De laatste jaren zijn patterns, met name design patterns, sterk in popularititeit gegroeid. Vaak wordt er vanuit gegaan dat iedereen volledig vertrouwd is met de concepten hierachter. In deze reeks artikelen behandelt Olav Maassen design patterns aan de hand van voorbeelden uit het dagelijks leven die in Java geprogrammeerd zouden kunnen worden. De doelstelling is om design patterns inzichtelijk en begrijpelijk te maken en zo de mystiek weg te nemen en er zelf mee aan de slag te kunnen.

thema

Het zijn maar patronen (7)

'Kun je niet meer als je broer zijn'

'Kijk nou eens naar je broer! Hem lukt het wel; kun je nou niet meer als hem zijn.' Menigeen zal zich dit soort uitspraken kunnen herinneren. Dit is uiteraard meer figuurlijk dan letterlijk bedoeld. Afgezien van de ethische discussie over het klonen van personen, zou je de technologie van het klonen kunnen gebruiken zodra je een voorbeeldig kind hebt. Je krijgt dan een genetische kopie van het origineel.

Voor het persen van muziek-cd's wordt gebruik gemaakt van een 'golden copy'. Alle geperste cd's zijn kopieën van deze ene master cd. Het is zo gezegd het prototype voor de te persen objecten. Hetzelfde principe (het technische wel te verstaan) is ook toepasbaar in Java. Zodra je een object hebt, dat aan je verwachtingen voldoet, kun je deze gaan gebruiken als prototype voor het maken van andere objecten. Zeker als het maken van dit originele object een grote inspanning was, loont dit de moeite.

Prototype Het Prototype pattern gebruik je om een instantie te creëren gebaseerd op een al bestaande instantie en je wil deze instantie kopiëren. Voor het kopiëren van een object heb je binnen Java een aantal mogelijkheden. Een paar mogelijkheden om het Prototype pattern toe te passen:

- Copy constructor – Een constructor in de class die als argument een instantie van dat type heeft
- Clone methode – De class Cloneable laten implementeren en meestal ook de clone() methode overriden.

Het Prototype pattern gebruik je om een instantie te creëren die gebaseerd is op een al bestaande instantie

- Eigen copy methode – In de class een methode toevoegen die als return type hetzelfde type heeft en in de body van de methode een copy maken van deze instantie

Wanneer kun je welke mogelijkheid nu het beste kiezen? Uiteraard is dit altijd afhankelijk van persoonlijke voorkeur, maar hier volgen wat overwegingen:

De copy constructor geeft het meest duidelijk aan wat er gebeurt: er wordt een nieuw object gemaakt (constructor) met als basis een andere instantie van hetzelfde type (het argument). Clone methode is aan te roepen op elk object aangezien deze in Object gedefinieerd is. Nadeel bij deze optie is dat als er een object is dat geen Cloneable implementeert en clone()wordt toch aangeroepen, een exceptie gegooid wordt. Verder zul je in de meeste gevallen nog moeten casten voordat je methoden aan kunt roepen op de instantie.

Bij de laatste optie heb je meer controle aangezien je dan alleen de copy() methode aan kunt roepen op

```
package familie;

public class Kind {
  private String gedrag;

  public Kind(Kind voorbeeld) {
     this.gedrag = voorbeeld.gedrag;
  }
}
```

Codevoorbeeld 1

```
package familie;

public class Kind implements Cloneable {
  private String gedrag = "voorbeeldig";

  public Object clone() {
     Kind copy = new Kind();
     copy.gedrag = this.gedrag;
     return copy;
  }
}
```

Codevoorbeeld 2

instanties die deze ook werkelijk implementeren. Verder heeft dit het voordeel dat je zelf bepaalt wat het return type moet zijn en je casting kunt voorkomen:

- Wanneer kun je het beste het Prototype pattern gebruiken? Een eerste voorwaarde is dat het voor het systeem niet uitmaakt hoe objecten geïnstantieerd worden. Verder zijn gunstige voorwaarden:
- Het dynamisch laden van classes. Niet alleen in de zin van het op runtime toevoegen van nieuwe classes, maar ook bij het opzetten van een framework waar per definitie niet alle classes al bekend zijn.
- Als andere oplossingen (zoals met factory's) complexe class hiërarchieën opleveren, waarbij ook nog eens de hiërarchie van de factory's synchroon loopt met de objecten die gemaakt worden.

```
package familie;

public interface Prototype {
  public Kind maakKopie();
}
```

Codevoorbeeld 3

- Als het type waar je een prototype wilt maken slechts een beperkt aantal staten kent waar het zich in kan bevinden.
- Als dan het voorvermelde type ook nog complex is te

```
package familie;

public class Kind {
  private String gedrag = "voorbeeldig";

  public Kind maakKopie() {
     Kind copy = new Kind();
     copy.gedrag = this.gedrag;
     return copy;
  }
}
```

Codevoorbeeld 4

creëren is het gebruik van een Builder pattern in combinatie met een prototype aan te raden.

Dynamisch Het mooie aan dit pattern is dat het de mogelijkheid biedt tot het eenvoudig toevoegen van nieuwe typen in de programmatuur op een dynamische wijze. Voeg aan de implementatie een prototype manager toe. Deze prototype manager is waar de client nieuwe prototypen toe kan voegen en de rest van de applicatie kan via een sleutel een copy van het juiste prototype ontvangen. Hiermee is het probleem opgelost dat je niet weet welke constructor je aan moet roepen. Voor de prototypen hoef je geen constructor te gebruiken, de prototype manager regelt dat.

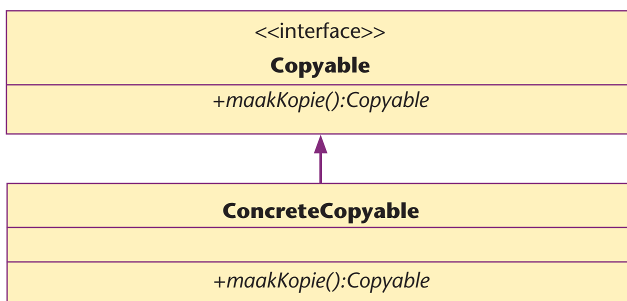
Overwegingen Let bij het maken van een copy goed op of je een deep- of een shallow-copy wil. Bij een deep-copy wordt van elke variabele een kopie gemaakt. De kopie heeft dan wel dezelfde data, maar het verwijst niet naar dezelfde objecten. Terwijl als je een shallow-copy maakt, het kopie-object dezelfde data heeft en in het geval van object-referenties naar dezelfde objecten verwijst.

Dit is een belangrijk verschil. Bij een shallow-copy kunnen bewerkingen op de kopie dus ook effect hebben op het prototype en vice versa.

Voordelen Prototype

- Er zijn minder subclasses nodig
- Programma is dynamisch configureerbaar
- Eenvoudiger creatie van instanties

Figuur 1. Prototype



Toepassen en herkennen Prototype Voor het gebruik van het Prototype pattern

- Copyable – de interface
- ConcreteCopyable – een implementatie van de Copyable interface, waarbij de copy methode een instantie van deze class teruggeeft.
- Client – het object dat geïnteresseerd is in een kopie van het origineel.

Relaties Wat is dan de relatie tussen het Prototype pattern en de Abstract Factory (besproken in deel 5)? Deze twee patterns zijn goed te combineren. De Abstract Factory kan een verzameling prototypen in zich huisvesten om deze op de juiste ogenblikken te gebruiken voor het fabriceren van een object. Andersom is ook mogelijk. Dat je op basis van een prototype factory een concrete implementatie krijgt van de Abstract Factory.

> Let bij het maken van een copy goed op of je een deep- of een shallow-copy wilt

Loop eens door je code en kijk of je een abstract factory hebt gebruikt die gemakkelijker was geweest met een prototype.

Tenslotte In de volgende aflevering van deze reeks nemen we de patterns Strategy en Flyweight onder de loep.

Literatuur

1) Design Patterns, Erich Gamma, et.al., Addison Wesley, 1995
2) Applied Java Patterns, Stephen Stelting & Olav Maassen, Prentice Hall, 2002

Olav Maassen (o.maassen@qnh.com) is senior software ontwikkelaar bij QNH/Delion B.V. te Gorinchem en co-auteur van 'Applied Java Patterns' met Stephen Stelting(2).