Поведенческий шаблон проектирования Iterator. Введение в JCF:

Внешняя система, на вход принимающая Basket, задача : абстрагироваться от способа хранения элементов и создать универсальный, а следовательно масштабируемый способ перебора элементов, на основе которого будет работать основной метод внешней системы : calculateTotalPrice(), для этого ShopAssistance должен оперировать абстракцией, а не конкретной корзиной 🡪необходим IF для перебора элементов, но стоит помнить , что шаблон проектирования это не панацея и если есть возможность реализовать логику проще, то следует придерживаться максимально простого варианта

StaticBasket(Fixed)

Basket

size() get(index)

ShopAssistance

DynamicBasket

ListBasket

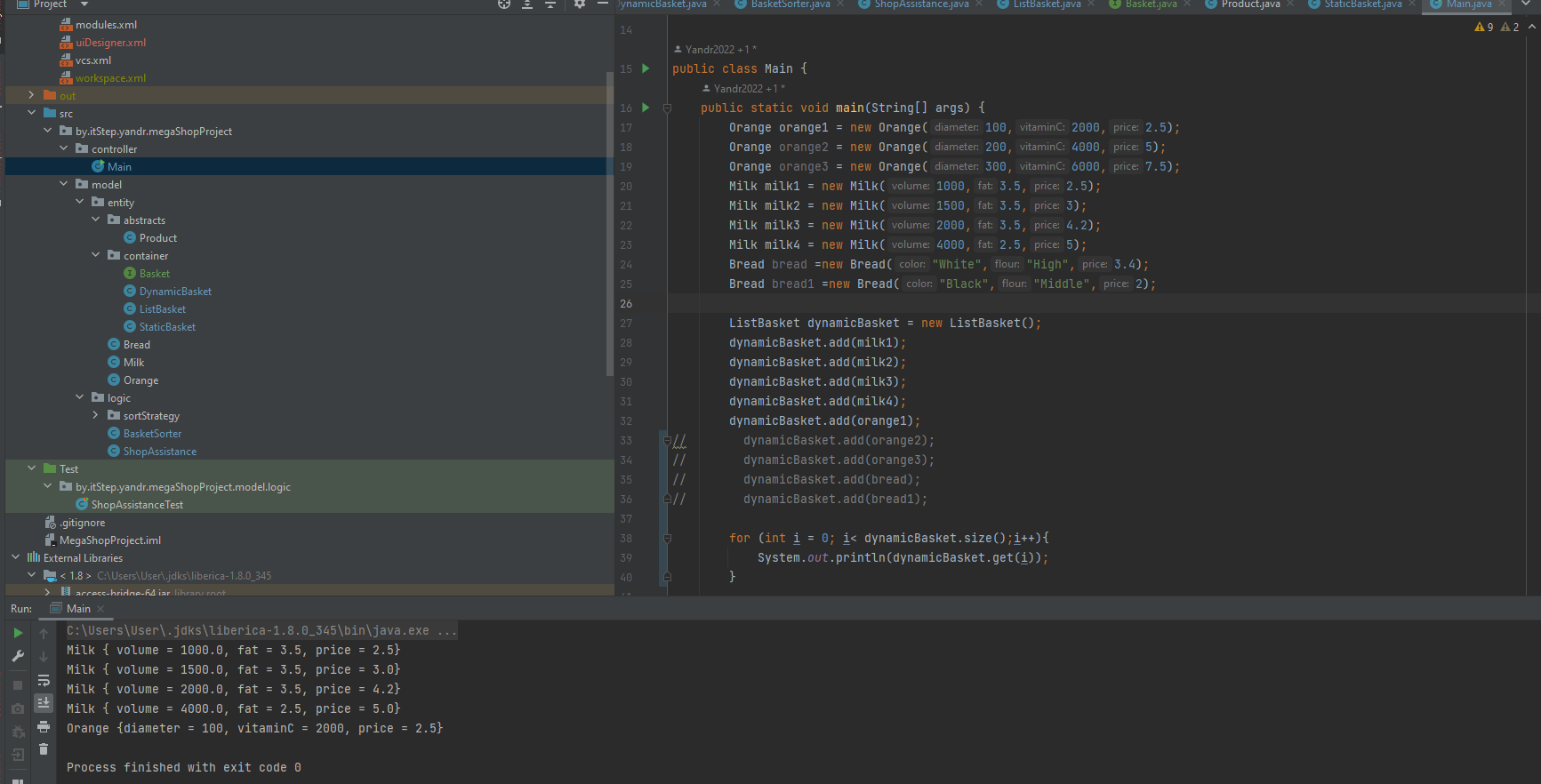
Для определения недостающего функционала, реализуем InterFace Basket

package by.itStep.yandr.megaShopProject.model.entity.container;  
  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
  
public interface Basket {  
 int size ();  
 Product get(int index);  
}

и имплементируем его всеми нашими видами корзин:

в DynamicBasket и StaticBasket данные методы уже реализованы, только переименуем, а вот ListBasket не имеет внутри контейнера, по-этому требуется отдельная реализация для get():

@Override  
public Product get(int index) {  
 if (isEmpty() || index < 0 || index >= size) {  
 return null;  
 }  
 Node temp = first;  
 for (int i = 0; i < index; i++) {  
 temp = temp.next;  
 }  
 return temp.product;  
}

Тестируем:

Отвязываем ShopAssistance от конкретных имплементтаций:

package by.itStep.yandr.megaShopProject.model.logic;  
  
import by.itStep.yandr.megaShopProject.model.entity.container.Basket;  
  
public class ShopAssistance {  
  
 private ShopAssistance() {  
 }  
  
 public static double calculateTotalPrice(Basket basket) {  
 double total = 0;  
 for (int i = 0; i < basket.size(); i++) {  
 total += basket.get(i).getPrice();  
 }  
  
 return total;  
 }  
}

Рассмотрим GOF – шаблон Итератор, представим, что мы получили контейнеры в виде готовых коллекций, в первую очередь нам нужно спрятать их за прослойку абстракции, что позволит нам добавить логику перебора контейнеров, не изменяя кода классов коллекций.Представим, что у нас есть готовый реализованный набор коллекций и требуется возможность перебора их элементов, для начала, необходимо спрятать наши контейнеры за абстракцию.

1. Механизм доступа к элементам:

С точки зрения абстракции, нужно создать возврат условного однонаправленного курсора, который на каждом вызове метода будет перемещаться на следующую позицию и возвращать элемент, который был пройден

MyIterator

next()/element()->item

hasNext()/hasNextElement()->boolean

ShopAssistance

1. Для каждого контейнера создаем конкретный класс, который будет имплементить этот интерфейс и реализовывать заявленные методы на базе контейнера, которые он будет инкапсулировать, реализовывается через классическую композицию, как класс-адаптер

DynamicBasket

StaticBasket

StaticBasketIterator

DynamicBasketIterator

ListBasket

ListBasketIterator

ListBasket

DynamicBasket

StaticBasket

Создаеь Интерфейс

ckage by.itStep.yandr.megaShopProject.model.entity.iterator;  
  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
  
public interface MyIterator {  
 Product next();  
 boolean hasNext();  
}

И вносим изменения в ShopAssistance

import by.itStep.yandr.megaShopProject.model.entity.iterator.MyIterator;  
  
public class ShopAssistance {  
 private ShopAssistance() {  
 }  
 public static double calculateTotalPrice(MyIterator iterator) {  
 double total = 0;  
 while (iterator.hasNext()) {  
 total += iterator.next().getPrice();  
 }  
 return total;  
 }  
}

Переходим к реализации конкретных Iterators, начнем со StaticBasket (FixedBasket):

- создаем ссылку на контейнер private (+ реализуем конструктор)

* Реализовываем методы:

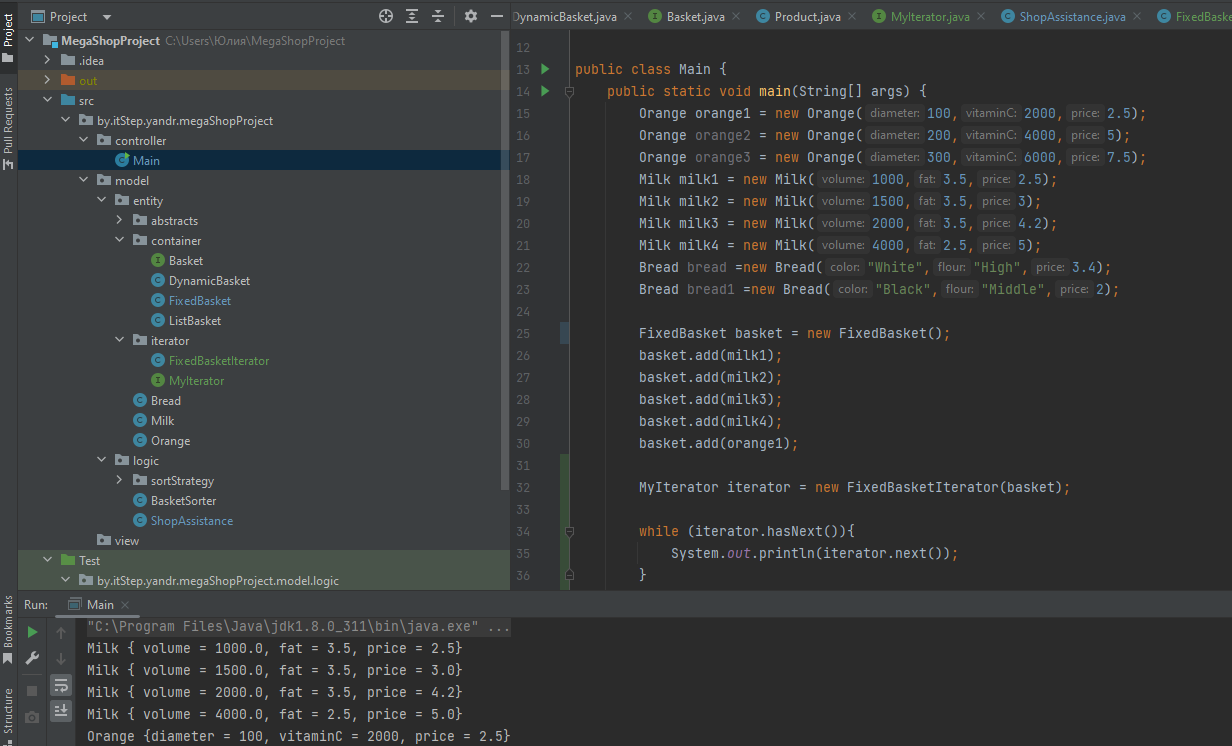
Next : тк контейнер реализован через массив, возвращаем значение по указанному индексу, значение которого хранит внутренняя переменная(private) -current(курсор), которая ссылается на текущую переменную, в конструкторе присваиваем ей значение 0

package by.itStep.yandr.megaShopProject.model.entity.iterator;  
  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
import by.itStep.yandr.megaShopProject.model.entity.container.FixedBasket;  
  
public class FixedBasketIterator implements MyIterator {  
 private FixedBasket basket;  
  
 private int current;  
  
 public FixedBasketIterator(FixedBasket basket) {  
 this.basket = basket;  
 current = 0;  
 }  
  
 @Override  
 public Product next() {  
 return basket.get(current++);  
 }

При реализации hasNext просто возвращаем сравнение текущего элемента с размером контейнера:

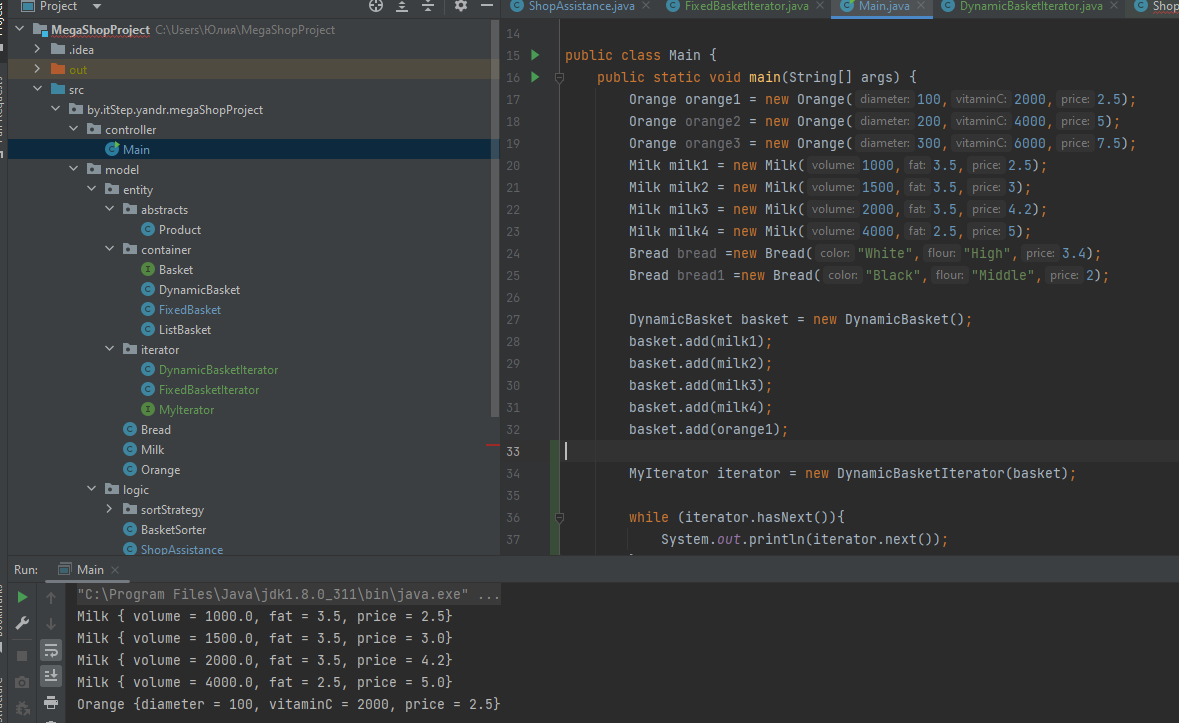
@Override  
public boolean hasNext() {  
 return current< basket.size();  
}

Тестируем:

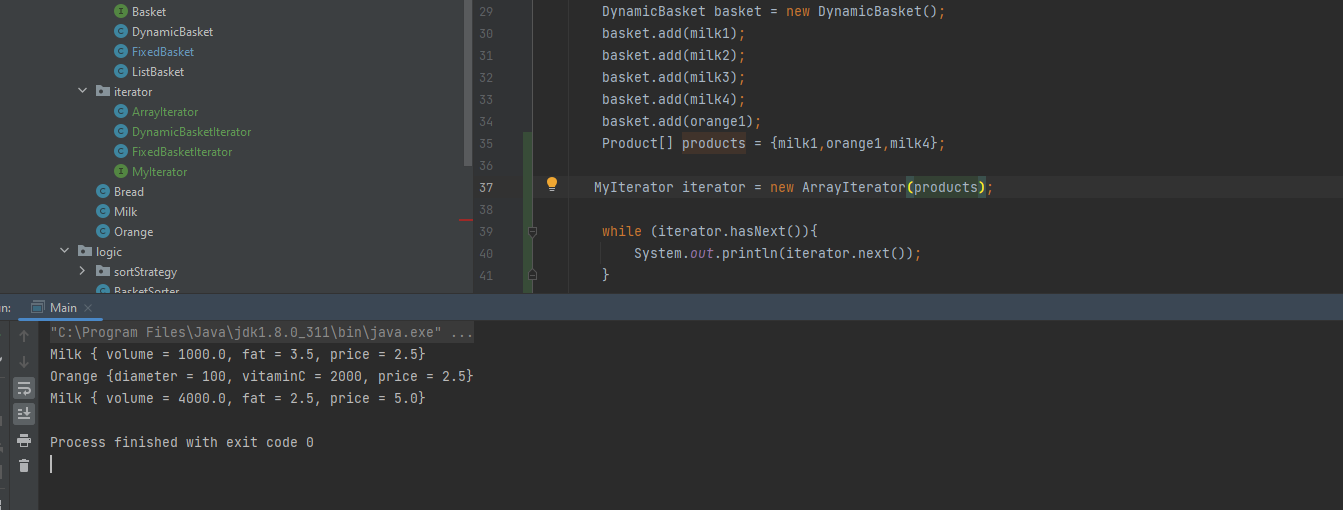


Повторяем реализацию и тест для DynamicBasket

package by.itStep.yandr.megaShopProject.model.entity.iterator;  
  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
import by.itStep.yandr.megaShopProject.model.entity.container.DynamicBasket;  
  
public class DynamicBasketIterator implements MyIterator {  
 private DynamicBasket basket;  
  
 private int current;  
  
 public DynamicBasketIterator(DynamicBasket basket) {  
 this.basket = basket;  
 current = 0;  
 }  
  
 @Override  
 public Product next() {  
 return basket.get(current++);  
 }  
  
 @Override  
 public boolean hasNext() {  
 return current< basket.size();  
 }  
}



Дополнительно для примера создаем итератор для перебора массива

package by.itStep.yandr.megaShopProject.model.entity.iterator;  
  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
  
public class ArrayIterator implements MyIterator {  
 private Product[] products;  
 private int current;  
  
 public ArrayIterator(Product[] products) {  
 this.products = products;  
 current = 0;  
 }  
  
 @Override  
 public Product next() {  
 return products[current++];  
 }  
  
 @Override  
 public boolean hasNext() {  
 return current < products.length;  
 }

Данный тип итератора будет одноразовым, тк при следующем вызове после перебора всех элементов, курсор будет на последнем элементе и hasNext будет возвращать false.

Для создания много разового итератора необходимо добавить метод обновляющий значение курсора

package by.itStep.yandr.megaShopProject.model.entity.iterator;  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
  
public interface MyIterator {  
 Product next();  
 boolean hasNext();  
 void reset();  
}

-|>

@Override  
public void reset() {  
 current = 0;  
}

Теперь помещаем соответствующий тип Итератора в сам контейнер,

Iterator

getIterator()

ShopAssistance

MyIterator

next()/element()->item

hasNext()/hasNextElement()->boolean

ListBasketIterator



DynamicBasketIterator



StaticBasketIterator

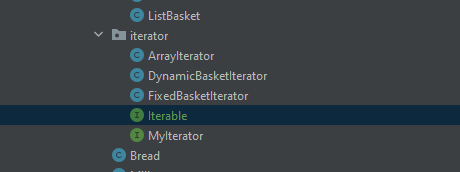


StaticBasket

ListBasket

DynamicBasket

Создаем еще один интерфейс

package by.itStep.yandr.megaShopProject.model.entity.iterator;  
  
public interface Iterable {  
 MyIterator getIterator();  
}

И имплементим его нашими контейнерами, после чего в унаследованном методе передаем нужный тип итератора, которому в свою очередь передаем неявную ссылку на сам объект

public class DynamicBasket implements Basket, Iterable

@Override  
public MyIterator getIterator() {  
 return new DynamicBasketIterator(this);  
}

Так как мы еще не реализовывали итератор для ListBasket, заполняем этот пробел

package by.itStep.yandr.megaShopProject.model.entity.iterator;  
  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
import by.itStep.yandr.megaShopProject.model.entity.container.ListBasket;  
  
public class ListBasketIterator implements MyIterator {  
 private ListBasket basket;  
 private int current;  
  
 public ListBasketIterator(ListBasket basket) {  
 this.basket = basket;  
 current = 0;  
 }  
  
 @Override  
 public Product next() {  
 return basket.get(current++);  
 }  
  
 @Override  
 public boolean hasNext() {  
 return current < basket.size();  
 }  
  
 @Override  
 public void reset() {  
 current = 0;  
 }  
}

по аналогии реализуем метод унаследованный новой имплементацией

@Override  
public MyIterator getIterator() {  
 return new ListBasketIterator(this);  
}-|>

@Override  
public MyIterator getIterator() {  
 return new FixedBasketIterator(this);  
}

Переходим ShopAssistance и в calculateTotalPrice передаем в качестве параметра уже объект имплементирующий класс Iterable, а во внутренней реализации метода создаем переменную типа MyIterator, значением которой будет полученный из аргумента объект.

package by.itStep.yandr.megaShopProject.model.logic;  
import by.itStep.yandr.megaShopProject.model.entity.iterator.Iterable;  
import by.itStep.yandr.megaShopProject.model.entity.iterator.MyIterator;  
  
public class ShopAssistance {  
  
 private ShopAssistance() {  
 }  
  
 public static double calculateTotalPrice(Iterable iterable) {  
 MyIterator iterator = iterable.getIterator();  
 double total = 0;  
 while (iterator.hasNext()) {  
 total += iterator.next().getPrice();  
 }  
 return total;  
 }  
}

Таким образом мы сформировали полноценную современную реализацию паттерна Итератор, состоящую из 2-х интерфейсов:

MyIterator , как способ формирования классов-адаптеров, для реализации унифицированного способа перебора элементов

Iterable, позволяющий возвращать подходящий данному контейнеру итератор.

Рассмотрение Java Collection FrameWork(since JDK5.0)

В Java есть 3 платформы, определяющая структуру программной системы:

1. Библиотека(ответственность за создание и использование объекта на коиенте)
2. Фреймворк(ответственность разделена между клиентом и платформой и как правило состоит из нескольких библиотек, пример Junit)
3. Контейнер

Коллекции это библиотека, которая целостно сформировалась с 5й версии JDK ,тк с этой же версии были введены дженерики, то и коллекции все дженерализированы. Обычно, с точки зрения терминологии, коллекциями называют те контейнеры, которые являются наследниками интерфейса Collection, но помимо коллекций там присутствуют и другие контейнеры, по-этому лучше их называть контейнером.

Вся библиотека коллекций делится на 3 части:

1. Интерфейсная часть 70% (Interfaces)
2. Реализации 20% (Implementations)
3. Логика 10% () Algorithms:

* Objects
* Arrays -s от services, тк классы-функциональные
* Collections

Интерфейсная часть 70% (Interfaces):

главный иф коллекции, предоставляющий способ перебора элементов

<<Iterator>>

next()

hasNext()

Iterable<E> (<<Iterable>>)

iterator()

этот IF позволил реализовать forEach с 5.0 JDK

Level 1

IF<<Collections>>

size()

isEmpty()

clear()

add()

remove()

…

задает общее поведение для всех коллекций

IF<<ListIterator>>

Level 2

Cпециализации поведения Collections

IF<<SortedSet>>

Упорядоченные уникальные объекты

<<Deque>>

LIFO – last input-first output

IF<<Queue>>

Используется в многопоточности, алгоритм обработки данных(FIFO – first input-first output)

IF<<Set>> уникальность объектов

IF<<List>> инд.доступ

get(index)

set(index, E)

… 90% использов-я

Level 3