thema

Design patterns zijn de laatste paar jaren steeds populairder geworden. Je kunt geen boek over programmeren openslaan of men maakt gebruik van design patterns. Het lijkt alsof geen boek compleet is zonder iets over patterns. Vaak blijft het gebruik van patterns dan beperkt tot de patterns uit het boek van de Gang of Four[1], maar er zijn nog veel meer goede boeken over patterns. In deze aflevering wordt het pattern Half Object plus Protocol (HOPP) besproken. Dit pattern is afkomstig van Gerard Meszaros en gepubliceerd in Pattern Languages of Programming Design[2].

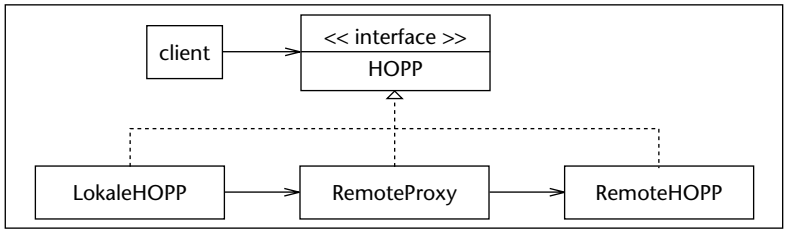
Het zijn maar patronen (3)

Half Object plus Protocol (HOPP)

"Dat heb ik vrij eenvoudig opgelost. Voor de communicatie met de server gebruikt de client een Proxy. Hij krijgt die proxy van de ProxyFactory die uiteraard een Singleton en een Abstract Factory is." Deze zin lazen we al in het eerste artikel van deze reeks (Java Magazine, april 2003). Toen werd ook de naam HOPP al aangestipt: "Is het niet beter om een HOPP in plaats van een Proxy te gebruiken?" Die specifieke vraag zal ik hier niet beantwoorden, omdat de keuze sterk afhankelijk is van de gekozen architectuur. En om die afweging te kunnen maken is het noodzakelijk om eerst, naast de Proxy[3], ook de HOPP te leren kennen.

De HOPP representeert een object dat in meerdere address spaces tegelijk code uitvoert

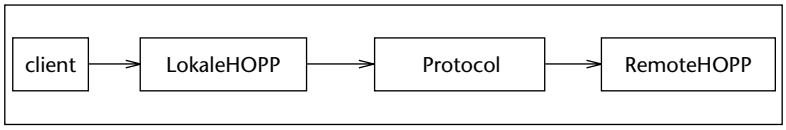
DYNAMISCH DISTRIBUEREN Java applicaties zijn steeds vaker gedistribueerd over verschillende address spaces. Een address space is in dit geval een Java Virtual Machine. Bij een gedistribueerde applicatie draait de applicatie op verschillende fysieke machines en dus in verschillende address spaces. Voor het goed verlopen van de applicatie is het nodig dat de verschillende virtual machines met elkaar kunnen communiceren. Het kan voorkomen dat een object op meerdere machines moet draaien. Als een object gebruik maakt van bepaalde fysieke apparaten die aan verschillende machines vastzitten, is het lastig om het object in logische objecten op te splitsen. Voor de communicatie tussen address spaces biedt de Java programmeertaal onder andere Remote Method Invocation (RMI). RMI stelt je in staat om zowel data als gedrag dynamisch te distribueren. Op een later tijdstip zullen we verder in gaan op de mogelijkheden die RMI je biedt. Belangrijk nu is dat RMI een van de mogelijkheden is voor gedistribueerde communicatie.



FIGUUR 1. UML diagram HOPP

SYNCHRONISEREN Bij het gebruik van de Remote Proxy pattern[3] is er een client en een remote object. RMI werkt op deze wijze. De client maakt gebruik van een stub die vervolgens elke methode aanroep doorstuurt naar het remote object. Het remote object handelt de methode af en geeft ofwel een returnwaarde ofwel een exception object terug. Het nadeel van deze remote proxy is dat elke methodeaanroep over het netwerk gestuurd wordt terwijl dat helemaal niet nodig hoeft te zijn. Sommige methoden zouden lokaal afgehandeld kunnen worden. Dit is waar de HOPP in het spel komt. De HOPP representeert een object dat in meerdere address spaces tegelijk code uitvoert. Het object dat in meerdere address spaces moet bestaan wordt in twee helften verdeeld en vervolgens voorzien van een protocol: twee helften en een protocol ertussen. Elke helft

is verantwoordelijk voor de samenwerking met andere objecten in zijn address space. Het protocol zorgt voor het synchroniseren van de gegevens van beide helften met elkaar en regelt de transport van dergelijke gegevens. De twee helften en het protocol vormen een drie-eenheid.



FIGUUR 2. Class diagram van HOPP met RemoteProxy

VARIANTEN Er zijn drie variaties op dit pattern:
- Gelijkwaardige helften
- Smart HOPP
- Asymmetrische HOPP

In de basis variant is het de client die een referentie heeft naar het remote gedeelte. Het is ook mogelijk om ook het remote gedeelte een referentie te geven naar de client. Het voordeel is dat de communicatie nu begonnen kan worden door het remote gedeelte. In wezen is hij dan de client voor het andere gedeelte. Beide kanten zijn qua communicatie gelijkwaardig. Deze variant werkt goed bij asynchrone communicatie tussen de twee helften.

Bij de tweede variant bevat de lokale implementatie de mogelijkheid om te kiezen wat hij als remote gedeelte wil gebruiken. Het grote voordeel is dat de client code niet hoeft te wijzigen, maar dat er wel andere functionaliteit of dezelfde functionaliteit in een flexibele address space aan kan roepen. Het spreekt voor zich dat deze variant het best toepasbaar is in een dynamische omge-

```
1: HOPP.java
2: LokaleHOPP.java
3: RemoteHOPP.java
4: Client.java
```

```
package hopp;

import java.rmi.RemoteException;

public interface HOPP {
   public void lokaleMethode() throws
   RemoteException;
   public void remoteMethode() throws
   RemoteException;
}
```

```
package hopp;

import java.rmi.RemoteException;

public class LokaleHOPP implements HOPP {
   private HOPP remote;

         public LokaleHOPP() {
            super();
      // Initieert de variabele remote met de
      waarde die
      // hij vindt in de rmiregistry.
         }

         public void lokaleMethode() {
      System.out.println("Dit wordt in de ene
      address space uitgevoerd.");
         }

         public void remoteMethode() throws
         RemoteException {
      remote.remoteMethode();
         }
}
```

```
package hopp;

import java.rmi.RemoteException;

public class RemoteHOPP implements HOPP {
         public RemoteHOPP() {
            super();
      // Registreert zichzelf bij de rmiregist-
      ry
         }

         public void lokaleMethode() {
      // Wordt niet aangeroepen.
   }

         public void remoteMethode() {
      System.out.println("Dit wordt uitgevoerd
      in de andere address space.");
         }
}
```

```
package hopp;

import java.rmi.RemoteException;

public class Client {
   public static void main(String[] args) {
      try {
         HOPP eenHOPP = new LokaleHOPP();
         // voor de client is het transparant
         waar de volgende
         // twee methoden uitgevoerd worden
         eenHOPP.lokaleMethode();
         eenHOPP.remoteMethode();
      } catch (RemoteException re) {
         re.printStackTrace();
      }
   }
}
```

ving waar machines komen en gaan. Het is niet per se noodzakelijk dat beide helften dezelfde interface implementeren. Dit is te zien in de Asymmetrische HOPP. De taken en verantwoordelijkheden van de beide helften zijn duidelijk gescheiden. Hierdoor vervalt de noodzaak om exact dezelfde interface te hebben. Om goed te functioneren hebben ze elkaar echter nog wel nodig.

ALTERNATIEF Een alternatief om hetzelfde te bereiken (de code gedeeltelijk lokaal, en gedeeltelijk remote uitvoeren) kan met RMI. Bij het genereren van de subcode voor de clientkant kun je de optie -keep aan de rmi compiler (rmic) meegeven. De Java sources van de stub blijven nu bewaard. Vervolgens kun je de sources aanpassen, zodat de methoden die je lokaal wilt hebben ook inderdaad lokaal uitgevoerd worden en de sources compiler met de Java compiler. Het voordeel van deze methode is dat je minder code nodig hebt. Je hebt geen client HOPP implementatie nodig. Dit weegt echter niet op tegen de nadelen.

Een nadeel van dit alternatief is dat je de stub-code elke keer opnieuw moet aanpassen als je de rmi-code genereert. Verder is het veranderen van het communicatie protocol tussen de twee helften veel omslachtiger. Als een ander protocol gekozen moet worden (bijvoorbeeld xml over http voor de gegevensuitwisseling) breekt dit je code en moet je de hele constructie opnieuw schrijven. In de HOPP zit het protocol in een protocolobject en hoeft alleen de code in het protocolobject aangepast te worden.

VOOR- EN NADELEN Het verschil tussen de twee address spaces wordt transparant voor de client zonder dat dit veel overhead oplevert. De transparantie kan zover doorgevoerd worden dat het voor de client niet duidelijk is dat hij met een andere address space communiceert. Bovendien biedt dit je het voordeel dat de communicatie geoptimaliseerd kan worden. Niet elke aanroep van een methode resulteert in een aanroep van een remote methode. De lokale helft kan bepalen wanneer en hoe hij de gegevens ophaalt uit zijn andere helft. Dit kan het netwerkverkeer aardig beperken. Daarnaast kan het communicatieprotocol netjes gescheiden worden van de rest van de logica. Mocht deze gewijzigd moeten worden is de impact op de rest van de code beperkt. Nadeel van dit pattern is dat sommige functionaliteit gedupliceerd zal zijn in beide helften. Er is immers voldoende informatie nodig om te kunnen omgaan met lokale objecten.

HERKENNEN EN TOEPASSEN Voor het gebruiken van het HOPP pattern zijn de volgende onderdelen nodig (zie figuur 1):

- LokaleHOPP - Sommige van de methoden worden lokaal uitgevoerd, anderen worden doorgestuurd naar de RemoteHOPP door middel van het protocol. LokaleHOPP kan tevens verschillende communicatie strategieën bevatten om te kunnen kiezen wat en wanneer naar de RemoteHOPP moet.
- Protocol - Deze class bevat het communicatie protocol tussen de client en de remote HOPP.
- RemoteHOPP - De helft van de HOPP die de code remote (in een andere address space) uitvoert.

De HOPP vertoont enige gelijkenissen met de RemoteProxy. In figuur 2 zie je de HOPP geïmplementeerd met de RemoteProxy als protocol tussen de twee helften van de HOPP.

- HOPP - De interface die de beschikbare methoden definieert. Beide helften van de HOPP implementeren deze interface. Eventueel kan zelfs het protocol de HOPP-interface implementeren (bijvoorbeeld als RMI als protocol gebruikt wordt).

TENSLOTTE Volgende keer keren we terug naar de patterns van de GoF en zijn de twee fabrieken aan de beurt: Factory Method en Abstract Factory.

Voorbeeldcode bij dit artikel is te downloaden via de bijbehorende website van Itis J-solutions:
www.itis.nl/patterns

> De twee helften en het protocol vormen een drie-eenheid

Literatuur

1 Design Patterns, Erich Gamma, et. al., Addison-Wesley, 1995
2 Pattern Languages of Program Design, Jim O. Coplien, Douglas C. Schmidt, Addison-Wesley, 1995
3 Het zijn maar patronen (2), Olav Maassen, in Java Magazine jrg. 2 nr. 2, juni 2003
4 Applied Java Patterns, Stephen Stelting & Olav Maassen, Prentice Hall, 2002

Olav Maassen (olav.maassen@itis.nl) is senior Java developer bij ITIS J-solutions B.V. te Maarsbergen. Samen met Stephen Stelting is hij co-auteur van het boek "Applied Java Patterns".