Universitat de Lleida

Actividad del Patrón Visitante

Ampliación de BBDD e Ingeniería del Software
Grado en Ingeniería Informática

Luis Carrasquer Sampietro
Dídac Cayuela Dolcet
Aleix Drudis Mola

25 de mayo de 2024

Índex

Introducció

Aquest informe descrivim les decisions de disseny preses per implementar el patró visitant en una jerarquia de productes que inclouen Items i Packs. El problema a resoldre consisteix en afegir operacions als productes sense modificar les seves classes, facilitant així l'extensibilitat i mantenibilitat del codi. Els objectius específics inclouen l'actualització dels preus dels Items que estan per sota d'un preu mínim i el còmput del nombre d'Items que es troben en aquesta situació, tot implementant aquestes funcionalitats mitjançant el patró visitant.

Implementació del Patró Visitant per Afegir Noves Operacions (Apartat a)

El primer pas per implementar el patró visitant ha consistit en definir una interfície `ProductVisitor`. Aquesta interfície declara mètodes de visita per cada tipus de producte (`Item` i `Pack`). Aquest enfocament permet que qualsevol visitant pugui operar tant en `Items` com en `Packs` de manera transparent. La interfície `ProductVisitor` es defineix a continuació:

```
public interface ProductVisitor {
    void visit(Item item);
    void visit(Pack pack);
}
```

També hem fet unes modificacions a la interfície `Product`, hem afegit el mètode `Accept` que implementarem a continuació.

```
import java.math.BigDecimal;
import java.util.HashSet;
import java.util.Set;

public interface Product {
    public BigDecimal getPrice();
    public void checkMinItemPrice(BigDecimal minItemPrice);
    long countItemsBelowMinPrice(BigDecimal minPrice, Set<Product>
visitedProducts);

    default long countItemsBelowMinPrice(BigDecimal minPrice) {
        Set<Product> visited = new HashSet<>();
        return countItemsBelowMinPrice(minPrice, visited);
    }

    void accept(ProductVisitor visitor);
}
```

Les classes `Item` i `Pack` implementen el mètode `accept` per acceptar un visitant i permetre que aquest executi l'operació corresponent. Afegint el següent mètode a cada classe, assegurem que aquestes poden rebre visitants:

Item.java

```
@Override
public void accept(ProductVisitor visitor) {
    visitor.visit(this);
}
```

Pack.java

```java
@Override
public void accept(ProductVisitor visitor) {
    visitor.visit(this);
}
```

Aquest enfocament permet afegir noves operacions simplement implementant nous visitants sense haver de modificar les classes `Item` i `Pack`, mantenint així el codi obert per a l'extensió però tancat per a la modificació, seguint el principi d'OCP (Open/Closed Principle).

Visitant per Modificar Preus per sota d'un MinPrice (Apartat b)

Per implementar un visitant que actualitzi els preus dels `Items` per sota d'un preu mínim (`minPrice`), hem dissenyat la classe `MinPriceUpdaterVisitor`. Aquest visitant encapsula la lògica d'actualització de preus i garanteix que aquesta es realitzi de manera coherent en tota la jerarquia de productes.

El `MinPriceUpdaterVisitor` conté una variable `minPrice` que s'inicialitza al seu constructor. Aquest visitant actualitza els preus dels `Items` si estan per sota del `minPrice`. El codi del visitant és el següent:

```
import java.math.BigDecimal;
import java.util.HashSet;
import java.util.Set;

public class CountItemsBelowMinPriceVisitor implements ProductVisitor
{
    private BigDecimal minPrice;
    private int count;
    private Set<Product> visited;

    private CountItemsBelowMinPriceVisitor(BigDecimal minPrice) {
        if (minPrice.compareTo(BigDecimal.ZERO) <= 0) {
            throw new IllegalArgumentException("Minimum price must be
greater than zero");
        }
        this.minPrice = minPrice;
        this.count = 0;
        this.visited = new HashSet<>();
    }

    @Override
    public void visit(Item item) {
        if (!visited.contains(item)) {
            visited.add(item);
            if (item.getPrice().compareTo(minPrice) < 0) {
                count++;
            }
        }
    }

    @Override
    public void visit(Pack pack) {
        if (!visited.contains(pack)) {
            visited.add(pack);
            for (Product product : pack.getProducts()) {
                product.accept(this);
            }
        }
    }

    public int getCount() {
        return count;
    }
```

```
    public static int countItemsBelowMinPrice(Product product,
BigDecimal minPrice) {
        CountItemsBelowMinPriceVisitor visitor = new
CountItemsBelowMinPriceVisitor(minPrice);
        product.accept(visitor);
        return visitor.getCount();
    }
}
```

Es defineix un mètode estàtic per iniciar el procés de visita, validant prèviament el valor de `minPrice` per assegurar que és positiu. Si el `minPrice` no és positiu, es llença una excepció `IllegalArgumentException`. Aquest mètode estàtic s'utilitza per iniciar la visita sobre un producte donat:

```
public static void updatePrices(Product product, BigDecimal minPrice)
{
    MinPriceUpdaterVisitor visitor = new
MinPriceUpdaterVisitor(minPrice);
    product.accept(visitor);
}
```

Aquest enfocament permet garantir que tots els `Items` dins d'un `Pack` (i de qualsevol estructura recursiva de `Packs`) s'actualitzin correctament si estan per sota del `minPrice`.

Ens agradaria deixar clar que el mètode com a tal no fa la comprovació que llança l'excepció ja que hem preferit implementar aquesta en el constructor per tal de mantenir una estructura mes coherent.

Visitant per Comptar Items per sota d'un MinPrice (Apartat c)

Per comptar el nombre d'`Items` que estan per sota d'un preu mínim, es va dissenyar el `CountItemsBelowMinPriceVisitor`. Aquest visitant encapsula la lògica de còmput i permet evitar cicles i comptar duplicats en estructures de productes potencialment complexes.

El `CountItemsBelowMinPriceVisitor` manté un conjunt (`Set`) de productes visitats per prevenir cicles i duplicats. La variable `count` manté el recompte d'`Items` que estan per sota del `minPrice`. El codi del visitant és el següent:

```
import java.math.BigDecimal;
import java.util.HashSet;
import java.util.Set;

public class CountItemsBelowMinPriceVisitor implements ProductVisitor
{
    private BigDecimal minPrice;
    private int count;
    private Set<Product> visited;

    private CountItemsBelowMinPriceVisitor(BigDecimal minPrice) {
        if (minPrice.compareTo(BigDecimal.ZERO) <= 0) {
            throw new IllegalArgumentException("Minimum price must be
greater than zero");
        }
        this.minPrice = minPrice;
        this.count = 0;
        this.visited = new HashSet<>();
    }

    @Override
    public void visit(Item item) {
        if (!visited.contains(item)) {
            visited.add(item);
            if (item.getPrice().compareTo(minPrice) < 0) {
                count++;
            }
        }
    }

    @Override
    public void visit(Pack pack) {
        if (!visited.contains(pack)) {
            visited.add(pack);
            for (Product product : pack.getProducts()) {
                product.accept(this);
            }
        }
    }
```

```
    public int getCount() {
        return count;
    }

    public static int countItemsBelowMinPrice(Product product,
BigDecimal minPrice) {
        CountItemsBelowMinPriceVisitor visitor = new
CountItemsBelowMinPriceVisitor(minPrice);
        product.accept(visitor);
        return visitor.getCount();
    }
}
```

Es defineix un mètode estàtic per iniciar el procés de visita i obtenir el resultat, assegurant-se que cada producte només es compta una vegada, fins i tot si es troba en cicles dins de l'estructura de `Packs`:

```
public static int countItemsBelowMinPrice(Product product, BigDecimal
minPrice) {
    CountItemsBelowMinPriceVisitor visitor = new
CountItemsBelowMinPriceVisitor(minPrice);
    product.accept(visitor);
    return visitor.getCount();
}
```

Aquest enfocament assegura que el recompte d'`Items` és precís i no es veurà afectat per estructures no arborescents o cicles dins dels `Packs`.

Observacions

Hem llegit amb deteniment les observacions proporcionades sobre l'exercici anterior, així doncs hem tingut en compte els errors de disseny que vam cometre i els hem intentat resoldre per tal d'intentar millorar i tenir una bona base a l'hora de fer aquest segon exercici de disseny. Ens sap greu no haver assolit de manera satisfactòria les expectatives en l'anterior entrega. A continuació mostrem els canvis implementats en les classes Item `Item.java` i `Product.java` per tal de corregir aquests errors.

Item.java

```
public class Item implements Product{

    private BigDecimal price;

    public Item(BigDecimal price) {
        checkGreaterThanZero(price);

        this.price = price.setScale(2, RoundingMode.HALF_UP);
    }

    private void checkGreaterThanZero(BigDecimal price) {
        if(price.compareTo(BigDecimal.ZERO) <= 0){
            throw new IllegalArgumentException("Price must be greater
than zero");
        }
    }

    @Override
    public BigDecimal getPrice() {
        return this.price;
    }

    public void setPrice(BigDecimal newPrice) {
        newPrice = newPrice.setScale(2, RoundingMode.HALF_UP);
        checkGreaterThanZero(newPrice);
        this.price = newPrice;
    }

    @Override
    public void checkMinItemPrice(BigDecimal minItemPrice){
        checkGreaterThanZero(minItemPrice);

        if(this.price.compareTo(minItemPrice) < 0){
            this.price = minItemPrice.setScale(2,
RoundingMode.HALF_UP);
        }
    }

    @Override
    public long countItemsBelowMinPrice(BigDecimal minPrice,
Set<Product> visitedProducts) {
        return this.price.compareTo(minPrice) < 0 ? 1 : 0;
    }
```

```
    @Override
    public void accept(ProductVisitor visitor) {
        visitor.visit(this);
    }
}
```

Pack.java

```
public class Pack implements Product{

    private List<Product> products;

    public Pack() {
        this.products = new ArrayList<>();
    }

    public void addProduct(Product product) {
        if (product == null) {
            throw new NullPointerException("Product is null");
        }

        products.add(product);
    }

    @Override
    public BigDecimal getPrice() {
        return products.stream()
                .map(Product::getPrice)
                .reduce(BigDecimal.ZERO, BigDecimal::add)
                .setScale(2, RoundingMode.HALF_UP);

    }

    public List<Product> getProducts() {
        return Collections.unmodifiableList(products);
    }

    @Override
    public void checkMinItemPrice(BigDecimal minItemPrice) {
        products.forEach(product ->
product.checkMinItemPrice(minItemPrice));
    }

    @Override
    public long countItemsBelowMinPrice(BigDecimal minPrice,
Set<Product> visited){
        visited.add(this);

        long count = 0;

        for (Product product : products){
            if (!visited.contains(product)){
                count += product.countItemsBelowMinPrice(minPrice,
visited);
            }
        }
        return count;
    }
```

```
    @Override
    public void accept(ProductVisitor visitor) {
        visitor.visit(this);
    }

}
```

Conclusió

La implementació del patró visitant permet afegir noves operacions als productes de manera elegant i escalable. Aquest enfocament millora la mantenibilitat del codi i facilita l'extensió de funcionalitats sense modificar les classes existents. La modularitat del codi es veu millorada, ja que les operacions es poden afegir mitjançant nous visitants en lloc de modificar les classes dels productes directament.

Solució Dídac

Exercici Patró Composite

Dídac Cayuela Dolcet

Ⓘ Product.java

```
import java.math.BigDecimal;

public interface Product {

    BigDecimal getPrice();
    Void checkMinItemPrice(BigDecimal minItemPrice);
    long countItemsBelowMinPrice(BigDecimal minPrice, set<Product> visitedProducts);

    default long countItemsBelowMinPrice(BigDecimal minPrice) {
        set<Product> visited = new HashSet<>();
        return countItemsBelowMinPrice(minPrice, visited);
    }

    Void accept(ProductVisitor visitor);
```

Ⓒ Item.java

```
Import java.math.BigDecimal;
Import java.math.RoundingMode;

public class Item implements Product {
    private BigDecimal price;

    public Item(BigDecimal price) {

        checkGreaterThanZero( price );

        this.price = price.setScale(2, RoundingMode.HALF_UP);
    }

    @Override
    public BigDecimal getPrice(){
        return this.price;
    }

    public void setPrice(BigDecimal newPrice) {
        newPrice = newPrice.setScale(2, RoundingMode.HALF_UP);
        checkGreaterThanZero( newPrice );
        this.price = newPrice;
```

```
}

@Override
public void checkMinItemPrice(BigDecimal minItemPrice) {
    checkGreaterThanZero(minItemPrice);
    if(this.price.compareTo(minItemPrice) < 0) {
        this.price = minItemPrice.setScale(2, RoundingMode.HALF_UP);
    }
}

private void checkGreaterThanZero(BigDecimal minItemPrice) {
    if(price.compareTo(BigDecimal.ZERO) <= 0) {
        throw new IllegalArgumentException("Price must be greater than zero");
    }
}

@Override
public long countItemsBelowMinPrice(BigDecimal minPrice, Set<Product> visitedProducts) {
    return this.price.compareTo(minPrice) < 0 ? 1 : 0;
}

@Override
public void accept(ProductVisitor visitor) {
    visitor.visit(this);
}
}
```

Pack.java

```
public class Pack implements Product {
    private List<Product> products;

    public Pack() {
        this.products = new ArrayList<>();
    }

    public void getPrice() {
        BigDecimal totalPrice = BigDecimal.ZERO;
        for(Product p : products) {
            totalPrice = totalPrice.add(p.getPrice());
```

```
    }
    return totalPrice.setScale(2, RoundingMode.HALF_UP);
}

public void AddProduct(Product product) {
    if (product == null) {
        throw new NullPointerException("Product is Null");
    }
    products.add(product);
}

public List<Product> getProducts() {
    return Collections.unmodifiableList(products);
}

public void checkMinItemPrice(BigDecimal minItemPrice) {
    for (Product p : products) {
        p.checkMinItemPrice(minItemPrice);
    }
}
```

@Override

```
public long countItemsBelowMinPrice(BigDecimal minPrice, Set<Product> visited) {
    visited.add(this);
    long count = 0;
    for (Product product : products) {
        if (!visited.contains(product)) {
            count += product.countItemsBelowMinPrice(minPrice, visited);
        }
    }
    return count;
}
```

@Override

```
public void accept(ProductVisitor visitor) {
    visitor.visit(this);
    for (Product product : products) }
```

```
        for (Product product : products) {
            product.accept(visitor);
        }
    }
```

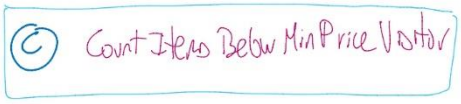
(I) ProductVisitor

```
public interface ProductVisitor {
    void visit(Item item);
    void visit(Pack pack);
}
```

(C) MinPriceUpdaterVisitor

```
public class MinPriceUpdateVisitor implements ProductVisitor {

    private BigDecimal minPrice;

    private MinPriceUpdateVisitor(BigDecimal minPrice) {
        if(minPrice.compareTo(BigDecimal.ZERO) <= 0) {
            throw new IllegalArgumentException("Minimum Price must be grecde than zero");
        }
        this.minPrice = minPrice.setScale(2, RoundingMode.HALF_UP);
    }

    @Override
    public void visit(Item item) {
```

```
public void visit(Item item) {
    if (item.getPrice().compareTo(minPrice) < 0) {
        item.setPrice(minPrice);
    }
}

@Override
public void visit(Pack pack) {
    for (Product product : pack.getProducts()) {
        product.accept(this);
    }
}

public static void updatePrices(Product product, BigDecimal minPrice) {
    MinPriceUpdateVisitor visitor = new MinPriceUpdateVisitor(minPrice);
    product.accept(visitor);
}
```




```
public class CountItemsBelowMinPriceVisitor implements ProductVisitor {
    private BigDecimal minPrice;
    private int count;
    private Set<Product> visited;

    private CountItemsBelowMinPriceVisitor(BigDecimal minPrice) {
        if (minPrice.compareTo(BigDecimal.ZERO) <= 0) {
            throw new IllegalArgumentException("Minimum price must be greater than Zero");
        }
        this.minPrice = minPrice;
        this.count = 0;
```

```
        this.minPrice = minPrice;
        this.count = 0;
        this.visited = new HashSet<>();
    }

    @Override
    public void visit(Item item) {
        if (!visited.contains(item)) {
            visited.add(item);
            if (item.getPrice().compareTo(minPrice) < 0) {
                count++;
            }
        }
    }

    @Override
    public void visit(Pack pack) {
        if (!visited.contains(pack)) {
            visited.add(pack);
            for (Product product : pack.getProducts()) {
                product.accept(this);
            }
        }
    }

    public int getCount() {
        return count;
    }

    public static int countItemsBelowMinPrice(Product product, BigDecimal minPrice) {
        CountItemsBelowMinPriceVisitor visitor = new CountItemsBelowMinPriceVisitor(minPrice);
        product.accept(visitor);
        return visitor.getCount();
    }
}
```

Solució Luis

Práctica 4

sábado, 25 de mayo de 2024 12:10

```
public interface Product {
    public BigDecimal getPrice();
    public void checkMinItemPrice(BigDecimal minItemPrice);
    long countItemsBelowMinPrice(BigDecimal minPrice, Set<Products> visitedProducts);
    default long countItemsBelowMinPrice(BigDecimal minPrice) {
        Set<Product> visited = new HashSet<>();
        return countItemsBelowMinPrice(minPrice, visited);
    }
    void accept(ProductVisitor visitor);
}
public interface ProductVisitor {
    void visit(Item item);
    void visit(Item item);
}

public class Pack implements Product {
    private List<Products> products;

    public Pack(){
      this.products= new ArrayList<>();
}
public void addProduct(Product product){
    if (product==null){
        throw new NullPointerException("Product is null");
    }
    products.add();
}

@Override
public BigDecimal getPrice() {
    return products.stream()
            .map(Product::getPrice)
            .reduce(BigDecimal.ZERO, BigDecimal::add)
            .setScale(2, RoundingMode.HALF_UP);
}
public List<Product> getProducts() {
    return Collection.unmodifiableList(products);
}
@Override
public void checkMinItemPrice(BigDecimal minItemPrice) {
    products.forEach(product ->product.checkMinItemPrice);
}

@Override
public long countItemsBelowMinPrice(BigDecimal minPrice,Set<Products> visited){
public long
```

```
        visitedItems.add(this);
        long count = 0;
        for (Product product : products) {
            if (!visitedItems.contains(product)) {
                count += product.countItemsBelowMinPrice(minPrice, visitedItems);
            }
        }
        return count;
    }

    @Override
    public void accept(ProductVisitor visitor) {
        visitor.visit(this);
        for (Product product : products) {
            product.accept(visitor);
        }
    }
}

public class Item implements Product {
    private BigDecimal price;

    public Item(BigDecimal price) {
        checkGreaterThanZero(price);
        this.price = price.setScale(2, RoundingMode.HALF_UP);
    }

    private void checkGreaterThanZero(BigDecimal price) {
        if (price.compareTo(BigDecimal.ZERO) <= 0) {
            throw new IllegalArgumentException("Price must be greater than zero");
        }
    }

    @Override
    public BigDecimal getPrice() {
        return this.price;
    }

    public void setPrice(BigDecimal newPrice) {
        newPrice = newPrice.setScale(2, RoundingMode.HALF_UP);
        checkGreaterThanZero(newPrice);
        this.price = newPrice;
    }

    @Override
    public void checkMinItemPrice(BigDecimal minItemPrice) {
        checkGreaterThanZero(minItemPrice);
        if (this.price.compareTo(minItemPrice) < 0) {
            this.price = minItemPrice.setScale(2, RoundingMode.HALF_UP);
```

```
        }
    }

    @Override
    public long countItemsBelowMinPrice(BigDecimal minPrice, Set<Product> visitedProducts){
        return this.price.compareTo(minPrice)<0?1:0;
    }
    public void accept(ProductVisitor visitor){
        visitor.visit(this);
    }
}
public class MinPriceUpdaterVisitor implements ProductVisitor{
    private BigDecimal minPrice;
    private MinPriceUpdaterVisitor(BigDecimal minPrice){
        if(minPrice.compareTo(BigDecimal.ZERO)<=0){
            throw new IllegalArgumentException("Minimum price must be greater than zero");
        }
        this.minPrice = minPrice.setScale(2,RoundingMode.HALF_UP);
    }
    @Override
    public void visit(Item item){
        if(item.getPrice().compareTo(minPrice)<0){
            item.setPrice(minPrice);
        }
    }

    @Override
    public void visit(Pack pack){
        for(Product product:pack.getProducts()){
            product.accept(this);
        }
    }
    public static void updatePrices(Product product, BigDecimal minPrice){
        MinPriceUpdaterVisitor visitor = new MinPriceUpdaterVisitor(minPrice);
        product.accept(visitor);
    }
}

public class CountItemsBelowMinPriceVisitor implements ProductVisitor{
    private BigDecimal minPrice;
    private int count;
    private Set<Products visited;

    private CountItemsBelowMinPriceVisitor(BigDecimal minPrice){
        if(minPrice.compareTo(BigDecimal.ZERO)<=0){
            throw new IllegalArgumentException("Minimum price must be greater than zero")
        }
        this.minPrice=
```

```
        this.count = 0;
        this.visited = new HashSet<>();
    }

    @Override
    public void visit(Item item) {
        if (!visited.contains(item)) {
            visited.add(item);
            if (item.getPrice().compareTo(minPrice) < 0) {
                count++;
            }
        }
    }

    @Override
    public void visit(Pack pack) {
        if (!visited.contains(pack)) {
            visited.add(pack);
            for (Product product : pack.getProducts()) {
                product.accept(this);
            }
        }
    }
    public int getCount() {
        return count;
    }
    public static int countItemsBelowMinPrice(Product product, BigDecimal minPrice) {
        CountItemsBelowMinPriceVisitor visitor = new CountItemsBelowMinPriceVisitor(minPrice);
        product.accept(visitor);
        return visitor.getCount();
    }
}
```

Solució Aleix

```
ALEIX DRUDIS MOLA
47696715-M

interface Visitor {
    void visit (Item item);
    void visit (Pack pack);
}

interface Product {
    void accept (Visitor visitor);
    BigDecimal getPrice ();
}

class Item implements Product {
    private BigDecimal price;
    public Item (BigDecimal price ) {
        this.price = price;
    }
    @Override
    public void accept (Visitor visitor) {visitor.visit (this);}
    @Override
    public BigDecimal getPrice () { return price;}
}
class Pack implements Product {
    private List <Product> products = new ArrayList <>();
    public void addProduct (Product product) {products.add (product);}
    @Override
    public void accept ( Visitor visitor) {visitor.visit (this); }
    public List <Product> getProducts () {return products;}
    @Override
    public BigDecimal getPrice () {
        return products.stream()
                .map (Product::getPrice)
                .reduce (BigDecimal.ZERO, BigDecimal::add );
    }
}
```

```
class PriceAdjustVisitor implements Visitor {
    private BigDecimal minPrice;
    public PriceAdjusterVisitor(BigDecimal minPrice) { this.minPrice = minPrice; }
    @Override
    public void visit(Item item) {
        if (item.getPrice().compareTo(minPrice) < 0) {
            item.setPrice(minPrice);
        }
    }
    @Override
    public void visit(Pack pack) {
        for (Product product : pack.getProducts()) {
            product.accept(this);
        }
    }
}
class ItemCounterVisitor implements Visitor {
    private BigDecimal minPrice;
    private int count = 0;
    public ItemCountVisitor(BigDecimal minPrice) { this.minPrice = minPrice; }
    @Override
    public void visit(Item item) {
        if (item.getPrice().compareTo(minPrice) < 0) { count++; }
    }
    @Override
    public void visit(Pack pack) {
        for (Product product : pack.getProducts()) {
            product.accept(this);
        }
    }
    public int getCount() { return count; }
}
```