De Vergeten Abstracties

Cesario Ramos
Senior Consultant bij Xebia B.V.
2009

Inleiding

Rollen zijn een belangrijk concept in object georiënteerde software ontwikkeling dat vaak vergeten wordt. Het gebruik van rollen verbetert het hergebruik van functionaliteit en maakt het mogelijk een strikte scheiding van verantwoordelijkheden te realiseren. Dit komt omdat je context specifieke functionaliteit in een rol plaatst en niet in de class zelf. Op deze manier kun je objecten verschillende rollen laten spelen en daarmee het gedrag van objecten tijdelijk uitbreiden zonder dat je daarvoor de class van het object hoeft te wijzigen.

In dit artikel wordt besproken wat rollen zijn, waarom we ze zouden willen gebruiken en hoe je rollen met ObjectTeams in Java zou kunnen gebruiken.

Objecten, Rollen en Collaboraties

Een rol is een verzameling attributen en gedrag dat een object kan aannemen in een bepaalde context. Denk bijvoorbeeld aan een film script waarin acteurs de rollen spelen beschreven in het script. Een acteur gedraagt zich volgens het script zolang de rol gespeeld wordt. Daarna is de acteur weer gewoon zichzelf.
Rollen hebben dan ook andere eigenschappen dan classes. Rollen kunnen worden gespeeld door objecten van verschillende classes en een object kan verschillende rollen spelen. Dit kan allemaal zonder dat de class van een object aangepast hoeft te worden. Een rol wordt dynamisch toegewezen aan een object en is van tijdelijke aard.

In een collaboratie zitten verschillende rollen die in samenwerking met elkaar een stuk functionaliteit realiseren. De collaboratie beschrijft hoe de rollen met elkaar samenwerken. Een Race biedt bijvoorbeeld de rol RaceAuto aan voor een Auto die aan de Race wil deelnemen. Een raceauto in een race heeft weer te maken met een coureur, een circuit en bijvoorbeeld ook het weer. Deze interactie tussen de verschillende rollen is beschreven in de collaboratie Race.
Een auto object dat de rol van RaceAuto krijgt toegewezen krijgt additionele eigenschappen en gedrag. Denk bijvoorbeeld aan het aantal laps dat het nog te gaan heeft. Al deze eigenschappen en interacties bestaan alleen binnen de context van een Race. De collaboratie zelf kan ook gedrag en state hebben. Zo weet de race wie de deelnemers zijn en in welke lap de race zit.

Welke problemen kun je ermee oplossen?

Polymorfisme zorgt er voor dat je objecten kunt aanspreken op een abstract niveau. Zo kun je bijvoorbeeld abstraheren van een collaboratie waarin de samenwerking tussen objecten is vastgelegd. Denk bijvoorbeeld aan het SubjectObserver **[1]** pattern waarin de collaboratie logica in de Subject en Observer geplaatst zijn. Zie hieronder in het kort een Java implementatie.

```
public abstract class Subject {
	protected Set<Observer> observers = …
	public void notify() {
		for(Observer aObserver: observers) {
			aObserver.update(this);
		}
	}

	public void attachObserver(Observer anObserver) {…}
…
}

public interface Observer {
	public void update(Subject theSubject);
}
```

Stel dat we het SubjectObserver pattern gebruiken zodat we aanpassingen aan producten ‘soft realtime’ op een scherm kunnen tonen.

Stel dat TV een product is en de rol van Subject speelt en dat Scherm de rol van Observer speelt. We realiseren dit door de TV class te laten erven van abstract class Subject en door Scherm het interface Observer te laten implementeren. Het gevolg is dat de class TV een Subject is geworden. Scherm is een Observer doordat de **update** methode geïmplementeerd wordt. De mengeling van Subject en Observer verantwoordelijkheden in TV en Scherm betekent dat je geen goede Seperation Of Concerns **[2]** hebt. De class TV hoort zich druk te maken om bijvoorbeeld volume, kleurinstellingen en te tonen zender. Nu dat het ook een Subject is moet het hiernaast ook kennis over het updaten van observers hebben. Deze verantwoordelijkheid hoort niet bij een TV en is ook strijdig met het Single Responsibility Principle **[3]**. Doordat je inheritance gebruikt is het ook zo dat alle objecten die je instantieert op basis van deze TV class een Subject zijn!
Stel dat er ook een rekenkundige module gemaakt moet worden. Deze module wordt gebruikt voor het uitbrengen van offertes en berekenen van kortingen etc. Deze module heeft niks met SubjectObserver functionaliteit te maken, toch zijn de producten zoals TV een Subject. Objecten in de offerte module hebben echter ander gedrag en state nodig specifiek voor de rekenkundige module. Het is nog maar de vraag of dit specifieke gedrag wel bij de TV class hoort! Waarschijnlijk niet. Je kunt je voorstellen dat naarmate er meer modules komen die TV in andere specifieke situaties gebruiken, de TV class verschillende clients moet ondersteunen. Hierdoor

loop je het risico dat de TV class al gauw complex en onoverzichtelijk wordt. Je kunt snel in complexe klasse structuren verzanden met als gevolg complexe code.
Je zou kunnen denken dat je met een sub-class specifiek voor de SubjectObserver functionaliteit en een sub-class specifiek voor de rekenkundige module een goede oplossing hebt, Dit is maar gedeeltelijk waar. Het probleem is namelijk dat een TV object dan niet zowel als Subject als in de rekenkundige module kunt gebruiken. Je hebt immers twee objecten, een object van elk van de sub-classes nodig. Dit kan object identiteit problemen geven aangezien je met twee verschillende TV objecten te maken hebt.

De kern van het probleem zit in de verschillende rollen dat een object speelt. Je ziet dat een TV object een bepaalde rol speelt afhankelijk van wat het **doet** niet van wat het **is**. Je ziet dat met alleen interfaces en abstracte classes je moeite hebt om object afhankelijke en object onafhankelijk zaken te scheiden. Een TV object is immers afhankelijk van de context waarin het gebruikt wordt dan wel een grafisch object of een offerte object. Het zou dan ook een grote verbetering zijn als een object afhankelijk van zijn rol, dynamisch gedrag en state krijgt die alleen relevant is in een bepaalde context. Iets van 'object inheritance' zou leuk zijn zonder dat de type hiërarchie hiervoor aangepast hoeft te worden.

Rollen binnen Java met Object Teams

Object Teams/Java (OT/J) **[4]** is een uitbreiding op Java. Het introduceert een aantal taal constructies die het programmeren met rollen en collaboraties mogelijk maakt. Een collaboratie in ObjectTeams wordt voorgesteld door een **team.** Binnen een team kun je rollen definiëren en hun onderlinge interactie. Het koppelen van een rol aan een klasse gebeurt met het **playedBy** keyword. Hierdoor kun je run-time een object uitbreiden met rol specifiek gedrag en kan het object de rol spelen in de collaboratie

Het ObjectTeams project biedt een tweetal mogelijkheden om een applicatie te ontwikkelen. Naast een command line compiler en tooling, biedt het ook zeer goede integratie aan met Eclipse. Het levert middels de Object Teams Development Tooling (OTDT) een uitbreiding op Eclipse waarin je volledige ondersteuning krijgt bij het programmeren. Om programma's geschreven in OT/J te kunnen executeren heb je de Object Teams Runtime Environment (OTRE) nodig. De OTRE zorgt ervoor dat met bytecode instrumentatie de rol functionaliteit gekoppeld wordt aan de base classes om vervolgens standaard .class files te krijgen. Je kunt de OTRE bijvoorbeeld inschakelen door de **-javaagent** optie te gebruiken bij het starten van de JVM.

SubjectObserver in ObjectTeams
Zie hieronder het SubjectObserver voorbeeld gemaakt in ObjectTeams.

```
public abstract team class SubjectObserver {
	…
	protected abstract class Subject {
		private LinkedList<Observer> observers = …
```

```
            public void attachObserver (Observer o) {…}
            ...
            public void notify() {
                  for (Observer observer : observers)
                        observer.update(this);
            }
      }

      protected abstract class Observer {
            abstract void update(Subject s);
      }
}
```

Het **team** SubjectObserver stelt de context ofwel collaboratie voor. In deze collaboratie zijn de twee rollen Subject en Observer gedefinieerd met rol specifiek gedrag.
Je kunt nu bijvoorbeeld een TV object de rol van Subject laten spelen en Screen de rol van Observer laten spelen. Hieronder zie je hoe dat zou kunnen.

```
public team class ObservingGraphical extends SubjectObserver {
      protected class Subject playedBy TV {
            notify <- after setPrice;
      }

      protected class ScreenObserver extends Observer playedBy Screen {
            update -> redraw;
      }
      …
}
```

Je ziet het **team** ObservingGraphical dat erft van SubjectObserver. Aangezien dezelfde rolnaam voor Subject is gebruikt in zowel team SubjectObserver als ObservingGraphical ziet ObjectTeams dit als impliciete overerving van de rol. Verder wordt met het **playedBy** keyword aangegeven dat de rol Subject door objecten van TV worden gespeeld en dat de rol Observer door objecten van Screen worden gespeeld. (Je kunt b.v. ook aangeven welke objecten van TV of Scherm de rol mogen spelen. Deze mogelijkheid wordt echter niet in dit artikel besproken. Zie **[5]** voor verdere informatie hierover.) Om de koppeling tussen objecten en rollen te maken is gebruik gemaakt van **callin** en **callout** statements. Met het **callin** statement '`notify <- after setPrice;`' geef je aan dat nadat de **setPrice** method van TV is aangeroepen, method **notify** van de Subject aan moet worden geroepen. Met **callout** statement '`update -> redraw;`' wordt de **update** methode doorverwezen naar de **redraw** methode van Screen. Op deze manier kun je dus specifiek voor de objecten die een rol willen spelen de mapping realiseren tussen de rol en het object dat de rol speelt.

Hoe je de collaboraties en bijbehorende rollen kunt gebruiken is hieronder aangegeven.

```
final ObservingGraphical observingGrph = new ObservingGraphical();
TV tv = new TV();
Screen screen = new Screen();
within(observingGrph) {
      observingGrph.attachSubjectObserver(tv, screen);
      tv.setPrice(1000);
}
```

Hierboven zie je dat na het aanmaken van team ObservingGraphical, met het **within** statement de rollen worden geactiveerd. De TV en de Screen objecten hebben het rol specifieke gedrag binnen de scope van het **within** statement. Binnen de scope van het **within** statement zal de aanroep van methode **setPrice** van object tv tot gevolg hebben dat methode **notify** van Subject aangeroepen wordt, vervolgens wordt de methode update van Observer en tot slot methode **redraw** van het object screen aangeroepen.

In bovenstaande oplossing is een strikte Separation Of Concerns te zien. Zowel de Screen als de TV class hebben geen enkele notie of afhankelijkheid naar de SubjectObserver collaboratie. Naast een goede Separation of Concerns hebben we tevens een goede realisatie van het Open Closed Principle **[6].**

Hergebruik van collaboraties

IStel dat je een PrinterService **[7]** hebt die twee typen gebruikers aankan. Een normale gebruiker die een beperkt aantal pagina's mag printen en een premium gebruiker die ongelimiteerd kan printen. De normale gebruiker moet het aantal geprinte pagina's bijhouden en voor administratieve doeleinden moet de PrinterService zelf het totaal aan geprinte pagina's bijhouden. Je wilt de PrinterService als component aanbieden aan verschillende clients en de logica m.b.t. printen en print administratie herbruiken.

Zie hieronder de PrinterService in ObjectTeams.

```
public team class PrinterService {
      private int totalPrintedPages = 0;

      private void print(Job job, Login login) {
            …
      }

      private void setTotalPrintedPages(int totalPrintedPages) {
                  this.totalPrintedPages = totalPrintedPages;
      }
```

```
	…

	public abstract class User {
		private static final int MAX_PAGES_USER = 100;
		int counter = 0;
		public int print(Job job) {
			if (counter + job.getNumberPages() >= MAX_PAGES_USER)
					return 0;
			counter += job.getNumberPages();
			PrinterService.this.print(job, getLogin() );
			return counter;
		}

		public int getPrintedPages() {
			return counter;
		}

		protected abstract Login getLogin();

	}

	public abstract class SuperUser {
		public int print(Job job) {
			PrinterService.this.print(job, getLogin());
			return getTotalPrintedPages();
		}
		public int getTotalpages(){
			return getTotalPrintedPages();
		}

		protected abstract Login getLogin();
	}
}
```

We zien de collaboratie (**team**) PrinterService en de abstracte rollen van User en SuperUser. De User kan maximaal 100 pagina's printen en voor de SuperUser is geen limiet. Het bijhouden van het aantal geprinte pagina's middels de counter property is rol specifiek. Verder zie je dat de **print** methode van de PrinterService naast de te printen Job ook vraagt om een Login. Met deze login bepaalt de PrinterService of toegang tot de PrinterService verleend mag worden. De **getLogin** methode is abstract gedefinieerd en moet nog gemapped worden op het specifieke object.
Willen we gebruik maken van deze printer service dan kunnen we dat doen door aan te geven welke objecten de rol van User en SuperUser willen spelen. Zie hieronder een voorbeeld.

```
public team class PrinterForPersons extends PrinterService {

	public class User playedBy Student {

		Login getLogin() -> Login getLogin();
	}

	public class SuperUser playedBy Teacher {

		Login getLogin() -> Login getLogin();
	}

	public SuperUser asSuperUser(Teacher as SuperUser superUser) {
		return superUser;
	}

	public User asUser(Student as User user) {
		return user;
	}
}
```

We zien dat rol User door Student en rol SuperUser door Teacher gespeeld wordt. De collaboratie methode **getLogin** wordt voor beide rollen rechtstreeks doorverwezen naar de **getLogin** methoden van Student en Teacher.
Verder zien we methoden asSuperUser en asUser. Deze methoden zorgen ervoor dat je middels **lifting** toegang tot de rol dat een object speelt kunt verkrijgen. Dit is een techniek dat je niet vaak zult gebruiken, omdat je de rollen aleen binnen de collaboraties wilt gebruiken. Met behulp van deze lifting methoden zou je zoals hieronder aangegeven de rollen buiten de collaboratie kunnen gebruiken.

```
final PrinterForPersons printerForPersons = new PrinterForPersons();

// verkrijgen van domain objecten
Student person1 = …
Teacher person2 = …
// rol activeren
printerForPersons.activate();

// externalized roles middels lifting
User<@printerForPersons> user = printerForPersons.asUser(person1);
SuperUser<@printerForPersons> superUser =
printerForPersons.asSuperUser(person2);
```

```
// rol specifieke use case
superUser.print(new Job(25));
user.print(new Job(5));

superUser.getPrintedPages());
user.getPrintedPages());

printerForPersons.getTotalPrintedPages();
// rol deactiveren
printerForPersons.deactivate();
```

Om de rollen buiten de collaboratie te gebruiken heb je zogenaamde **externalized roles** nodig. Om een externalized role te declareren moet je aangeven bij welk team het hoort. Dit is nodig om statisch type controles te kunnen uitvoeren. In het geval je niet aangeeft bij welk team een externalized role hoort kun je run time problemen krijgen als je rollen uit verschillende teams door elkaar gaat gebruiken. Het statement '`User<@printerForPersons> user`' declareert de rol User als externalized en geeft aan dat de role bij het printerForPerson team hoort.
Het is dus mogelijk om het collaboratie specifieke gedrag ook buiten de collaboratie te halen waarmee je extra flexabiliteit krijgt. Vaak zal verstandig zijn om juist specifieke user story gedrag in te kapselen in een collaboratie. Dit gedrag kan dan worden hergebruikt door verschillende domain objecten de rollen te laten spelen.

Conclusie

Zoals uit bovenstaande voorbeelden blijkt maakt ObjectTeams het mogelijk om binnen Java met rollen te werken. Hierdoor krijg je binnen Java extra mogelijkheden tot modularisatie en object compositie die je in staat stellen om tot een systeem te komen met hoge herbruik- en uitbreidbaarheid.
Je kunt user story specifieke logica inkapselen in een collaboratie waarin je uitsluitend met rollen werkt. Afhankelijk van de situatie koppel je dan domain objecten aan de rollen. Op deze manier krijg je een scheiding tussen wat een object **doet** en wat het **is** waardoor je op een meer Agile manier kunt omgaan met veranderende requirements.

Referenties

[1] E. Gamma et al (1994). Design Patterns: Elements of Reusable Object-Oriented Software.
[2] Edsger W.Dijkstra (1974). HYPERLINK "http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html" http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html.
[3] Robert C. Martin (2002). Agile Software Development: Principles, Patterns, and Practices.
[4] ObjectTeams. HYPERLINK "http://www.objectteams.org" http://www.objectteams.org/

[5] ObjectTeams language definition. HYPERLINK "http://www.objectteams.org/def/1.2/index.html" http://www.objectteams.org/def/1.2/index.html.
[6] Bertrand Meyer (1997). Object-Oriented Software Construction, Second Edition.
[7] Matteo Baldoni et el. The Interplay between Relationships, Roles and Objects.
[8] Trygve Reenskaug, Role Modeling. HYPERLINK "http://folk.uio.no/trygver/themes/roles/roles-index.html" http://folk.uio.no/trygver/themes/roles/roles-index.html.