Practicum OO – lente 2016 **Opgave 9**

Het visitor pattern

Leerdoelen

In deze opgave vragen we je om in Java een representatie voor logische formules te ontwerpen en te implementeren. Na afloop van deze opdracht ben je in staat om:

- Recursieve datastructuren te realiseren gebaseerd of zelfverwijzende klassen;
- een geschikt interface/geschikte abstracte klasse te introduceren als basis voor de representatie van de verschillende soorten knopen in een expressieboom;
- elke concrete knoop te definiëren als uitbreiding van deze basisklasse;
- nieuwe operaties te introduceren gebaseerd op het visitor pattern;
- te kunnen aangeven wat de voor- en de nadelen zijn dit design pattern t.o.v. een oplossing waarbij de nieuwe operaties als (abstracte) methode aan de basisklasse worden toegevoegd;
- UML-diagrammen (klassendiagram en sequentie-diagram) te gebruiken bij het ontwerp van een objectgeoriënteerd programma.

1 Logische formules

Deze opgave gaat over het representeren en manipuleren van expressies uit de propositielogica. Een *logische expressie* (ook wel *formule* genoemd) is opgebouwd uit constantes (*true* en *false*), atomaire formules (hier gerepresenteerd als een String) gecombineerd met de logische operatoren $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies) en $\neg$ (not). We kunnen de *syntax* van deze formules als volgt compact weergeven:

$$F ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{String} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Rightarrow F_2 \mid \neg F$$

Een concrete formule kunnen we als platte tekst weergeven, maar ook in boomvorm. De bladeren in zo'n boom komen overeen met constantes of atomaire formules en iedere interne knoop met een toepassing van één van de operatoren. Dit betekent dat sommige interne knopen 2 opvolgers en andere precies 1 opvolger zullen hebben. De boomvorm heeft als voordeel dat het vaak makkelijker is om (ingewikkelde) operaties op formules te definiëren.

In deze opdracht gaan we derhalve gebruik maken van een boomrepresentatie van formules. Deze lijkt enigszins op de representatie die gebruikt werd voor Quadtrees. We gebruiken weer een interface waarmee we alle knopen uit de formuleboom aanduiden en introduceren voor ieder knooptype een aparte concrete klasse die dit interface implementeert. Wat we dit keer anders doen t.o.v. de Quadtrees is de manier waarop we nieuwe operaties op bomen definiëren. In het Quadtrees voorbeeld werden operaties als abstracte methodes aan het basisinterface toegevoegd. In deze opdracht laten we het basisinterface ongewijzigd en gebruiken we het *visitor pattern*.

Op het college is uitgelegd dat het visitor pattern gebaseerd is op twee interfaces (enige gelijkenis vertonend met het *observer pattern*). Eén van beide interfaces is uiterst eenvoudig:

```
public interface Visitable {
    void accept( FormVisitor visitor );
}
```

Alle concrete knopen uit de boom dienen dit interface te implementeren. Het hierin gebruikte type `FormVisitor` is zelf ook interface.

```
public interface FormVisitor {
    void visit( Form form );
}
```

Dit interface dient straks nog te worden aangepast. We kunnen nu het basisinterface voor de representatie van formules geven.

```
public interface Form  extends Visitable {
}
```

Opmerking: Het gebruik van het interface `Visitable` is een beetje overkill: we hadden net zo goed de abstracte methode uit dit interface direct in `Form` op kunnen nemen. Het interface is ook niet in andere andere situaties bruikbaar omdat het gebruik maakt van `FormVisitor`. En `FormVisitor` is onlosmakelijk met de gekozen representatie van formules verbonden.

Zoals gezegd worden de concrete knopen in de formuleboom klasses die `Form` implementeren. Een voorbeeld is de klasse voor *and*-knopen:

```
public class AndForm implements Form  {
    private Form leftOperand;
    private Form rightOperand;

    public AndForm( Form left, Form right ) {
        this.leftOperand  = left;
        this.rightOperand = right;
    }

    public Form getLeft(){
        return leftOperand;
    }

    public Form Right(){
        return leftOperand;
    }

    public void accept( FormVisitor v ) {
        v.visit( this );
    }
}
```

Deze definitie zou voor zich moeten spreken. Aangezien `Form` de abstracte methode `accept` bevat zijn we verplicht om hier een definitie van deze methode te geven. Deze heeft (nagenoeg) altijd bovenstaande vorm.

Zodra we op vergelijkbare wijze klasses voor alle andere knopen aan het project hebben toegevoegd moeten we `FormVisitor` nog aanpassen. Dit interface bestaat nu nog uit één abstracte methode `visit`. Deze wordt echter vervangen door nieuwe abstracte methodes, en wel één voor ieder concreet knooptype.

2 Opdrachten

1. Completeer bovengegeven representatie: Voeg klasses toe (met geschikte attributen) voor ieder knoop-type en pas `FormVisitor` aan. Geef voordat je met de implementatie start een UML *klassendiagram* van alle gebruikte klasses.

2. Voeg in je hoofdklasse een aantal statische methodes toe die elk een logische formuleboom opbouwen. Bedenk zelf een aantal interessante gevallen. Deze ga je straks gebruiken om je operaties uit te testen.

3. Definieer een klasse `PrintFormVisitor` die het `FormVisitor` interface implementeert en die als resultaat de formule als gewone platte tekst op het scherm zet. Probeer de formule met zo min mogelijk haakjes af te drukken door gebruik te maken van de prioriteiten van de verschillende operatoren. De gebruikelijke volgorde van hoge naar lage prioriteit is: $\neg, \wedge, \vee, \Rightarrow$. Dus $\neg$ bindt sterker dan $\wedge$, etc.

4. Laat d.m.v. een *sequence diagram* de interacties zien tussen verschillende objecten. Dit zou inzicht moeten geven in de werking van het visitor pattern.

Door aan iedere propositievariabele een waarheidswaarde toe te kennen (een functie die zoiets doet wordt ook wel een *valuatie* genoemd) kunnen we de waarde van een formule uitrekenen. Je gaat nu een `EvalFormVisitor` maken die voor een gegeven valuatie de waarde van een formule bepaalt. Als valuatie kun je het beste een Java `Map<K,V>` gebruiken. Omdat `EvalFormVisitor` iets uitrekent is het handig om de interfaces van het visitor pattern aan te passen zodat zowel `visit` als `accept` iets op kunnen leveren. We zetten dit algemeen op door gebruik te maken van *generics*.

```
public interface Visitable {
    public <R> R accept( FormVisitor<R> visitor );
}

public interface FormVisitor<R> {
    R visit( NotForm form );
    ...
}
```

5. Pas je klasses en interfaces aan zodat gebruik wordt gemaakt van deze nieuwe interfaces. Zorg ervoor dat je `PrintFormVisitor` ook weer werkt.

6. Implementeer de `EvalFormVisitor` en test deze uit met verschillende valuaties en verschillende formules.

Inleveren

Vóór zondag 17 april, 23:59 uur, via Blackboard.