Het is weer lente en tijd voor de grote lenteschoonmaak. Alle spullen naar buiten, schoonkloppen, schoonmaken en de meubels kunnen weer naar binnen. Uiteraard gaat dan alles niet terug op dezelfde plek als daarvoor. Na de grote schoonmaak begint het grote herinrichten, met als resultaat een gerefactorde woonkamer.

thema

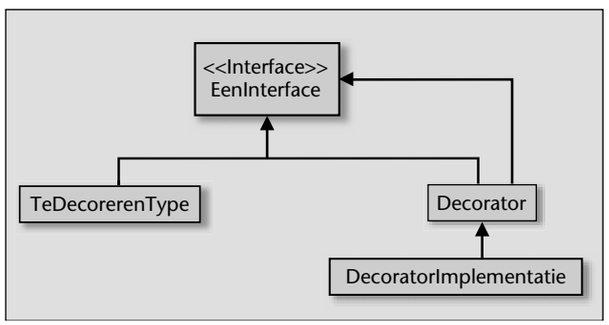
Het zijn maar patronen (4)

Decorator: code opnieuw inrichten

De spullen zijn grotendeels hetzelfde, maar de plaats is anders. Dezelfde gegevens, maar met een ander perspectief. Zoals je je huis kunt laten inrichten door een 'interior decorator', kun je het design pattern Decorator gebruiken om anders tegen je code of de gegevens aan te kijken zonder deze zelf te veranderen. Een kwestie van inrichten.

Patronen zijn er in vele kleuren en vormen. Elke vorm kan herhaald worden om zo herkenning te vereenvoudigen en weer andere patronen te vormen. In het geval van gedrag is het een kwestie van conditionering. Van kinds af aan leren we in patronen te kijken en te denken. Een collectie van een mond, neus en twee ogen is voor iedereen bijna meteen een gezicht. Het afzonderlijk bekijken en beoordelen van de verschillende componenten neemt teveel tijd in beslag. Daarnaast hebben we geleerd hoe een gezicht er uitziet en hoe we dit kunnen herkennen.

Die patroonherkenningsvaardigheden zijn van grote waarde in ons dagelijkse leven om zo snel mogelijk belangrijke informatie te kunnen verwerken. Door veel met Java te werken zul je eveneens in toenemende mate de zich herhalende concepten zien. Meer dan eens zie je componenten die doen wat jij wilt maar net niet helemaal. Je kunt dan snel een selectie maken uit een verzameling voor een bewerking op slechts een gedeelte van die verzameling, om dat vervolgens in een gesorteerde en genummerde tabel te tonen, met totalen onder de kolommen met een getal erin. In het hierna volgende zal ik een aantal van deze voorbeelden uitwerken, in de hoop dat je het patroon (in dit geval de Decorator) zult ontdekken.



FIGUUR 1. Class diagram Decorator

DECORATOR Volgens van Dale is de betekenis van Decorator als volgt:

Decorator ~ **0.1** ***afwerker (v. huis)*** => *(huis)schilder, stukadoor, behanger*

0.2 interior decorator *binnenhuis architect* [2]

Decorate ~ **0.1** ***afwerken*** => *verven; schilderen; behangen*

0.2 versieren => *verfraaien*

0.3 decoreren => *ridderen, onderscheiden* [2]

Net als bij het afwerken en versieren van een huis gebruiken we de Decorator om onvolkomenheden te verbergen. We creëren een andere voorstelling van zaken. De interface blijft dezelfde, maar de gegevens zijn anders. In de Java programmeertaal worden de filters in de java.io package (te) veel gebruikt als voorbeeld voor de Decorator. Alle stromen lezen uiteindelijk informatie, maar zij doen dat ieder op hun eigen wijze, waarbij zij echter wel op elkaar aangesloten worden. Een voorbeeld daarvan is:

```
BufferedReader reader = new
BufferedReader(new FileReader(new File("")));
```

De BufferedReader instantie werkt nu als Decorator voor de FileReader. Als je de FileReader uit wilt lezen moet dat per karakter array. De BufferedReader voegt nu echter extra functionaliteit toe: het lezen van hele regels in één keer. Het grote voordeel is dat FileReader ongewijzigd blijft. Mocht je op een andere manier gegevens willen lezen uit de file gebruik je een andere reader.

Als voorbeeld voor de Decorator gaan we nummering toevoegen aan een *javax.swing.table.TableModel*.

```
package decorator;

import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.AbstractTableModel;
import javax.swing.table.TableModel;

public class TableModelDecorator
   extends AbstractTableModel
   implements TableModelListener {

  private TableModel model;

  public TableModelDecorator(TableModel model)
{
     this.model = model;
     this.model.addTableModelListener(this);
  }

  public void tableChanged(TableModelEvent
event) {
     this.fireTableChanged(event);
  }

  public int getColumnCount() {
     return (model == null) ? 0 :
model.getColumnCount();
  }

  public int getRowCount() {
     return (model == null) ? 0 :
model.getRowCount();
  }

  public Class getColumnClass(int column) {
     return (model == null) ? null :
model.getColumnClass(column);
  }

  public String getColumnName(int column) {
     return (model == null) ? null :
model.getColumnName(column);
  }

  public boolean isCellEditable(int row, int
column) {
     return model.isCellEditable(row, column);
  }
```

TableModelDecorator bevat de basis functionaliteit van de decorators. De TableModelDecorator heeft een referentie naar het onderliggende TableModel. De methoden in de AbstractTableModel worden allemaal netjes doorgespeeld aan het onderliggende model.

```
package decorator;

import javax.swing.table.TableModel;

public class NummerDecorator extends
TableModelDecorator {
  public NummerDecorator(TableModel model) {
     super(model);
  }

  public int getColumnCount() {
     return super.getColumnCount() + 1;
  }

  public Class getColumnClass(int column) {
     return (column == 0)
        ? Integer.class
        : super.getColumnClass(column - 1);
  }

  public String getColumnName(int column) {
     return (column == 0)
       ? "Nummer"
       : super.getColumnName(column - 1);
  }

  public boolean isCellEditable(int row, int
column) {
     return (column == 0)
        ? false
        : super.isCellEditable(row, column -
1);
  }

  public Object getValueAt(int row, int
column) {
     return (column == 0)
        ? new Integer(row + 1)
        : super.getValueAt(row, column - 1);
  }

  public void setValueAt(Object value, int
row, int column) {
     super.setValueAt(value, row, column -
1);
  }
}
```

De NummerDecorator decoreert de tablemodel met een extra eerste kolom met daarin de regelnummers.

```
package decorator;
import javax.swing.table.TableModel;
```

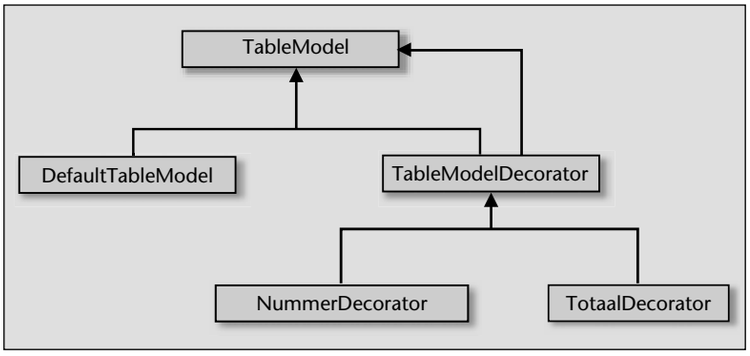
```
public class TotaalDecorator extends
TableModelDecorator {
  public TotaalDecorator(TableModel model) {
     super(model);
  }

  public int getRowCount() {
     return super.getRowCount() + 1;
  }

  public boolean isCellEditable(int row, int
column) {
     return (row == this.getRowCount()-1)
        ? false
        : super.isCellEditable(row, column);
  }

  public Object getValueAt(int row, int
column) {
     return (row == this.getRowCount()-1)
        ? maakTotaal(column)
        : super.getValueAt(row, column);
  }

  private Object maakTotaal(int column) {
     int value = 0;
     int rowCount = super.getRowCount();
     for (int row=0; rowCount > 0 &&
getColumnClass(column).equals(Integer.class)
; row++) {
        value += ((Integer)getValueAt(row,
column)).intValue();
     }
     return new Integer(value);
  }

  public void setValueAt(Object value, int
row, int column) {
     super.setValueAt(value, row, column -
1);
  }
}
```

De TotaalDecorator voegt als laatste row een totaal toe, waarbij de waarde gelijk is aan de som van de getallen in de kolom:

```
    TableModel model = new
TotaalDecorator(new NummerDecorator(model1));
```

Je ziet hier de stapelende werking van de Decorator.

VARIATIES Het voorbeeld dat hier is beschreven, is geen 'pure' Decorator. Bij een Decorator zoals die in dit artikel is uitgewerkt, is de volgorde van het toevoegen van de decorators van belang. Een pure Decorator voegt alleen functionaliteit toe door het toevoegen van methoden en hoeft de volgorde niet van belang zijn.



FIGUUR 2. Class Diagram nummer 2

VOOR- EN NADELEN De Decorator maakt het mogelijk functionaliteit toe te voegen of te verwijderen van een component. De functionaliteit hoeft dus niet meer in de oorspronkelijke component te worden opgenomen, maar zit nu in een apart type. Je krijgt een beter overzicht. Wellicht dat je de decorators op meerdere plaatsen kunt gebruiken. Het gevaar is dat er teveel decorators gebruikt worden, waardoor het ontstane overzicht weer teniet wordt gedaan. Het testen en debuggen wordt in zo'n situatie een uitdaging.

HERKENNEN EN TOEPASSEN Voor het implementeren van de Decorator implementeer je de volgende zaken:

- *WerkelijkeComponent* - hoeft waarschijnlijk niet geprogrammeerd te worden. Hoogstens dat de class moet declareren dat deze de ComponentInterface implementeerd.
- *ComponentInterface* - de externe interface van het te decoreren type.
- *Decorator* - Abstract class met het default gedrag voor een Decorator.
- *DecoratorImplementatie* - Implementatie van een bepaalde strategie voor het decoreren.

TENSLOTTE De Decorator is zeer nuttig voor het dynamisch kunnen toevoegen of verwijderen van functionaliteit aan een reeds bestaand type zonder de externe functies of interface te hoeven wijzigen. In de volgende aflevering uit deze serie zullen de factory's aan de beurt komen: Abstract Factory en Factory Method.

LITERATUUR

1 Design Patterns, Erich Gamma, et. al., Addison-Wesley, 1995
2 Van Dale handwoordenboek Engels - Nederlands
3 Applied Java Patterns, Stephen Stelting & Olav Maassen, Prentice Hall, 2002

Olav Maassen (olav.maassen@delion.com) is senior Java developer bij Delion B.V. te Gorinchem en co-auteur van het boek "Applied Java Patterns"[3] samen met Stephen Stelting.