Högskolan i Gävle

Miniprojekt 2

Dokument

Hanna Medén, Niklas Nordgren

2020-02-02

Miniprojekt 2 i kursen Objektorienterad design och programmering II, DVG503

Innehållsförteckning

1 Inledning

I detta projekt skulle det implementeras ett back-end system för att generera och manipulera textdokument. Kriterierna för implementationen var att den skulle innehålla några olika lämpliga designmönster samt att programmeringen skulle ske på en abstrakt nivå för att öppna upp för återanvändning och utökning av koden.

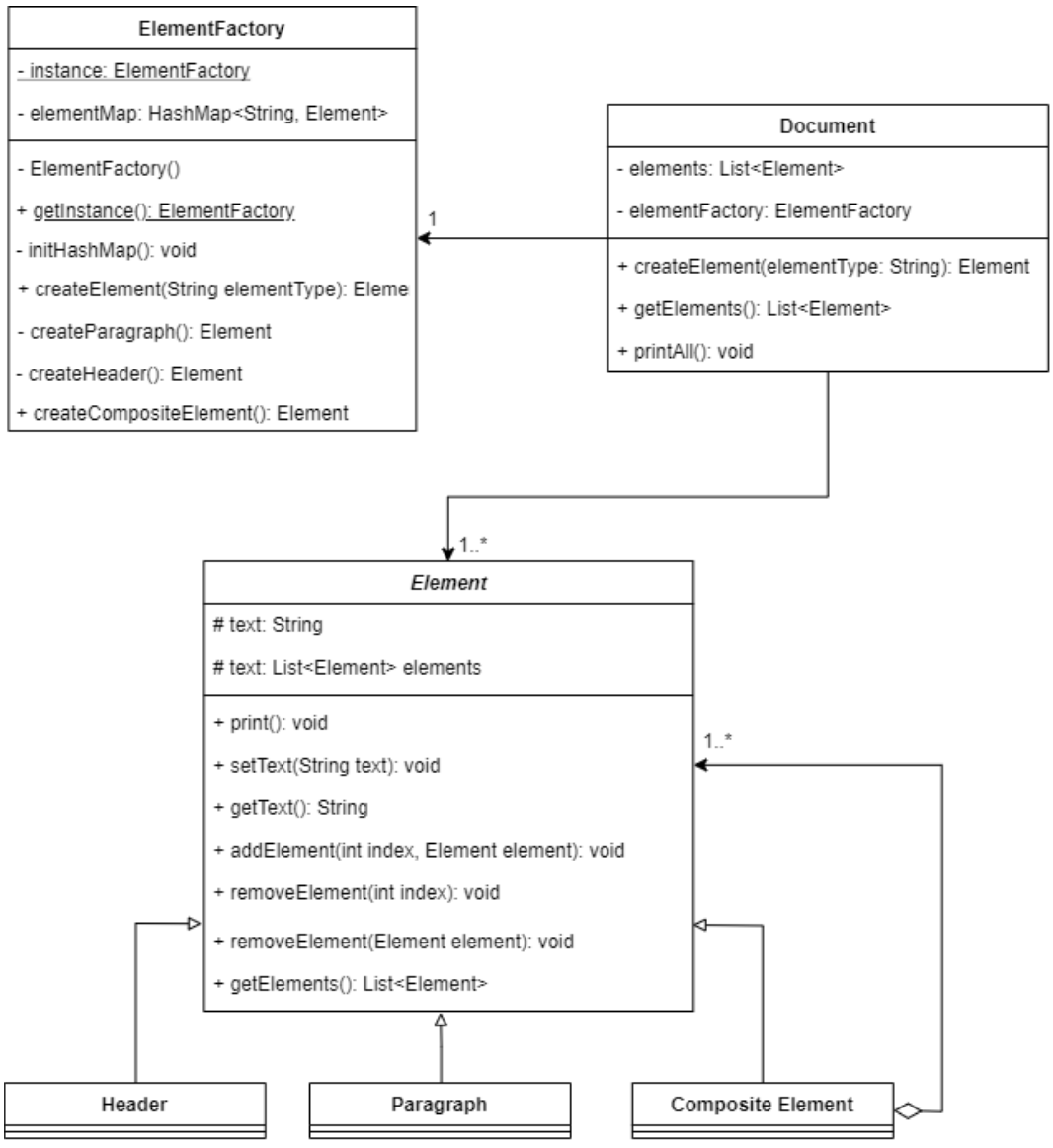
2 Metod

Inledningsvis identifierades vilka klasser som skulle ingå i systemet. Vi valde att bygga upp det på så sätt att det finns en klass `Document` som har en lista bestående av objekt av typen `Element`. De olika elementen är några olika delar som kan finnas i ett dokument, såsom `Header`, `Paragraph` och `CompositeElement`. Dessa är subklasser till superklassen `Element`. Därefter började arbetet med att identifiera de olika attribut och metoder varje klass behövde, och vad som var gemensamt för alla och därmed skulle finnas i den abstrakt klassen `Element`. Attribut som vi ansåg alla elementen krävde var ett – text. Till detta attribut skapades set/get-metoder. Det skapades även en lista av element samt metoden `addElement(int index, Element element)` och `getElements()` som `CompositeElement` instansierar och implementerar. Klassen `Element`'s version av `addElement(int index, Element element)`-metoden utför ingenting utan existerar endast för att subklassen `CompositeElement` ska kunna implementera metoden genom överskuggning, samma sak gäller även för metoden `getElements()` som returnerar värdet `null` i klassen `Element` och listan av alla element som ett `CompositeElement` håller i. Även två olika varianter av publika borttagningsmetoder som verkar på listan `Elements` skapades där den ena tar emot ett index för objektet som skall tas bort och den andra det objekt som skall tas bort från listan. Klassen `Element` definierades som en abstrakt klass vilket motverkar att objekt av den klassen instansieras och implicit att metoderna som saknar beteende används på ett felaktigt sätt.

3 Resultat

Det slutgiltiga programmet är ett väldigt generiskt sådant där man genom hårdkodning kan skapa element, bestämma texter för dessa och även hämta samt skriva ut dem. Alla element hamnar i en lista som finns i *Document* där man kan iterera igenom alla element i listan och det finns även en metod för att skriva ut dem till konsolen. Composite element byggdes på så sätt att det i sig själv är en lista av element, där kan användaren lägga till flera olika element, exempelvis en rubrik följt av två paragrafer – ett avsnitt i en bok.

Det finns även en *ElementFactory*, den används av *Document* för att skapa olika *Element*. *ElementFactory* är som namnet berättar – av designmönstret Factory och även Singleton. *Document* fungerar som en Facade i detta projekt och innehåller implicit även en Iterator. Element och dess relation till subklasserna följer mönstret Composite.



Figur 1: UML klassdiagram av systemet.

4 Diskussion

Vi tyckte det var väldigt svårt att arbeta med så hög abstraktionsnivå. Skulle hjälpt mycket att ha någon form av konkretisering, hur det ska användas (utöver att vi sedan ska lägga till Visitor) och hur det ska användas rent praktiskt.

Såvida vi inte missförstått uppgiften väldigt mycket (tillsammans med några kurskamrater) så anser vi att det var en lagom stor uppgift för en vecka och det svåraste var som sagt att få ner faktisk kod trots så hög abstraktion. Något vi tydligen behöver jobba mer med.

5 Bilaga – kodlistningar

5.1 Klassen Element

```java
package composite;

import java.util.List;

public abstract class Element {

    protected String text;
    protected List<Element> elements;

    public void print() {
        System.out.println(text);
    }

    public void setText(String text) {
        this.text = text;
    }

    public String getText() {
        return this.text;
    }

    public void addElement(int index, Element element) {
    }

    public void removeElement(int index) {
    }

    public void removeElement(Element element) {
    }

    public List<Element> getElements() {
        return this.elements;
    }

}
```

5.2 Klassen CompositeElement

```java
package composite;

import java.util.ArrayList;

public class CompositeElement extends Element {

    public CompositeElement() {
        this.elements = new ArrayList<Element>();
    }

    @Override
    public void print() {
        for (Element e : elements)
            System.out.println(e.text);
    }

    @Override
    public void addElement(int index, Element element) {
        this.elements.add(index, element);
    }

    @Override
    public void removeElement(int index) {
        this.elements.remove(index);
    }

    @Override
    public void removeElement(Element element) {
        this.elements.remove(element);
    }

    @Override
    public List<Element> getElements() {
        return this.elements;
    }

    @Override
    public void setText(String text) {
    }

}

```

5.3 Klassen Header

```java
package composite;

public class Header extends Element {

    public Header() {

    }

}

```

5.4 Klassen Paragraph

```java
package composite;

public class Paragraph extends Element {

    public Paragraph() {

    }

}
```

5.5 Klassen ElementFactory

```java
import java.util.HashMap;

public class ElementFactory {

    private static ElementFactory instance = null;
    private HashMap<String, Element> elementMap;

    private ElementFactory() {
        initHashMap();
    }

    public static ElementFactory getInstance() {
        if (instance == null)
            instance = new ElementFactory();
        return instance;
    }

    private void initHashMap() {
        elementMap = new HashMap<String, Element>();
        elementMap.put("paragraph", this.createParagraph());
        elementMap.put("header", this.createHeader());
        elementMap.put("compositeelement", this.createCompositeElement());
    }

    public Element createElement(String elementType) {
        if (elementMap.containsKey(elementType))
            return elementMap.get(elementType);
        return null;
    }

    private Element createParagraph() {
        return new Paragraph();
    }

    private Element createHeader() {
        return new Header();
    }

    private Element createCompositeElement() {
        return new CompositeElement();
    }

}
```

5.6 Klassen Document

```java
import java.util.ArrayList;

public class Document {

    private List<Element> elements;
    private ElementFactory elementFactory;

    public Document() {
        this.elementFactory = ElementFactory.getInstance();
        this.elements = new ArrayList<Element>();
    }

    public Element createElement(String elementType) {
        Element element = this.elementFactory.createElement(elementType);
        this.elements.add(element);
        return element;
    }

    public List<Element> getElements() {
        return this.elements;
    }

    public void printAll() {
        for (Element e : elements) {
            e.print();
        }

    }
}

```

5.7 Klassen Main

```java
import composite.Element;

public class Main {

    public static void main(String[] args) {

        Document document = new Document();

        Element paragraph = document.createElement("paragraph");
        paragraph.setText("Test paragraph");

        Element header = document.createElement("header");
        header.setText("Test header");

        Element compositeElement = document.createElement("compositeelement");
        compositeElement.setText("Test");

        compositeElement.addElement(compositeElement.getElements().size(), paragraph);
        compositeElement.addElement(compositeElement.getElements().size(), header);

        document.printAll();

    }

}

```