"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." De tekst hierboven zal afhankelijk van het kennisniveau van design patterns andere vragen oproepen. Voor de mensen die de patterns niet kennen zal het bovenstaande weinig betekenen en zal het systeem uitgebreid uitgelegd moeten gaan worden. De kenner echter krijgt via deze twee zinnen een heleboel informatie over het ontwerp en zal gelijk mee kunnen gaan denken: "Is het niet beter om een HOPP in plaats van een Proxy te gebruiken?"

Het zijn maar patronen

Design patterns en Java

De laatste jaren zijn patterns, met name design patterns, sterk in populariteit gegroeid. In artikelen en boeken wordt verwezen naar design patterns en wordt er vanuit gegaan dat iedereen volledig vertrouwd is met de concepten hierachter. In de praktijk valt de hoeveelheid kennis van patterns nog wel eens tegen. Design patterns zijn omgeven met een soort magisch veld. Het is een manier om je te onderscheiden van je collega ontwikkelaars (zie ook het kader 'Design Patterns' voor een korte uitleg). Design patterns worden echter ook te vaak gebruikt om rookgordijnen op te werpen. "Laten we maar wat design patterns opnoemen dan lijkt het wat", is dan het motto. In deze reeks artikelen zullen design patterns behandeld worden aan de hand van voorbeelden uit het dagelijkse 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.

SINGLETON November en begin december zijn in Nederland altijd een spannende periode voor de kinderen onder de zeven jaar oud. Het is namelijk de periode waarin de goedheiligman ons land bezoekt en cadeaus uitdeelt ter ere van zijn verjaardag. Kinderen zingen liederen om de Sint te bekoren en leveren verlanglijstjes in in de hoop dat vertaald te zien in cadeaus.

Wat heeft dit te maken met design patterns en Java, zult u zich afvragen. Meer dan u denkt: het levert een situatie die in software ook geregeld voorkomt. De eigenschappen van een Sinterklaas staan beschreven in een Sinterklaas class. Er bestaat echter maar één Sinterklaas. Daarom is er een constructie nodig die ervoor kan zorgen dat er te allen tijde slechts één instantie van deze Sinterclass kan zijn en dat het creëren van meer instanties niet mogelijk is. Anders kan iedereen zijn eigen Sinterklaas maken.

UITDAGINGEN Een eerste uitdaging vormt de constructor. Als er in de class geen constructor staat, levert het systeem een default constructor. En iedereen kan dan een Sinterklaas maken. In de class moet dus een constructor staan die niet toegankelijk (private) is. Een member of constructor die private is kan alleen vanuit de eigen class aangeroepen worden.

> In plaats van steeds dezelfde referentie terug te geven, kan de applicatie een object uit een pool kiezen

De volgende uitdaging is: hoe kom je dan aan een referentie naar een Sinterklaas instantie? Op dezelfde wijze als de kinderen. Zij zingen bij de schoorsteen, radiator of achterdeur in de hoop dat zij gehoord worden. Het enige wat zij weten is dat er een type Sinterklaas bestaat; wie dat is en hoe zij hem kunnen bereiken weten ze niet. In een Java-programma kun je dit bereiken door het gebruik van static methoden. In plaats van een instantie aan te maken via een construc-

```
import sint.Kado;

public class Sinterklaas {
  private static Sinterklaas deEnige = new
Sinterklaas();

  private Sinterklaas() {}

  public static Sinterklaas zingLied() {
    return deEnige;
  }

  public Kado[] vraagKados(String[] verlang-
lijst) {
    //op basis van de verlanglijst wordt een
keuze
    //gemaakt. In deze implementatie wordt er
geen
    //kado teruggegeven.
    return new Kado[0];
  }
}
```

CODE 1: Java uitwerking van een Singleton

tor kan dit ook via een static methode. Dit principe, een static methode in plaats van een constructor, is ook een pattern op zich: de Factory Method. Over dat pattern in een later artikel meer.

In de Sinterklaas class staat een static methode: zingLied(). Wat een kind daarvan verwacht is dat hij een referentie krijgt naar Sinterklaas, ofwel: het returntype van de methode moet Sinterklaas zijn, de naam van de class waar de methode in staat. Om er nu voor te zorgen dat er ook werkelijk maar een instantie is en blijft, moet de class elke keer dezelfde referentie terug geven. In de Sinterklaas class nemen we dan een attribute van type Sinterklaas op. Deze hoeft slechts een keer gevuld te worden en moet continu hergebruikt worden (zie code 1).

Nadat via de static methode een referentie naar Sinterklaas is verkregen, kan deze behandeld worden als elk ander object. Naast allerlei andere methoden zal Sinterklaas bijvoorbeeld de 'normale' methode vraagKados hebben. Deze methode wordt nu dus altijd op de enige echte instantie aangeroepen. De hier voorgaande constructie heet in design patterns een Singleton. De Singleton wordt veel in de Java API's toegepast. Twee voorbeelden hiervan zijn javax.swing.ToolTipManager en java.rmi.server.RMISocketFactory.

Voordelen van een Singleton

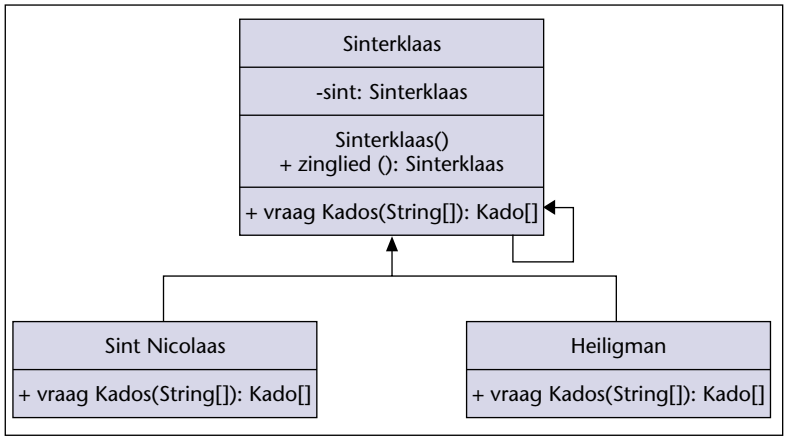
- Alleen de class zelf kan een instantie van de class maken. De buitenwereld (alles buiten de class) moet altijd de static methoden gebruiken om een referentie naar een instantie te kunnen krijgen
- De referentie naar een Singleton object hoeft niet doorgegeven te worden aangezien iedereen via de static methode toegang heeft tot hetzelfde object.

VARIATIES Naarmate de kinderen langer op school verblijven, zullen zij meer van rekenen gaan begrijpen. Dit heeft ook consequenties voor hun beleving van het Sinterklaasfeest. Kijkende naar het journaal horen zij dat er meer dan twee miljoen gezinnen Sinterklaas vieren. Het zal enige tijd duren, maar zij zullen begrijpen dat het voor die ene instantie heel moeilijk wordt iedereen van de juiste cadeaus te voorzien en overal te verschijnen. Gelukkig heeft de Sint in die drukke dagen assistentie van een aantal hulpsinterklazen. Zij reizen net als de Sint door het hele land en doen alles wat de echte Sint ook zou doen: cadeaus uitdelen, naar liedjes luisteren, enzovoort.

De situatie lijkt veel op de Singleton. De externe werking en interface blijft hetzelfde. Voor de buitenwereld lijkt het nog steeds alsof er maar één Sinterklaas instan-

```
import sint.Kado;

public class Sinterklaas {
  private static Sinterklaas[] hulpSint = {
    new Sinterklaas(),
    new Sinterklaas(),
    new Sinterklaas(),
    new Sinterklaas()
  };

  private static int index=-1;

  private Sinterklaas() {}

  public static Sinterklaas zingLied() {
    index = (index == hulpSint.length-1) ? 0
: index+1;
    return hulpSint[index];
  }

  public Kado[] vraagKados(String[] verlang-
lijst) {
    //op basis van de verlanglijst wordt een
keuze
    //gemaakt. In deze implementatie wordt er
geen
    //kado teruggegeven.
    return new Kado[0];
  }
}
```

CODE 2: Sinterklaas met hulp van de hulpsinterklazen



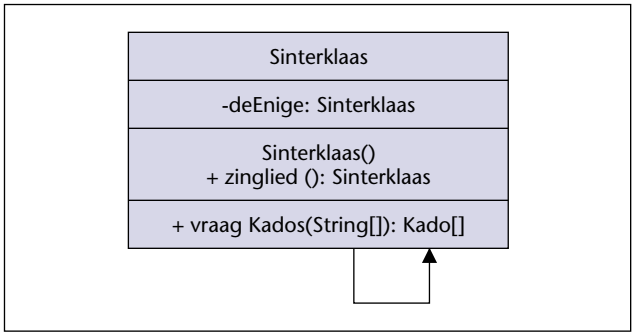
UML class diagram 1

tie is. De interne werking is anders dan voorheen. In plaats van steeds dezelfde referentie terug te geven, kan de applicatie een object uit een pool kiezen om op die manier het werk te verdelen over de verschillende instanties. In het tweede codevoorbeeld wordt dit idee verder uitgewerkt, waarbij de verschillende instanties gerouleerd worden (zie code 2).

Een tweede variant op de Singleton maakt gebruik van subtypes om terug te geven. Dit heeft wat implicaties voor de Java code. Aangezien we subclasses van de Singleton gaan maken, moeten de subtypes gebruik kunnen maken van een constructor in hun superclass. Zo kan de constructor niet meer private zijn, maar zal deze protected moeten worden.

> Elke keer dat er een nieuw subtype geïntroduceerd wordt, moet de Singleton aangepast worden

In de static toegangsmethode (heet ook wel accessor method) zal de Singleton bepalen welke instantie hij terug wil gaan geven. Dat zou uiteraard een instantie van de Singleton zelf kunnen zijn, maar waarschijnlijker is het een subtype. Zie hiervoor ook uml class diagram 1.



UML class diagram 2

```
public static Sinterklaas zingLied() {
  Sinterklaas returnValue = null;
  String sint =  System.getProperty("sinter-
klaas");
  if (sint != null) {
    try {
      Class sintClass = Class.forName(sint);
      returnValue =
(Sinterklaas)sintClass.newInstance();
    } catch (Exception e) {
      // we doen niks met de eventuele
excepties
    }
  }
  return (returnValue != null) ? returnValue
: deEnige;
}
```

CODE 3

Het gevaar bij deze benadering is dat de Singleton als superclass op de hoogte moet zijn van alle subtypes. Elke keer dat er een nieuw subtype geïntroduceerd wordt, moet de Singleton aangepast worden. Dit is natuurlijk verre van wenselijk, omdat dit fouten in de hand gaat werken en je onderhoudbaarheid van de code verlaagd. Om dit probleem te omzeilen kan de Singleton gebruik maken van 'dynamic class loading'. De static toegangsmethode maakt de instance van de subclass door een property uit te lezen. Deze property bevat als waarde de naam van de subclass, zoals ook te zien is in het volgende codefragment (zie code 3).

Door het zetten van de systeem property kan het gedrag van de Singleton gewijzigd worden zonder dat wij de code zelf hoeven te wijzigen. De code lijkt wat gekunsteld, maar biedt de programmeur veel flexibiliteit en wordt ook werkelijk in de Java API's gebruikt. Classes die de Singleton met deze variant hebben toegepast zijn bijvoorbeeld java.awt.Toolkit, java.sql.ResourceBundle en java.rmi.server.RMISocketFactory.

ALTERNATIEF Uiteraard hoef je niet steeds bij dit probleem een Singleton pattern toe te passen. Een alternatief zou kunnen zijn om alle methoden en attributen static te maken om er op die manier voor te zorgen dat er maar een instantie is. Deze strategie zien we ook in de Java API's voorkomen, zoals bijvoorbeeld in java.sql.DriverManager.

Het voordeel van de static variant is dat alle methoden direct toegankelijk zijn zonder eerst over de instan-

Design Patterns

Waar komen de design patterns nu eigenlijk vandaan? Het meest invloedrijke boek op het gebied van objectoriëntatie en design patterns is zonder twijfel 'Design Patterns' van Erich Gamma, Richard Helm, Ralph Johnson en John Vlissides (1). De auteurs worden samen ook de 'Gang of Four' genoemd, vaak afgekort tot GoF.

Het GoF-boek beschrijft 23 vaak voorkomende problemen en de bijbehorende oplossingen in algemene termen. Deze 23 patterns zijn de meest bekende patterns en vormen de basis voor deze artikelreeks.

De GoF was op haar beurt weer geïnspireerd door het boek van Christopher Alexander (2). Hoewel zijn boek over huis architectuur gaat, beschrijft het ideeën vanuit een pattern benadering. Eerst wordt het probleem geïntroduceerd, vervolgens een beschrijving in algemene termen van de oplossing en vervolgens een mogelijke implementatie van de oplossing. Belangrijk hierbij zijn ook de voor- en nadelen van een bepaald pattern en de krachten die een rol spelen bij het pattern.

Het GoF-boek maakt gebruik van C++ en Smalltalk voor haar code voorbeelden. Er zijn tegenwoordig een heleboel boeken beschikbaar die design patterns vanuit Java verklaren.

tie reference te beschikken. Gewoon de methode op de class aanroepen is voldoende. De constructor is in de meeste gevallen ook private aangezien het maken van een instantie geen zin heeft met louter alleen static methoden in de class.

Het nadeel van deze strategie is dat de methoden al meteen beschikbaar zijn op het moment dat de class geladen wordt. De class kan slechts met veel kunst en vliegwerk initialisatie afdwingen voordat bepaalde methoden aangeroepen worden. De methode controleert eerst of er een initialisatie gebeurd is en zo niet, dan gooit de methode een exceptie of voert de standaardinitialisatie uit.

De natuurlijke oplossing hiervoor zou dan toch de Singleton zijn. Via de toegangsmethode kan de initialisatie afgedwongen worden en - belangrijker nog - beïnvloed worden door de aanroeper. De instantie is dus altijd beschikbaar, maar wordt pas geïnitialiseerd als de gebruiker van de class dat wil, op de manier die hij wil.

Voordelen Patterns

- bewezen oplossing voor een algemeen probleem
- snellere communicatie
- delen van ervaring
- complexe problemen worden eenvoudiger

Toepassen en herkennen Voor het bereiden van een Singleton zijn de volgende ingrediënten nodig (zie tevens UML class diagram 2):

- private constructor zodat alleen vanuit de class zelf een instance gemaakt kan worden
- private static instance (1 of meerdere) van het type van de Singleton zodat altijd hetzelfde object gebruikt kan worden door iedereen
- public static toegangsmethode om iedereen toegang te geven tot die ene instantie (heet ook wel static factory method; een ander design pattern, waarover in een later artikel meer).

Tenslotte De Singleton is het eerste pattern welke besproken is in deze reeks over design patterns en Java. De volgende keer gaat over de body double oftewel de Proxy.

Literatuur

1 Design Patterns, Erich Gamma, et. al., Addison-Wesley, 1995

2 Pattern Language, Christopher Alexander, et.al., Oxford University Press, 1977

3 Applied Java Patterns, Stephen Stelting & Olav Maassen, Prentice Hall, 2002

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

Olav Maassen is senior Java developer bij ITIS J-solutions B.V. te Maarsbergen (e-mail: olav.maassen@itis.nl)