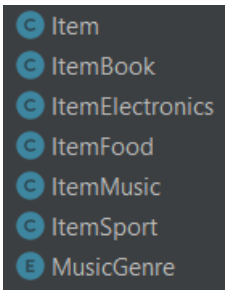
Projektowanie Obiektowe

Albert Gierlach

Zadanie 2.1 Dodatkowe właściwości dla produktów w poszczególnych kategoriach

Na początku stworzono klasy dziedziczące po klasie *Item* (które rozszerzają ją o dodatkowe właściwości) oraz do klasy nadrzędnej dodano metodę, która pobiera dodatkowe właściwości obiektu. Dodatkowa metoda będzie przeciążana przez obiekty dziedziczące z *Item*.



Enum *MusicGenre* określa gatunek muzyki (z enum korzysta klasa *ItemMusic*)

Przykładowa implementacja rozszerzonej klasy:

```java
public class ItemMusic extends Item {
    private boolean video;
    private MusicGenre genre;

    public ItemMusic(String name, Category category, int price, int quantity, boolean video, MusicGenre genre){
        super(name, category, price, quantity);
        this.video = video;
        this.genre = genre;
    }

    public ItemMusic(){
        super();
    }

    @Override
    public Map<String, Object>  getAdditionalProperties(){
        Map<String, Object> propMap = new LinkedHashMap<>();
        propMap.put("Wideo", video);
        propMap.put("Gatunek", genre.getDisplayName());
        return propMap;
    }
}
```

Zmodyfikowano klasę *PropertiesHelper* tak, by korzystała ona z metody *getAdditionalProperties()*

```
public class PropertiesHelper {

    public static Map<String, Object> getPropertiesMap(Item item) {
        Map<String, Object> propertiesMap = new LinkedHashMap<>();

        propertiesMap.put("Nazwa", item.getName());
        propertiesMap.put("Cena", item.getPrice());
        propertiesMap.put("Kategoria", item.getCategory().getDisplayName());
        propertiesMap.put("Ilość", Integer.toString(item.getQuantity()));
        propertiesMap.put("Tanie bo polskie", item.isPolish());
        propertiesMap.put("Używany", item.isSecondhand());
        propertiesMap.putAll(item.getAdditionalProperties());

        return propertiesMap;
    }
}
```

Teraz trzeba zmodyfikować wczytywanie danych z plików .csv tak, by tworzone były obiekty odpowiedniego typu. Wykorzystano do tego wzorzec *Factory.*

Ponieważ nasze klasy dziedziczące po *Item* przyjmują różną liczbę argumentów w konstruktorze, będziemy potrzebowali przekazywać kontener, w którym będą argumenty. Aby to zrealizować rozszerzono funkcjonalność *CSVReadera* o dodatkową metodę, która rozpakowuje nam wszystkie dane z danej linii pliku. Metoda zwraca kontener danych typu klucz-wartość.

```
    public Map<String, String> unpackValues(String[] line) {
        Map<String, String> unpackedValues = new HashMap<>(header.size());
        for(String key : header.keySet()){
            unpackedValues.put(key, line[header.get(key)]);
        }
        return unpackedValues;
    }
```

Metoda *readItems()* z klasy *ShopProvider* została przystosowana do współpracy z metodą *unpackValues()*

```
private static List<Item> readItems(CSVReader reader, Category category) {
    List<Item> items = new ArrayList<>();

    try {
        reader.parse();
        List<String[]> data = reader.getData();
        Map<String, String> unpackedValues;
        for (String[] dataLine : data) {
            unpackedValues = reader.unpackValues(dataLine);

            Item item = ItemFactory.getItem(category, unpackedValues);
            item.setPolish(ItemFactory.toBoolean("Tanie bo polskie"));
            item.setSecondhand(ItemFactory.toBoolean("Używany"));

            items.add(item);
            unpackedValues.clear();
        }

    } catch (IOException e) {
        e.printStackTrace();
    }

    return items;
}
```

Teraz mamy już wszystko co potrzebne do stworzenia naszej klasy *ItemFactory.* Po implementacji nasza metoda do tworzenia odpowiednich obiektów wygląda tak:

```java
public class ItemFactory {
    public static Item getItem(Category category, Map<String, String> params){
        if(category == null){
            return null;
        }

        String name = params.get("Nazwa");
        int price = toInt(params.get("Cena"));
        int quantity = toInt(params.get("Ilość"));

        if(category == Category.BOOKS){
            int pages = toInt(params.get("Liczba stron"));
            boolean cover = toBoolean(params.get("Twarda oprawa"));

            return new ItemBook(name, category, price, quantity, pages, cover);
        }
        if(category == Category.ELECTRONICS){
            boolean mobile = toBoolean(params.get("Mobilny"));
            boolean warranty = toBoolean(params.get("Gwarancja"));

            return new ItemElectronics(name, category, price, quantity, mobile, warranty);
        }
        if(category == Category.FOOD){
            Date expiryDate = toDate(params.get("Data przydatności do spożycia"));

            return new ItemFood(name, category, price, quantity, expiryDate);
        }
        if(category == Category.MUSIC){
            boolean video = toBoolean(params.get("Wideo"));
            MusicGenre genre = MusicGenre.valueOf(params.get("Gatunek").toUpperCase());

            return new ItemMusic(name, category, price, quantity, video, genre);
        }
        if(category == Category.SPORT){
            return new ItemSport(name, category, price, quantity);
        }

        return null;
    }
```

Do klasy dodano parę metod, które uproszczą konwersję typów niektórych danych:

```java
public static int toInt(String s){
    return Integer.parseInt(s);
}

public static boolean toBoolean(String s){
    return Boolean.parseBoolean(s);
}

public static Date toDate(String s){
    SimpleDateFormat formatter = new SimpleDateFormat("dd-MM-yyyy HH:mm");
    Date parsedDate;
    try{
        parsedDate = formatter.parse(s);
    }catch (ParseException e) {
        e.printStackTrace();
        parsedDate = new Date();
    }
    return parsedDate;
}
```

Do plików .csv dodano odpowiednie dane. Program został uruchomiony, aby zweryfikować zmiany. Przykładowy widok szczegółów przedmiotu zaprezentowano poniżej

Zadanie 2.2 Rozszerzenie panelu

Pierwszym krokiem było przerobienie klasy *ItemFilter* tak, aby tworzyła ona obiekt *itemSpec* w zależności od kategorii, w której się znajdujemy. Zostało to rozwiązane przez konstruktor oraz dodanie nowej metody do klasy *ItemFactory*

```
public class ItemFilter {

    private Item itemSpec;

    public ItemFilter(Category cat){
        this.itemSpec = ItemFactory.getItem(cat);
    }
```

```
    public static Item getItem(Category category){
        if(category == null){
            return new ItemBook();
        }

        if(category == Category.BOOKS){
            return new ItemBook();
        }
        if(category == Category.ELECTRONICS){
            return new ItemElectronics();
        }
        if(category == Category.FOOD){
            return new ItemFood();
        }
        if(category == Category.MUSIC){
            return new ItemMusic();
        }
        if(category == Category.SPORT){
            return new ItemSport();
        }

        return null;
    }
```

Po wnikliwej analizie mechanizmu sortowania stwierdzono, że najlepiej będzie jeśli obiekty typu *Item*, będą zgłaszały swoje pola, które są typu *boolean*, a także będą udostępniać metody do ustawiania/pobierania tych pól. Stworzone zostaną także funkcje zwracające mapy typu *<String,Function>*, dzięki którym uproszczona zostanie implementacja filtrowania dla wszystkich elementów na raz. Mapy te będą tworzone w funkcji nadrzędnej, a obiekty dziedziczące będą mogły dodawać swoje wartości do danych mapy. Przy okazji przekonwertowano klasę *Item* na klasę abstrakcyjną.

```
public abstract class Item {
    private String name;
    private Category category;
    private int price;
    private int quantity;
    private boolean secondhand;
    private boolean polish;

    protected Map<String, Consumer<Boolean>> booleanSettersMap = new HashMap<>();
    protected Map<String, Supplier<Boolean>> booleanGettersMap = new HashMap<>();

    public Item(String name, Category category, int price, int quantity) {
        this();
        this.name = name;
        this.category = category;
        this.price = price;
        this.quantity = quantity;
    }

    public Item() {
        booleanSettersMap.put("Tanie bo polskie", this::setPolish);
        booleanGettersMap.put("Tanie bo polskie", this::isPolish);

        booleanSettersMap.put("Używany", this::setSecondhand);
        booleanGettersMap.put("Używany", this::isSecondhand);
    }
```

```
    public Map<String, Consumer<Boolean>> getPropertiesSetters(){
        return booleanSettersMap;
    }
    public Map<String, Supplier<Boolean>> getPropertiesGetters(){
        return booleanGettersMap;
    }
```

Przykładowa klasa dodająca swoje metody do mapy:

```java
public class ItemBook extends Item {
    private int pageNumber;
    private boolean hardcover;

    public ItemBook(String name, Category category, int price, int quantity, int pages, boolean cover){
        super(name, category, price, quantity);
        this.pageNumber = pages;
        this.hardcover = cover;
        this.registerHooks();
    }

    public ItemBook(){
        super();
        this.registerHooks();
    }

    private void registerHooks(){
        booleanSettersMap.put("Twarda oprawa", this::setHardcover);
        booleanGettersMap.put("Twarda oprawa", this::isHardcover);
    }

    public boolean isHardcover() {
        return hardcover;
    }

    public void setHardcover(boolean hardcover) {
        this.hardcover = hardcover;
    }
```

Operacja dodawania checkboxów do filtrowania danych uprościła i skróciła się znacząco:

```java
    public void fillProperties() {
        removeAll();

        ItemFilter filter = new ItemFilter(shopController.getCurrentCategory());
        filter.getItemSpec().setCategory(shopController.getCurrentCategory());

        var setters = filter.getItemSpec().getPropertiesSetters();
        for (String displayName : setters.keySet()) {
            add(createPropertyCheckbox(displayName, event -> {
                setters.get(displayName).accept(((JCheckBox) event.getSource()).isSelected());

                shopController.filterItems(filter);
            }));
        }
    }
```

Tak samo sama procedura filtrowania:

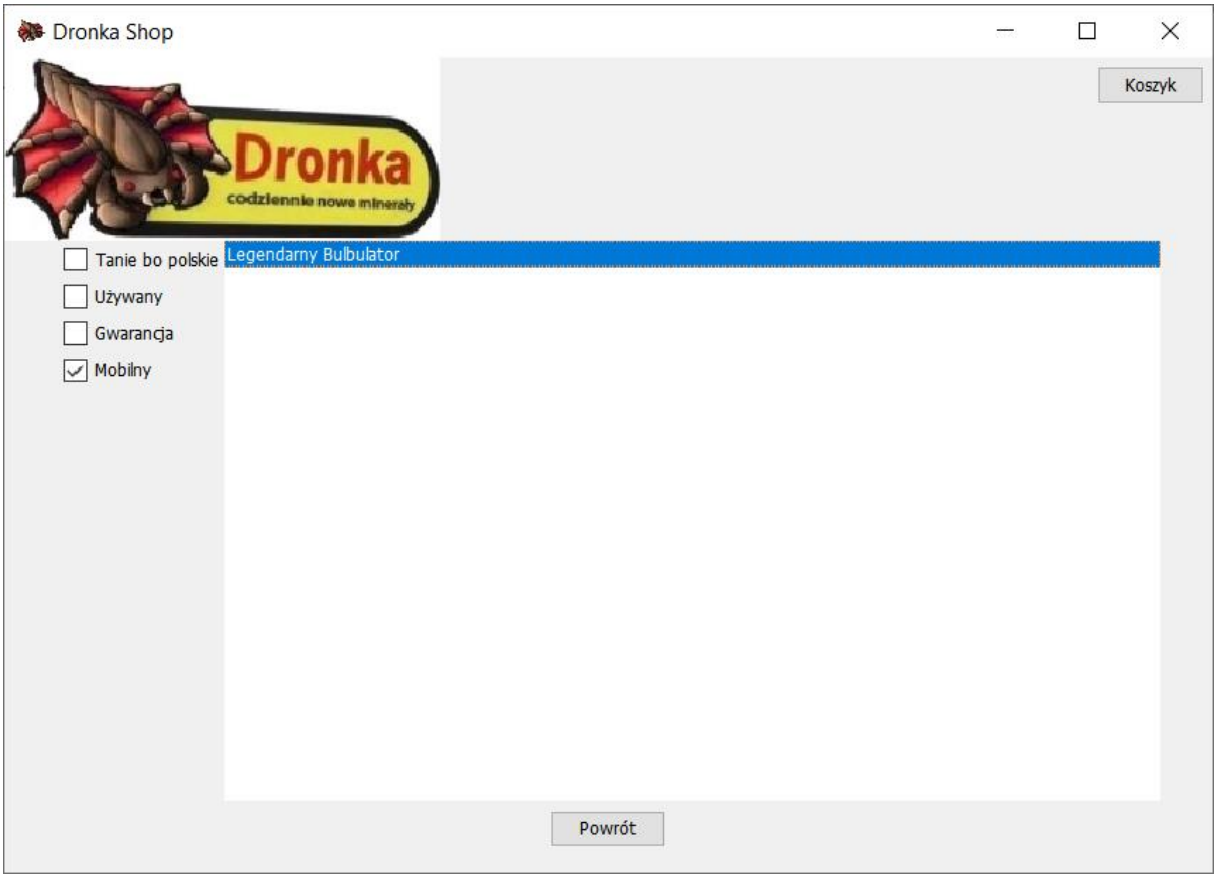
```
public ItemFilter(Category cat){
    this.itemSpec = ItemFactory.getItem(cat);
}

public Item getItemSpec() {
    return itemSpec;
}
public boolean appliesTo(Item item) {
    if (itemSpec.getName() != null
            && !itemSpec.getName().equals(item.getName())) {
        return false;
    }
    if (itemSpec.getCategory() != null
            && !itemSpec.getCategory().equals(item.getCategory())) {
        return false;
    }

    var getters = itemSpec.getPropertiesGetters();
    var getters_toCheck = item.getPropertiesGetters();
    for (String displayName : getters.keySet()) {
        var itemSpecValue = getters.get(displayName).get();
        var itemToCheckValue = getters_toCheck.get(displayName).get();
        if(itemSpecValue && !itemToCheckValue){
            return false;
        }
    }

    return true;
}
```

Dzięki zaprezentowanym mapom, możliwe jest dodanie nowych wartości typu *boolean* do klas dziedziczących z *Item* i zostaną one od razu uwzględnione w filtrowaniu.

Po głębszym zastanowieniu się nad zaprezentowanym rozwiązaniu stwierdzam, iż ciężko byłoby dopasować sensowny wzorzec projektowy, ponieważ klasy nie są jednolite (mają różną ilość pól typu *boolean*) oraz różne metody do ustawiania/pobierania tychże pól.

Po uruchomienu programu przetestowano zaimplementowane zmiany. Przykładowe widoki po zastosowaniu filtrów pokazano poniżej:


Dronka Shop
Koszyk
Dronka
codziennie nowe minerały
Tanie bo polskie
Używany
Twarda oprawa
Nowe Przygody Gangu Czworga
When The Smoke is Going Down : Wprowadzenie do testów wydajnościowych
Powrót


Dronka Shop
Koszyk
Dronka
codziennie nowe minerały
Tanie bo polskie
Używany
Gwarancja
Mobilny
Legendarny Bulbulator
Powrót