the iterator object, NOT on the collection: this simplifies the interface of the collection and places the responsibility of traversal where it should be
- Big benefit of Iterator is that it gives you a **uniform interface** for iterating over all your collections that behave polymophically
  - Each iterator knows the details of its underlying collection and "does the right thing" as you access the `hasNext()` and `next()` methods
    - It doesn't matter if the collections are trees, lists, hash tables, or on a different machine (remote iterator anyone?)

Iterator Pattern: Structure



Each concrete collection is responsible for creating a concrete iterator. The `ConcreteIterator` has an association back to the original collection and keeps track of its progress through that collection.

Typically, the behavior of multiple iterators over a single collection is supported as long as the collection is not modified during those traversals.

Return of Single Responsibility

- Remember the Single Responsibility Principle?
  - "A class should have only one reason to change"
- SRP is behind the idea that collections should not implement traversals
  - Collections focus on storing members and providing access to them
  - An iterator focuses on looping over the members of a collection
    - For trees you can have multiple iterators, one each for in-order, pre-order, and post-order traversals, and perhaps others that iterate over members that match a given criteria
    - Similar to `printVegetarianMenu()` mentioned earlier

Adding Iterator: Define Iterator Interface

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

Simple!

Create Iterator for DinerMenu

```
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;
    }
  }
}
```

Create Iterator for PancakeMenu

```java
import java.util.ArrayList;

public class PancakeHouseMenuIterator implements Iterator {
  ArrayList items;
  int position = 0;

  public PancakeHouseMenuIterator(ArrayList items) {
    this.items = items;
  }

  public Object next() {
    Object object = items.get(position);
    position = position + 1;
    return object;
  }

  public boolean hasNext() {
    if (position >= items.size()) {
      return false;
    } else {
      return true;
    }
  }
```

Update Menu Classes

- For each class
  - Delete `getMenuItems()` method
  - Add `createIterator()` method that returns the appropriate iterator
- Now, our implementation details are hidden and our client can switch to coding to an interface

Update Server

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

  public Server(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.print(menuItem.getName() + ", ");
      System.out.print(menuItem.getPrice() + " -- ")
      System.out.prinln(menuItem.getDescription());
    }
  }
}
```

Only one loop; no more code duplication!

Iteration behaves polymorphically, working across multiple types of collection classes

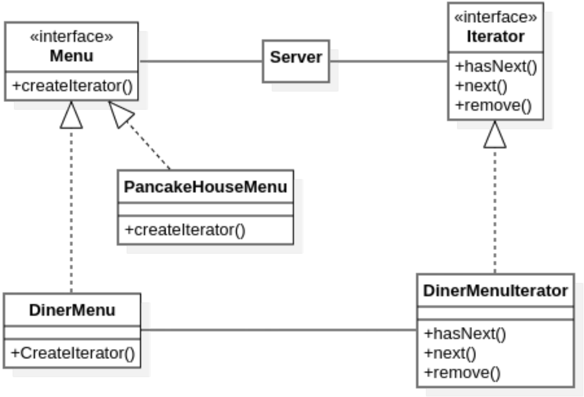
What’s Missing?

- Currently `Server` is still tied to our specific `Menu` classes
- Create `Menu` interface with a single method: `createIterator()`
- Update `Menu` classes to implement `Menu` interface
- Update `Server` to hold a collection of menu objects (accessing them only via the `Menu` interface)
  - Now `Server` depends on two interfaces `Menu` and `Iterator`
    - Concrete `Menu` and `Iterator` classes may now come and go with ease

Why Reinvent the Wheel?

- Java provides an `Iterator` interface in `java.util`
  - It has one extra method than our homegrown iterator: `remove()`
- Lets switch our code to make use of this interface
  - Delete `PancakeHouseMenuIterator` class: `ArrayList` provides its own implementation of `java.util.Iterator`
  - Update `DinerMenuIterator` to implement `remove()` method
- With these changes, we have the following structure...

Menu Example's Structure



Demonstration

Note: this design is extensible; what happens if we were to add a new menu that stores items in a hash table?

Wrapping Up

- Iterator: separate the management of a collection from the traversal of a collection
- Iterator can be used within the context of the next pattern: Composite Pattern

Are there any questions?