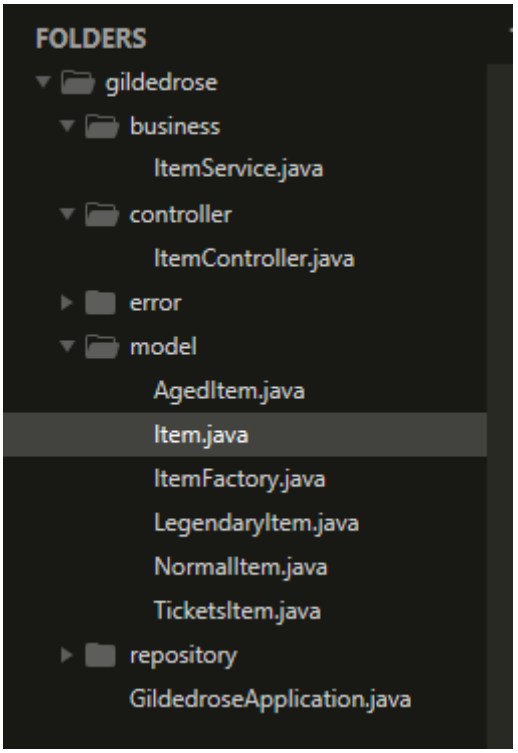
GildedRose Refactorization

By: Juan Jose Salazar Cardona& Juan Guillermo Sarrias Escudero

The structure of the project directories was maintained, but a number of classes were added to the "model" directory, as shown in the image below:



Among the previous classes of model directory, the "ItemFactory" class is used to try to apply the "Factory Method" design pattern, with which it is intended that the "Item" instances are created with static methods, only in the Factory.

Following this logic, the classes "Aged item", "Normal item", "Legendary item" and "Tickets item" inherit from the Item class and overload the method in charge of updating the quality of the items.

- **Normal Item class:**

NormalItem.java

```java
package com.perficient.praxis.gildedrose.model;


public class NormalItem extends Item {

    public NormalItem(Item item) {
        super(item.getId(), item.name, item.sellIn, item.quality, Type.NORMAL);
    }

    @Override
    public Item updateQuality() {
        this.decreaseSellIn();
        if (this.isExpired()) {
            this.decreaseQualityBy(2);
        } else {
            this.decreaseQualityBy(1);
        }
        return this;
    }
}
```

- **Aged Item class:**

AgedItem.java

```java
package com.perficient.praxis.gildedrose.model;


public class AgedItem extends Item {

    public AgedItem(Item item) {
        super(item.getId(), item.name, item.sellIn, item.quality, Type.AGED);
    }

    @Override
    public Item updateQuality() {
        this.decreaseSellIn();
        if (this.isExpired()) {
            this.increaseQualityBy(2);
        } else {
            this.increaseQualityBy(1);
        }
        return this;
    }
}
```

- **Legendary Item class:**

LegendaryItem.java

```java
package com.perficient.praxis.gildedrose.model;


public class LegendaryItem extends Item {

    public LegendaryItem(Item item) {
        super(item.getId(), item.name, item.sellIn, item.quality, Type.LEGENDARY);
    }

    @Override
    public Item updateQuality() {
        return this;
    }
}
```

- **Tickets Item class:**

TicketsItem.java

```java
package com.perficient.praxis.gildedrose.model;

public class TicketsItem extends Item {

    public TicketsItem(Item item) {
        super(item.getId(), item.name, item.sellIn, item.quality, Type.TICKETS);
    }

    @Override
    public Item updateQuality() {
        this.decreaseSellIn();
        if (this.isExpired()) {
            this.decreaseQualityBy(this.quality);
        } else if (remainingDaysForConcert() <= 5) {
            this.increaseQualityBy(3);
        } else if (remainingDaysForConcert() <= 10) {
            this.increaseQualityBy(2);
        } else {
            this.increaseQualityBy(1);
        }
        return this;
    }

    public int remainingDaysForConcert() {
        return this.sellIn;
    }

}
```

- **The Item class:**

The methods shown in the image below were added to the "Item" class.

- The "updateQuality" method of the Item was made to reduce the coupling that the ItemService class had in his previous “updateQuality” method, making that only the Item and its subclasses are responsible for updating its quality attribute.
- The methods "increaseQualityBy" and "decreaseQualityBy" were created to receive an amount of quality which will be subtracted or added to the current quality of the Item, without going out of the ranges established by the business rules (quality between 0 and 50). These also facilitate the reading of the code.
- The "decreaseSellIn" and “isExpired” methods was created for readability when decreasing the remaining days for item expiration and checking if the item is expired.

With this change, we allow each of the classes (aged, legendary, etc) to be in charge of updating the quality of the item, thus reducing responsibilities in the updateQuality() method.

- **Item service class:**

ItemService.java

```java
public List<Item> updateQuality() {
    ArrayList<Item> items = new ArrayList<>(itemRepository.findAll());
    ListIterator<Item> listPosition = items.listIterator();
    while (listPosition.hasNext()) {
        Item actualItem = listPosition.next();
        actualItem = ItemFactory.createTypedItem(actualItem);
        actualItem.updateQuality();
        actualItem = ItemFactory.createUntypedItem(actualItem);
        listPosition.set(actualItem);
        itemRepository.save(actualItem);
    }
    return items.stream().toList();
}
```

In this method you can see the implementation of the above mentioned, where for each of the items in the list, it types them and depending on the type of the item, the method of the corresponding class is applied to it. Then, in order to save the item, it is returned to its default type.

- **Item factory class:**

ItemFactory.java

```java
package com.perficient.praxis.gildedrose.model;

import com.perficient.praxis.gildedrose.error.ResourceNotFoundException;

public class ItemFactory {
    public static Item createTypedItem(Item baseItem) {
        Item typedItem;
        switch (baseItem.type) {
            case NORMAL:
                typedItem = new NormalItem(baseItem);
                break;
            case AGED:
                typedItem = new AgedItem(baseItem);
                break;
            case LEGENDARY:
                typedItem = new LegendaryItem(baseItem);
                break;
            case TICKETS:
                typedItem = new TicketsItem(baseItem);
                break;
            default:
                throw new ResourceNotFoundException("a");

        }
        return typedItem;
    }

    public static Item createUntypedItem(Item typed) {
        Item untypedItem = new Item(typed.getId(), typed.name, typed.sellIn, typed.quality, typed.type);
        return untypedItem;
    }
}
```

Now passing to the item factory (the basis of this process), the type of item is validated and depending on that a new instance of the class corresponding to the type is created.