

# Software-patronen

M.J.B. van Elswijk  
SERC

<b>1. Inleiding .....</b>	<b>2</b>
<b>2. Ontwikkelen van object-georiënteerde software .....</b>	<b>3</b>
2.1 Inleiding .....	3
2.2 Het ontstaan van object-oriëntatie .....	3
2.3 Opbouw van het OO-ontwikkeltraject .....	3
2.4 Beperkingen van OO-ontwikkelmethoden .....	4
2.5 De oorsprong van patronen .....	4
<b>3. Patronen.....</b>	<b>6</b>
3.1 Inleiding.....	6
3.2 Beschrijven van patronen .....	7
3.3 Grafische notatie .....	9
3.4 Classificatie .....	10
<b>4. Verschijningsvormen .....</b>	<b>12</b>
4.1 Inleiding .....	12
4.2 Analysepatronen .....	12
4.3 Architectuurpatronen .....	14
4.4 Ontwerppatronen .....	15
4.5 Idiomen.....	18
<b>5. Patronen in de praktijk .....</b>	<b>21</b>
5.1 Inleiding .....	21
5.2 Ervaringen .....	21
5.3 Gebruik .....	22
5.4 Ontdekken van patronen .....	22
<b>6. Tot besluit .....</b>	<b>24</b>
<b>7. Literatuur .....</b>	<b>25</b>

## 1. Inleiding

Het maken van een goed object-georiënteerd model is niet eenvoudig. Het is een intensief en creatief proces. De ontwerper kan hulpmiddelen als ontwerpprincipes, vuistregels en heuristieken gebruiken, maar zonder een goede dosis creativiteit en ervaring is de kans van slagen klein. Creativiteit is niet eenvoudig aan te leren, maar een ontwerper kan wel leren van de ervaring van anderen.

Naast het ontbreken van een algemene formule om een goed model te maken, is er geen eenduidig begrip van wat een 'goed' model is. Martin Fowler geeft dit als volgt weer: "een model is niet goed of fout, het is meer of minder bruikbaar" [Fowler97]. De bruikbaarheid van een model hangt voor een groot deel af van de context waarin het gebruikt wordt.

Gelukkig wordt de ontwerper niet volledig aan zijn lot overgelaten: hij kan gebruik maken van patronen. Algemeen geformuleerd is een *patroon* (Eng. *pattern*) een beschrijving van een oplossingsstrategie voor een specifiek probleem. Een patroon heeft pas bestaansrecht als de beschreven oplossing in de praktijk meerdere malen succesvol is toegepast. Anders gezegd: het legt ervaring vast. *Software-patronen* beschrijven oplossingen voor software-problemen. Hieronder vallen onder andere ontwerp- en implementatieproblemen.

In dit artikel een overzicht gegeven van de verschillende vormen van patronen voor object-georiënteerde software. Het doel is de lezer bekend te maken met de verschillende soorten software-patronen, de literatuur die daarover verschenen is en hoe ze toegepast kunnen worden. Kennis over object-oriëntatie in het algemeen en object-modellering in het bijzonder wordt daarbij aanwezig verondersteld; zie voor deze informatie bijvoorbeeld [Troyer93] of [Florijn95].

Hoofdstuk 2 beschrijft het ontwikkelen van object-georiënteerde software, zowel vanuit een historisch perspectief als vanuit een procesmatig perspectief. Daarbij wordt kort ingegaan op de rol van patronen binnen beide gezichtspunten.

In hoofdstuk 3 worden een definitie en een overzicht gegeven van de verschillende soorten patronen. Daarnaast wordt aandacht besteed aan de meest gebruikte beschrijvingsvormen en classificaties. Deze kennis is nodig om de boeken die over patronen zijn verschenen snel te kunnen gebruiken.

Hoofdstuk 4 demonstreert het gebruik van de verschillende soorten patronen door middel van het oplossen van een kleine casus. Daarbij wordt dieper ingegaan op de specifieke kenmerken van de diverse patroonvarianten.

Hoofdstuk 5 behandelt het praktisch toepassen van patronen in het algemeen. Het bevat richtlijnen over hoe patronen ingezet kunnen worden en wanneer ze toegepast moeten worden, maar ook enkele ervaringen uit software-projecten waar patronen zijn gebruikt. Tot besluit van het hoofdstuk wordt ingegaan op het 'ontdekken' van (nieuwe) patronen.

## 2. Ontwikkelen van object-georiënteerde software

### 2.1 Inleiding

Software-patronen ontstaan en worden gebruikt tijdens het ontwikkelen van software. In dit hoofdstuk wordt aandacht besteed aan het ontwikkelen van object-georiënteerde programma's. De eerste paragraaf geeft een kort historisch overzicht van object-oriëntatie. De daarop volgende paragraaf beschrijft het proces van OO-softwareontwikkeling. In paragraaf 2.4 worden enkele tekortkomingen van bestaande ontwikkelmethoden behandeld. Ten slotte wordt in de laatste paragraaf de oorsprong van patronen beschreven.

### 2.2 Het ontstaan van object-oriëntatie

Eind jaren zestig werd gesproken van een 'software crisis'. In die periode werden voor het eerst grote applicaties gebouwd, en begonnen het onderhoud en het in stand houden van bestaande software een rol te spelen. De complexiteit van software en de omvang van de systemen, maar ook het ontwikkelproces zelf bleken niet eenvoudig te beheersen. Zowel de ontwikkelingsprocessen als de daarbij opgeleverde producten<sup>1</sup> waren aan verbetering toe.

Het ontwikkelproces moest sneller, effectiever en voorspelbaarder worden. Oplossingen hiervoor waren, onder andere, hergebruik van bestaande software, incrementeel ontwikkelen en het betrekken van toekomstige gebruikers bij het ontwikkelen. De producten (applicaties) en de tussenproducten (ontwerpen) moesten beter onderhoudbaar, aanpasbaar en betrouwbaarder worden. Om daar aan te voldoen werd structuur in de producten aangebracht en werden verschillende ontwerpprincipes bedacht.

Modulair programmeren en het opdelen van systemen in afzonderlijke componenten bleken stappen in de goede richting. Deze ontwikkelingen zouden tenslotte leiden tot het paradigma 'object-oriëntatie', waarbij software is opgebouwd uit zelfstandige, onderling samenwerkende objecten. (Zie voor uitleg en achtergrondinformatie over object-oriëntatie ook [Troyer93] of [Florijn95].)

### 2.3 Opbouw van het OO-ontwikkeltraject

Er zijn veel verschillende methoden om object-georiënteerde software te ontwikkelen, maar bij vrijwel al deze bestaat het ontwikkeltraject uit de volgende drie fasen:

- **analyse** – In de analysefase wordt het op te lossen probleem in kaart gebracht. Het resultaat van deze fase is een objectmodel waarin de klassen, relaties en objecten, die overeenkomen met concepten uit de werkelijkheid, zijn opgenomen.
- **ontwerp** – In de ontwerpfase wordt een oplossing voor het probleem beschreven. Net als in de analysefase wordt een objectmodel gemaakt, maar dit model beschrijft de applicatie of het systeem dat het probleem moet oplossen.
- **implementatie** – Tijdens de implementatiefase wordt het ontwerpmodel omgezet in een werkend systeem (of prototype). Pas in deze fase is de feitelijke keuze van programmeertaal en platform van belang.

Na de implementatiefase moet de gemaakte software uiteraard getest worden. Testen vindt echter tijdens en na alle fasen plaats. De test na de analysefase bestaat uit het controleren of het beschreven probleem(domein) overeenkomt met de werkelijkheid; hierbij is inbreng van gebruikers en domein-

---

<sup>1</sup> Bij producten moet hier niet alleen gedacht worden aan software, maar ook bijvoorbeeld aan ontwerpen, modellen en documentatie.

deskundigen nodig. Na de ontwerpfase moet onder andere gecontroleerd worden of het ontwerpmodel het volledige probleem oplost.

Tijdens de ontwikkeling van grote systemen bevindt zich tussen de analyse en het ontwerp vaak nog de *architectuurfase*. In deze fase worden beslissingen genomen over de globale structuur van de software (de architectuur); beslissingen die overal in de ontwerpfase invloed hebben. Aspecten die bekeken worden zijn bijvoorbeeld de gelaagdheid van het systeem (client-server, multi-tier,...) en de inzet van bepaalde frameworks<sup>2</sup>.

## 2.4 Beperkingen van OO-ontwikkelmethoden

Een mogelijke verklaring voor de (plotselinge) populariteit van patronen kan liggen in het feit dat object-georiënteerde ontwikkelmethoden niet voldoen aan de verwachtingen. De methoden kijken slechts naar de kleinste bouwstenen van de programmatuur: objecten en klassen. Aan de verbanden tussen de bouwstenen en de structuur van de software op grotere schaal wordt vrijwel geen aandacht besteed. Zonder ervaring, goede richtlijnen en hulpmiddelen is de kans op een ‘goed’, herbruikbaar en begrijpelijk ontwerp klein.

Daarnaast zijn de methoden (met opzet) onafhankelijk van de op te lossen problemen, zodat ze voor alle probleemgebieden inzetbaar zijn. Ze zijn daardoor echter zó generiek dat de kans groot is dat ontwikkelaars weinig houvast vinden bij het oplossen van hun specifieke probleem. Volgens [Jackson95] is het nut van een methode omgekeerd evenredig met zijn algemeenheid.

Object-georiënteerde ontwikkelmethoden zijn vooral geschikt voor de analyse- en ontwerpfase van een project. Eén van de tekortkomingen van bestaande methoden is echter dat er nauwelijks ondersteuning is voor ontwerp voor hergebruik en ontwerp met hergebruik. Het maken van herbruikbare software (en modellen) is complex, zeker als de vorm van toekomstig hergebruik nog niet bekend is. Daarnaast kost het meer tijd en geld om ontwerpen generieker te maken dan voor één specifiek probleem nodig is.

Met de term ‘ontwerpen met hergebruik’ wordt bedoeld het toepassen van bestaande software en modellen in het ontwikkeltraject. Om dit efficiënt in de praktijk te kunnen toepassen is een complete organisatiestructuur nodig: ontwikkelaars zijn langer voor hun ontwerpen verantwoordelijk dan het bouwen van één enkel systeem, herbruikbare componenten moeten volledig en duidelijk gedocumenteerd zijn en deze componenten moeten eenvoudig beschikbaar zijn.

Patronen kunnen niet alle bovenstaande problemen oplossen, maar kunnen het leed wel verzachten. Door patronen te gebruiken, is er in elk geval hergebruik van kennis en ervaring. Daarnaast beschrijven ze ‘kleine’ problemen met hun oplossing en zijn daardoor eenvoudiger te documenteren en opnieuw te gebruiken. Veel patronen hebben tot doel de flexibiliteit van de software te vergroten en daarmee de herbruikbaarheid. Dus ontwerpen voor hergebruik kan gedeeltelijk door het op de juiste manier toepassen van bestaande patronen.

Naast patronen kan gebruik gemaakt worden van principes, vuistregels of heuristieken. Zij geven niet een oplossing voor een specifiek probleem, maar beschrijven richtlijnen voor het gehele ontwerp.

## 2.5 De oorsprong van patronen

Eind jaren zeventig had de architect Christopher Alexander zich tot doel gesteld het maken van goede architecturen in de bouwsector te stimuleren met behulp van patronen. Om een kamer, een gebouw of een wijk op de juiste manier te ontwerpen, moest gebruik gemaakt worden van bestaande, bewezen oplossingen. Zijn patronen schreven voor hoe een straat door een wijk moest lopen, in welke richting de

---

<sup>2</sup> Een framework is een ‘skelet’ om applicaties te bouwen, waarbij een groot deel van de applicatie reeds bestaat en waarbij de ontwikkelaars idealiter slechts details hoeven in te vullen.

deuren in de gang van een huis moesten opengaan en waar de ramen in een kamer geplaatst moesten worden.

Een voorbeeld van een dergelijk patroon uit [Alexander77] is het Window Place-patroon, waarmee hij het volgende probleem probeert op te lossen:

*Iedereen heeft graag een zitplaats bij het raam, een erker, en grote ramen met lage vensters waar gemakkelijk een stoel bij geplaatst kan worden.*

In de verder patroonbeschrijving geeft hij een definitie van een *window place*, die op verschillende manieren te interpreteren en in te vullen is. Voorbeelden van window places die Alexander geeft omvatten onder andere een laag venster, een glazen alkoof en een zithoekje bij het raam. Als algemene oplossing voor het probleem geeft hij:

*Maak in iedere kamer, waar men een groot deel van de dag doorbrengt, van minstens één raam een 'window place'.*

De beschrijving wordt afgesloten met een verwijzing naar andere relevante patronen, die bijvoorbeeld beschrijven hoe een alkoof gemaakt kan worden.

Alle patronen van Alexander zijn volgens een vast stramien beschreven: het op te lossen probleem met één of meer voorbeelden, de context waarin de oplossing bruikbaar is, een beschrijving van de oplossing, overwegingen hoe de oplossing uitgevoerd kan worden en gevolgen van het toepassen van het patroon.

Begin jaren negentig werd het idee van patronen voor (object-georiënteerde) software verkend door Ward Cunningham en Kent Beck. Zij bedachten een vijftal patronen voor het ontwerpen van grafische user interfaces. Een ander belangrijk moment was het uitkomen van het proefschrift van Erich Gamma [Gamma91]. Ook anderen, waaronder Peter Coad [Coad92], begonnen aandacht te besteden aan OO-patronen; ze werden opgenomen als onderwerp bij de vooraanstaande conferentie OOPSLA (Object-Oriented Programming, Systems, Languages and Applications).

Hoewel het proefschrift van Gamma geen breed publiek vond, lag het aan de basis van het beroemde boek "Design Patterns – Elements of Reusable Software" [Gamma94] geschreven door de 'Gang-of-Four': Erich Gamma, Richard Helm, Ralph Johnson en John Vlissides. Dit boek heeft ervoor gezorgd dat patronen bij een breed publiek bekend en gewaardeerd werden.

De meeste software-ontwikkelaars en -ontwerpers die het boek lezen, krijgen een gevoel van 'déjà-vu'. De problemen die erin staan zijn algemeen bekend en ook de oplossingen zijn vaak herkenbaar. Het bijzondere aan het boek is vooral dat voor het eerst een heldere en systematische beschrijving van deze oplossingen wordt gegeven, aangevuld met overwegingen hoe en wanneer ze te gebruiken. Het boek maakt het mogelijk om te praten over grootschaligere software-bouwstenen dan klassen.

Sinds het verschijnen van het boek van de Gang-of-Four zijn patronen niet meer weg te denken uit de OO-wereld; vele boeken en artikelen zijn inmiddels verschenen. Jaarlijks worden speciale conferenties gehouden, zoals PLOP (Pattern Languages of Programming), EuroPLOP en ChiliPLOP, die enkel gericht zijn op software-patronen.

In de in 1997 officieel tot standaard verklaarde notatie-methode UML (Unified Modeling Language) zijn speciale constructies opgenomen om patronen weer te geven. In paragraaf 3.3 wordt dieper ingegaan op de grafische notatie van patronen.

## 3. Patronen

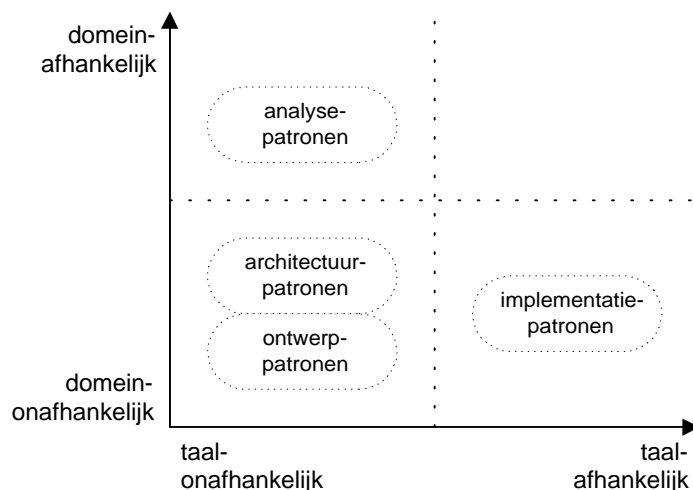
### 3.1 Inleiding

Een *patroon* is een oplossingsstrategie voor een specifiek probleem in een bepaalde context. De problemen waarop patronen zich richten zijn nogal wisselend van aard. In feite kent elk van de vier fasen in het ontwikkeltraject (analyse-, architectuur-, ontwerp- en implementatiefase) zijn eigen soort problemen. Voor elk van deze categorieën bestaan aparte patronen: analysepatronen, architectuurpatronen, ontwerppatronen (design patterns) en implementatiepatronen, die *idiomen* worden genoemd.

Een ontwerpprobleem kan bijvoorbeeld zijn dat een component uit de user interface automatisch gewijzigd moet worden als de toestand van een bepaald object verandert. Om deze afhankelijkheid te modelleren kan het Observer-patroon worden gebruikt. Dit patroon schrijft voor dat objecten, die op de hoogte gebracht moeten worden van wijzigingen in een object (het subject), zich 'abonneren' op die veranderingen; als het object verandert, worden automatisch de abonnees geïnformeerd.

De oplossingen die software-patronen geven, worden beschreven in termen van *rollen*. In het bovenstaande voorbeeld staan twee rollen: de rol van objecten die een bepaald subject observeren en de rol van het subject zelf. In het uiteindelijke ontwerp worden deze rollen ingevuld door specifieke klassen, waarbij het goed mogelijk is dat een klasse meer dan één rol speelt in verschillende patronen.

De eerder genoemde patronen worden gewoonlijk verdeeld volgens de volgende twee criteria: domeinafhankelijkheid en taalafhankelijkheid. Domeinafhankelijkheid wil zeggen dat het toepassingsgebied van patronen beperkt is tot een beperkt aantal domeinen. Taalafhankelijkheid wil zeggen dat de toepasbaarheid van de patronen afhankelijk is van de taal waarin de programmatuur wordt geschreven. In de volgende afbeelding worden de vier soorten patronen schematisch ingedeeld:



**Figuur 1 - Indeling patronen**

Architectuurpatronen en ontwerp-patronen zijn beide zowel taal- als domeinonafhankelijk. Het verschil tussen beide soorten zit in de problemen die ze behandelen en de bijbehorende schaal. Architectuurpatronen beschrijven de structuur van complete software-systemen met behulp van deelsystemen en relaties daartussen. Ontwerp-patronen daarentegen beschrijven een oplossing voor een kleinschalig ontwerp-probleem waarvan de oplossing bedoeld is voor verfijning van een deelsysteem. Architectuurpatronen gebruiken vaak subsystemen als bouwstenen, terwijl ontwerp-patronen elementaire

klassen gebruiken. Een belangrijk verschil in de patronen zit in de consequenties van het gebruik. De keuze van een bepaald architectuurpatroon kan mogelijk invloed hebben op de te gebruiken hardware, netwerkprotocol en platform, terwijl ontwerppatronen enkel de structuur van de software behandelen.

Dit hoofdstuk behandelt het beschrijven, visualiseren en classificeren van patronen. De verschillende soorten patronen worden in het volgende hoofdstuk behandeld en geïllustreerd met enkele voorbeelden. Ze worden daarbij toegepast in een kleine casus.

## 3.2 Beschrijven van patronen

De patronen die Christopher Alexander voor de architectuur van gebouwen beschrijft, zijn allemaal volgens een vast stramien opgezet. Het feit dat alle patronen op dezelfde wijze worden beschreven, is ook van toepassing op software-patronen. Hoewel er geen eenduidige opzet is, bevat de beschrijving van een patroon altijd de volgende componenten:

- **naam** – de naam van het patroon. Dit is een belangrijk aspect; de naam moet pakkend zijn zodat ontwerpers hem gemakkelijk kunnen onthouden en software kunnen bespreken in termen van patronen;
- **probleem** – het probleem dat door het patroon wordt opgelost. De beschrijving moet kort en bondig zijn, zodat ontwikkelaars snel kunnen beslissen of het patroon geschikt is voor hun probleem;
- **context** – de context waarin het patroon gebruikt moet worden. Een probleem en een oplossing staan nooit op zichzelf. Een oplossing is, afhankelijk van de context, meer of minder bruikbaar. Om de ontwikkelaar een beter inzicht te geven in de bruikbaarheid van een patroon, wordt de voor toepassing juiste context beschreven;
- **oplossing** – de oplossing of oplossingsstrategie voor het probleem. De oplossing is meestal tekstueel beschreven, maar kan geïllustreerd worden met een grafische representatie, zoals een scenario.

Daarnaast bevatten de meeste beschrijvingen ook nog één of meer van de volgende componenten:

- **gevolgen** – de gevolgen van het toepassen van het patroon,
- **aanwijzingen** – aanwijzingen voor de toepassing van het patroon,
- **varianten** – verschillende varianten van het patroon,
- **voorbeelden** – voorbeelden van het probleem en van de oplossing,
- **relaties** – relaties met andere patronen en
- **gebruik** – enkele praktijkgevallen waarin het patroon (succesvol) is gebruikt.

Een verzameling van bij elkaar horende patronen wordt traditioneel een *pattern language* (patroon ‘taal’) genoemd [Alexander77]. Een meer precieze definitie van een pattern language vinden we in [Woolf95]: een verzameling patronen die elkaar aanvullen om een compleet domein van problemen op te lossen. De naam is ietwat ongelukkig gekozen omdat ‘taal’ (in wiskundige zin) suggereert dat er een formele beschrijving van de patronen en hun toepasbaarheid is. Een andere naam die gebruikt wordt is ‘*pattern system*’ [Buschmann96].

Als de verzameling meer een opsomming geeft in plaats van strikt samenhangende patronen, wordt gesproken van een *catalogus*. De meest bekende catalogus is waarschijnlijk het boek “Design Patterns – Elements of Reusable Object-Oriented Software” [Gamma94]. De onderstaande tabel bevat het stramien volgens welke de patronen daaruit beschreven zijn. De kolom “Aspect” bevat per onderdeel één of meer van de eerder genoemde tien aspecten van een patroon (naam, probleem, context, etc.). De onderdelen van het patroon zoals ze in het boek staan, zijn opgenomen in de kolom “Gamma”. De vierde kolom bevat een beknopte beschrijving van het onderdeel.

Een andere, veelgebruikte catalogus is “Pattern-Oriented Software Architecture – A System of Patterns” [Buschmann96]. De kolom “Buschmann” van de onderstaande tabel bevat de onderdelen van de patronen uit deze catalogus.

Voorbeelden van de patronen uit beide catalogi zijn opgenomen in hoofdstuk 4.

Gamma	Buschmann	Aspect	Beschrijving
<i>Pattern name</i>	<i>Name</i>	naam	de naam van het patroon
<i>Pattern classification</i>			de classificatie van het patroon; zie volgende paragraaf voor nadere toelichting
<i>Intent</i>	<i>Summary</i>		een beknopte samenvatting van het probleem en de geboden oplossing
<i>Also known as</i>		naam	één of meer alternatieve namen voor het patroon
<i>Motivation</i>	<i>Problem, Example</i>	probleem	een beschrijving van het probleem met een voorbeeld
<i>Applicability</i>	<i>Context</i>	context	een opsomming van de voorwaarden voor zinvolle toepassing van het patroon
	<i>Solution</i>	oplossing	een samenvatting van de oplossingsstrategie
<i>Structure</i>	<i>Structure</i>	oplossing	de structuur van de oplossing weergegeven in een OMT-achtig model
<i>Participants</i>		oplossing	een beschrijving van de rollen in het model
<i>Collaborations</i>	<i>Dynamics</i>	oplossing	een beschrijving van de samenwerking tussen de onderdelen van het model
<i>Consequences</i>	<i>Consequences</i>	gevolgen	de gevolgen van het toepassen van het patroon voor de rest van het ontwerp
<i>Implementation</i>	<i>Implementation, Variants</i>	aanwijzingen, varianten	een aantal adviezen, suggesties en waarschuwingen voor het implementeren van het patroon, aangevuld met enkele varianten van het patroon (als die er zijn)
<i>Sample code</i>		voorbeelden	een uitgewerkt voorbeeld in programmacode
<i>Known uses</i>	<i>Known uses</i>	gebruik	een aantal systemen waar het patroon met succes is toegepast
<i>Related patterns</i>	<i>See also</i>	relaties	een overzicht van de patronen die gerelateerd zijn aan of die vaak gebruikt worden in combinatie met het beschreven patroon

Hieronder is ter illustratie een gedeelte van de beschrijving van het Observer-patroon uit [Gamma94] gegeven.

**Intent** – *Definieer een 1-op-n afhankelijkheid tussen objecten zó dat wanneer een object van toestand verandert, alle afhankelijke objecten op de hoogte gebracht (en eventueel aangepast) kunnen worden.*

**Also known as** – *Dependents, Publish-Subscribe*

**Motivation** – *Het partitioneren van een systeem tot een verzameling samenwerkende klassen leidt meestal tot de noodzaak de consistentie van gerelateerde objecten te bewaken. Het is echter niet wenselijk de consistentie te bereiken door de klassen te sterk te koppelen, want dat verlaagt hun herbruikbaarheid.*

*Veel grafische user interface toolkits scheiden bijvoorbeeld de presentatie in de user interface van de onderliggende applicatie-data.*

...

*Het Observer-patroon beschrijft hoe deze relaties [tussen een presentatie-objecten en onderliggende gegevens] opgezet kunnen worden. De belangrijkste objecten in dit patroon zijn subject en observer. Een subject kan willekeurig veel afhankelijke observers hebben. Alle observers worden op de hoogte gebracht als het subject van toestand verandert. Vervolgens vraagt elke observer de toestand van het subject om zijn eigen toestand weer consistent te maken.*

**Applicability** – *Gebruik het Observer-patroon in een van de volgende situaties:*

- *als een abstractie twee verschillende aspecten heeft, de een afhankelijk van de ander. Encapsulatie van deze aspecten in verschillende objecten maakt het mogelijk ze afzonderlijk te herbruiken en wijzigen.*
- *als het wijzigen van een object wijzigingen in andere objecten ten gevolg heeft en vooraf niet bekend is hoeveel objecten aangepast moeten worden.*

- als een object in staat moet zijn andere objecten te informeren zonder aannames te maken over die objecten. Met andere woorden: de objecten mogen niet sterk gekoppeld zijn.

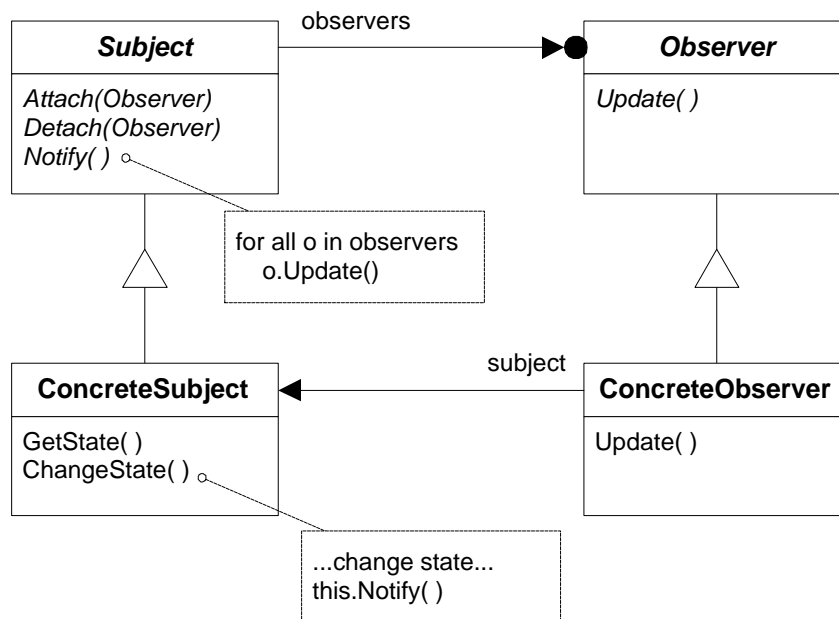
De hierboven gegeven beschrijving moet de ontwikkelaar voldoende informatie geven om te beslissen of het patroon (mogelijk) bruikbaar is voor zijn probleem. De rest van de beschrijving bevat onder andere de structuur van de oplossing en een grafische representatie.

De structuur van de oplossing wordt meestal beschreven door de rollen die verdeeld moeten worden (bijvoorbeeld aan klassen), een scenario-diagram waarbij de communicatie tussen de onderdelen wordt weergegeven en een tekstuele uitleg. Ter verduidelijking van de oplossing wordt meestal een diagram gegeven; de grafische notatie van patronen wordt in de volgende paragraaf verder behandeld.

### 3.3 Grafische notatie

De oplossing die in een patroonbeschrijving gegeven wordt, wordt meestal geïllustreerd met een schematische voorstelling van de structuur. In de tot nu toe verschenen literatuur is daarvoor meestal gebruik gemaakt van een OMT-achtige notatie. Het doel van de illustraties is de lezer snel inzicht te geven in de rollen die klassen en objecten spelen en de interactie daartussen.

Het onderstaande schema komt uit [Gamma94] en toont de structuur van het hierboven genoemde Observer-patroon.

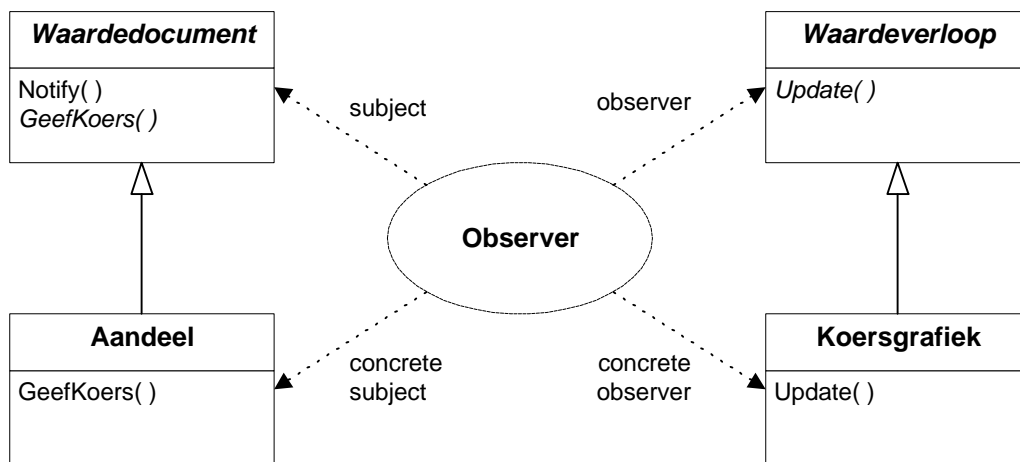


**Figuur 2 - Schematische weergave van het Observer-patroon [Gamma94]**

In het diagram worden vier rollen getoond (Subject, ConcreteSubject, Observer en ConcreteObserver) en de relaties daartussen. Een Subject kent een willekeurig aantal Observers en een ConcreteObserver kent een ConcreteSubject. Daarnaast is in pseudo-OMT weergegeven wat er moet gebeuren als de methode Notify van een Subject wordt uitgevoerd. Deze rollen worden in het uiteindelijke ontwerp ingevuld door specifieke klassen.

De opvolger van de OMT-notatie is UML (*Unified Modeling Language*). UML is een door Rumbaugh, Booch en Jacobson opgestelde notatie-techniek voor object-georiënteerde modellen [UML97, Greefhorst98]. Sinds 1997 is UML een door de Object Management Group (OMG) geaccepteerde standaard. UML biedt voorzieningen om patronen weer te geven; deze worden *collaborations* genoemd. Collaborations zijn diagrammen waarin objecten en de communicatie daartussen worden gevisualiseerd.

Een patroon kan in UML ook als bouwsteen in een (groter) ontwerp worden gebruikt. Het wordt dan weergegeven als een ellips (zie Figuur 3). De klassen die rollen binnen het patroon vervullen worden aangeduid met pijlen. Deze notatie biedt de mogelijkheid om patronen als aparte bouwstenen van het ontwerp te beschouwen. Het ontbreekt momenteel echter nog aan goede ondersteuning door ontwikkeltools.



Figuur 3 - Het Observer-patroon als ontwerpbouwsteen

### 3.4 Classificatie

Zoals in paragraaf 3.1 al is aangegeven, biedt een patroon een oplossing voor een specifiek probleem. Om uit het grote aanbod van patronen het juiste te kunnen kiezen, moeten ze op een of andere manier geclassificeerd worden. Een gangbare classificatie is hierboven al beschreven: analyse-, architectuur-, ontwerp- en implementatiepatronen. Deze verdeling is gemaakt op basis van het soort model waarop en de fase waarin het patroon toepasbaar is. Andere criteria voor classificatie kunnen zijn: het doel van het patroon, het type probleem dat wordt opgelost en de soort oplossing dat het patroon biedt.

In de praktijk blijkt het groeperen van gerelateerde patronen in pattern languages handig. Zo bestaan er pattern languages voor het modelleren van specifieke domeinen, zoals voor brandalarmen [Molin97] en wegtransport [Foster97]. Daarnaast zijn er pattern languages voor bepaalde families van problemen, zoals voor het ontwerpen van een relationele database voor OO-programma's [Brown96], persistente objecten [Soukup94] en voor communicatie-software [Schmidt95].

Een andere classificatie wordt voorgesteld in [Buschmann96]. Daar is het soort probleem dat het patroon oplost als criterium gebruikt. De patronen worden ingedeeld in de volgende categorieën:

- **from mud to structure** – patronen die decompositie van een systeemtaak in samenwerkende deeltaken ondersteunen. Het doel van deze patronen is de structuur van het systeem te verbeteren;
- **distributed systems** – patronen die infrastructuur beschrijven voor systemen waarvan de componenten verdeeld zijn over verschillende processen en subsystemen;
- **interactive systems** – patronen die interactie met gebruikers ondersteunen;

- **adaptable systems** – patronen voor systemen die door regelmatig veranderende eisen aanpasbaar en uitbreidbaar moeten zijn;
- **structural decomposition** – patronen die voorzien in decompositie van een systeem of complexe componenten in samenwerkende onderdelen;
- **organization of work** – patronen die definiëren hoe componenten moeten samenwerken om een complexe taak te vervullen;
- **access control** – patronen die de toegang tot diensten en componenten bewaken en controleren;
- **management** – patronen die het beheer van verzamelingen objecten, diensten en componenten ondersteunen;
- **communication** – patronen voor de communicatie tussen componenten;
- **resource handling** – patronen die helpen bij het beheren van gemeenschappelijke componenten en objecten.

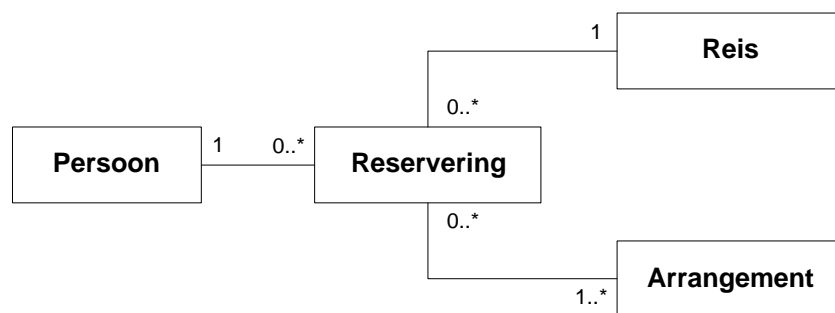
De beschrijving, visualisatie en classificatie van patronen zijn onafhankelijk van het soort patroon dat wordt beschreven. In het volgende hoofdstuk wordt dieper ingegaan op de verschillende verschijningsvormen van patronen.

## 4. Verschijningsvormen

### 4.1 Inleiding

In de hierna volgende vier paragrafen worden de vier verschillende soorten patronen besproken. Om de praktische toepassing van patronen te demonstreren wordt een kleine casus gebruikt. Het is echter niet de bedoeling een volledig systeem te beschrijven – de nadruk ligt op het gebruiken van de patronen.

Het te ontwerpen systeem is bedoeld voor de verwerking van reserveringen binnen een klein reisbureau. Het reisbureau biedt klanten reizen naar verschillende bestemmingen aan, mogelijk aangevuld met speciale arrangementen (bijvoorbeeld een stedenreis). Een eerste vluchtige analyse levert het volgende objectmodel:



Figuur 4 – Globaal analysemodel van de casus

Het te bouwen systeem moet de baliemedewerkers ondersteunen met behulp van een grafische interface. De systemen die momenteel voor klant- en reserveringsadministratie worden gebruikt, moeten aangesloten worden op het nieuwe systeem.

### 4.2 Analysepatronen

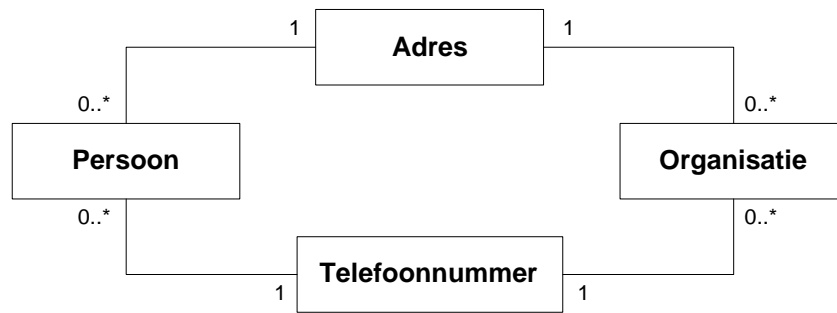
#### Algemeen

Analysepatronen leggen fragmenten uit een bepaald domeinmodel vast. Ze bevatten domeinkennis en hebben meestal tot doel het domeinmodel, waarop het latere ontwerpmodel gebaseerd wordt, flexibeler of generieker te maken. Deze patronen worden geclassificeerd aan het hand van het domein waartoe ze behoren. In [Fowler97] wordt een groot aantal analysepatronen beschreven; de patronen die voor de casus worden gebruikt zijn daaruit afkomstig.

Er wordt gewerkt aan analysepatronen voor veel verschillende domeinen, zoals bijvoorbeeld voor financiële administratie, medische toepassingen, telecommunicatie en planningssystemen. De algemene beschikbaarheid van deze patronen kan worden beperkt door het feit dat ze vaak bedrijfsspecifieke kennis vastleggen – uit concurrentieoverweging kan worden besloten ze niet openbaar te maken.

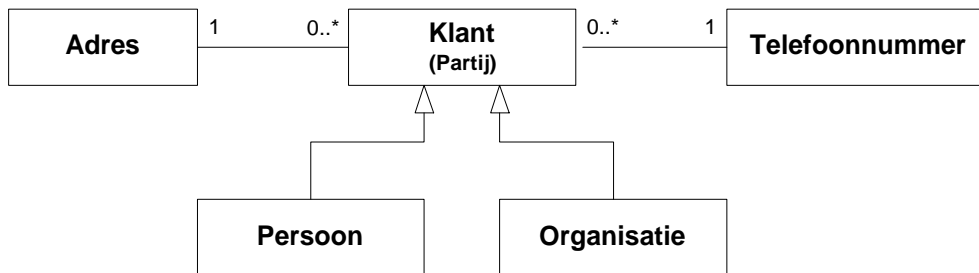
#### Toepassing

Tijdens het uitvoeren van de analyse blijkt dat niet alleen individuen een reis boeken, maar ook organisaties (bijvoorbeeld een sportvereniging). Zowel individuen als organisaties hebben een postadres en een telefoonnummer. Dit is in de volgende afbeelding weergegeven.



**Figuur 5 – Klassenmodel van Persoon en Organisatie**

In dergelijke situaties kan het Party-patroon worden gebruikt. Het patroon schrijft voor dat alle klassen, zoals Organisatie en Persoon, die adresgegevens (en telefoonnummers) hebben, een subklasse moeten worden van een klasse Partij (Party). De rol van ‘Partij’ kan in de casus gespeeld worden door de (nieuwe) klasse ‘Klant’.

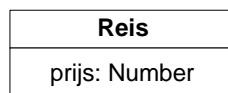


**Figuur 6 – Klassenmodel met Persoon en Organisatie als subklasse van Klant**

Een voordeel van dit patroon is het feit dat alle klanten (personen of organisaties), uniform beschreven zijn. Als bijvoorbeeld voor het benaderen van klanten in de toekomst gebruik gemaakt kan worden van e-mail, hoeft enkel de klasse *Klant* te worden aangepast.

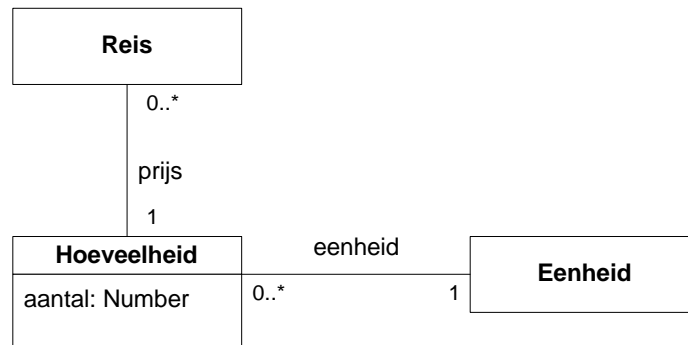
De applicatie kan nu reserveringen maken voor een ‘Klant’ en waar nodig personen gelijk behandelen als organisaties (polymorfisme). Het patroon is herkenbaar als analysepatroon omdat het structuur van domeinklassen (Persoon, Adres, ...) beschrijft.

Een ander, eenvoudig analysepatroon is het Quantity-patroon. Dit patroon schrijft voor dat op alle plaatsen waar een numeriek attribuut wordt gebruikt (zie voorbeeld in Figuur 7) en waar bij dit attribuut eenheden, zoals kilo’s, meters en guldens, zijn weggelaten, gebruik gemaakt moet worden van een object van klasse Hoeveelheid (Quantity).



**Figuur 7 – Klasse Reis met een numeriek attribuut**

De klasse Hoeveelheid heeft één attribuut *aantal* en een verwijzing naar een Eenheid.



**Figuur 8 – Klasse Reis en klasse Hoeveelheid**

Eén van de voordelen van de nieuwe structuur is dat nu altijd duidelijk is welke eenheden worden gebruikt. Daarnaast kan de klasse Eenheid eenvoudig worden uitgebreid met berekeningen om waarden naar een andere Eenheid om te zetten, bijvoorbeeld van kilogrammen naar milligrammen of van dollars naar gulden.

### 4.3 Architectuurpatronen

#### Algemeen

De architectuurpatronen beschrijven de globale structuur van het te ontwikkelen systeem. De meeste van deze patronen verdelen het systeem in kleinere componenten en geven de relaties en afhankelijkheden hiertussen aan. De componenten uit een architectuurpatroon komen meestal niet direct overeen met één enkele klasse uit het domeinmodel, maar eerder met een groep klassen uit het ontwerpmodel.

#### Toepassing

De te bouwen software moet gebruik maken de bestaande systemen voor klant- en reserverings-administratie (er is communicatie nodig met de databases van beide systemen). Van alle onderdelen van het systeem zal de gebruikersinterface het meest frequent (moeten kunnen) veranderen. Er is dus een architectuur nodig die verschillende componenten van het systeem ontkoppelt.

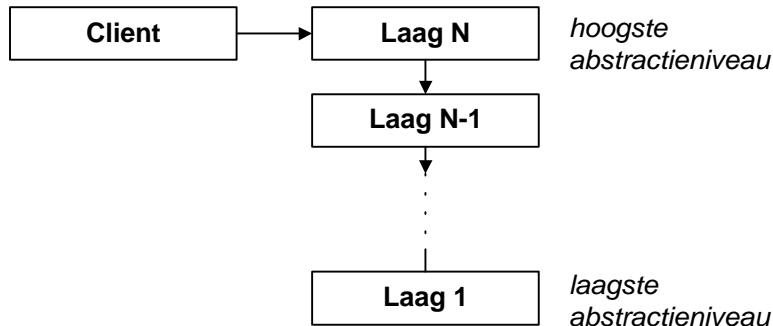
Patronen die hiervoor mogelijk te gebruiken zijn, zijn het Three-Tier Architecture patroon [Aarsten96] of het meer algemene Layers-patroon [Buschmann96]. Beide patronen geven als oplossing dat het systeem opgedeeld moet worden in verschillende ‘lagen’ (layers, tiers).

Volgens het Three-Tier Architecture patroon moet het systeem in de drie volgende lagen worden verdeeld:

- een client-laag – deze bevat de gebruikersinterface;
- een applicatielaag of functielaag – hierin wordt de ‘bedrijfslogica’ beschreven. De laag bevat domeinspecifieke objecten en hun relaties;
- een data-laag – in deze laag bevinden zich de database en andere bestaande systemen waarmee gecommuniceerd moet worden.

Het Layers-patroon beschrijft de oplossing algemener: verdeel het systeem in een aantal lagen en plaats die bovenop elkaar; begin met het laagste abstractieniveau en plaats er steeds een laag van een hoger niveau op (zie Figuur 9). Een laag mag alleen gebruik maken van de diensten van een laag direct onder

hem. Een bekende toepassing van dit patroon is het OSI-model [Tanenbaum92] waarin netwerklagen volgens de bovenstaande wijze worden gedefinieerd.



**Figuur 9 - Verdeling van systeem in lagen**

Beide architectuurpatronen leggen de structuur van de software op een hoog niveau vast. Hoewel de een veel algemener is dan de ander, verdelen ze beide de verantwoordelijkheden binnen het systeem. Een laag in het architectuurontwerp komt meestal niet overeen met één specifieke klasse, maar met een aantal klassen.

Hoe de feitelijke verdeling van het systeem in lagen tot stand komt, ligt buiten de strekking van dit artikel. In de beschrijving van het Layers patroon in [Buschmann96] wordt een aantal overwegingen gegeven voor de implementatie. In hetzelfde boek zijn nog meer architectuurpatronen te vinden zoals bijvoorbeeld Blackboard, Broker, Model-View-Controller en Microkernel. Al deze patronen beschrijven de structuur van de software op een hoog abstractieniveau waarbij ze specifieke problemen oplossen. Het is goed mogelijk dat het uiteindelijke systeem gebruik maakt van meer dan één architectuurpatroon.

## 4.4 Ontwerppatronen

### Algemeen

Ontwerppatronen (design patterns) beschrijven een oplossing of oplossingsstrategie voor een vaak terugkerend ontwerpprobleem. Door hun taal- en domeinonafhankelijkheid en het beschikbare aanbod zijn dit waarschijnlijk de meest toegepaste en meest bruikbare patronen.

In [Gamma94] wordt de classificatie van ontwerppatronen volgens twee criteria bepaald: scope en doel. De scope van een patroon geeft aan of het betrekking heeft op klassen of op objecten. Volgens de catalogus kunnen patronen drie doelen dienen:

- **creatie-patronen** (*creational patterns*) – hebben betrekking op de creatie van objecten;
- **structuur-patronen** (*structural patterns*) – schrijven een structuur van samenwerkende klassen en objecten voor;
- **gedrag-patronen** (*behavioral patterns*) – behandelen het gedrag en de verantwoordelijkheden van klassen en objecten.

Een ontwerppatroon wordt meestal vergezeld van een klein, schematisch objectmodel waarin de structuur van de oplossing is weergegeven. Hoewel het lijkt alsof er specifieke klassen in deze modellen staan, beschrijven ze de rollen die klassen spelen. Klassen kunnen, zoals eerder vermeld, meer dan één rol spelen.

## Toepassing

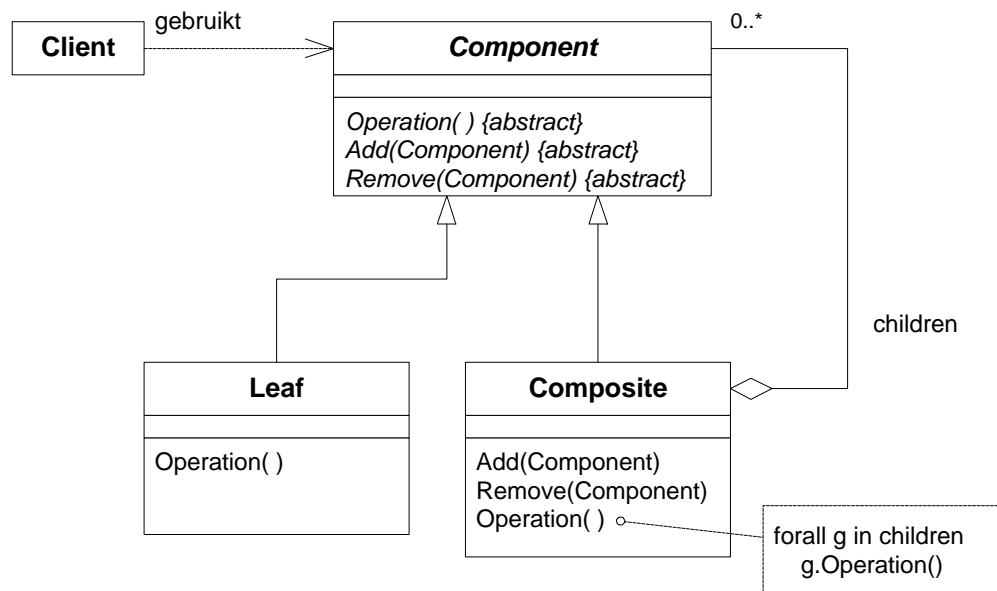
De arrangementen die het reisbureau aanbiedt, kunnen zelf weer bestaan uit 'kleinere' arrangementen. Een stedenreis kan bijvoorbeeld bestaan uit een hotel-arrangement en een wandeltocht. Alle arrangementen hebben een prijs.

Om dit op te lossen, biedt de catalogus van Gamma het Composite-patroon (een structuur-patroon). Bij de voorwaarden voor toepassing staat:

*Gebruik het Composite-patroon als*

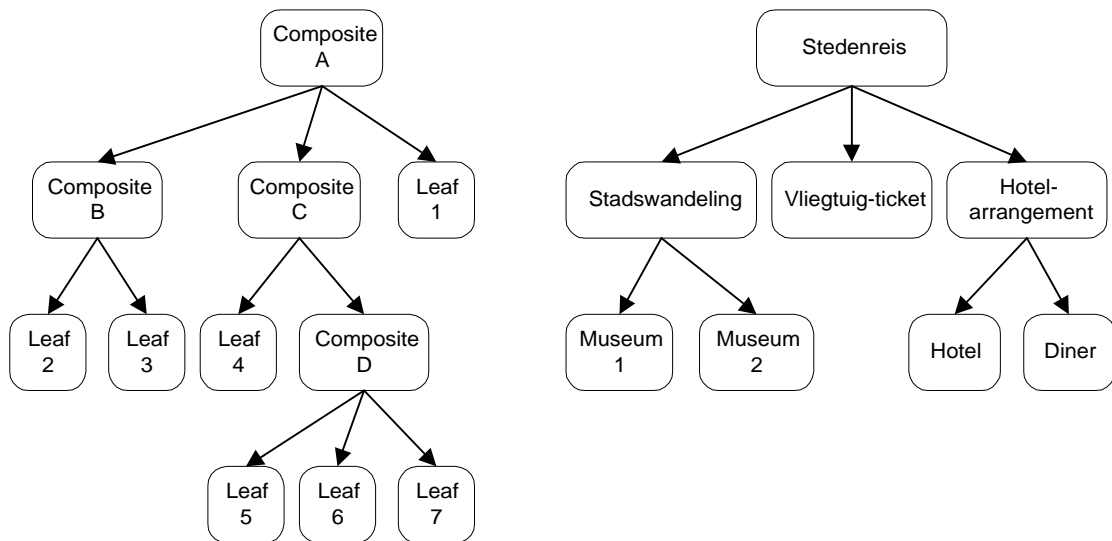
- je “onderdeel-geheel” (part-whole) hiërarchieën wilt representeren.
- gebruikers van de objecten geen onderscheid hoeven te maken tussen een samengesteld en een enkelvoudig object.

De structuur van de oplossing is in het volgende model weergegeven:



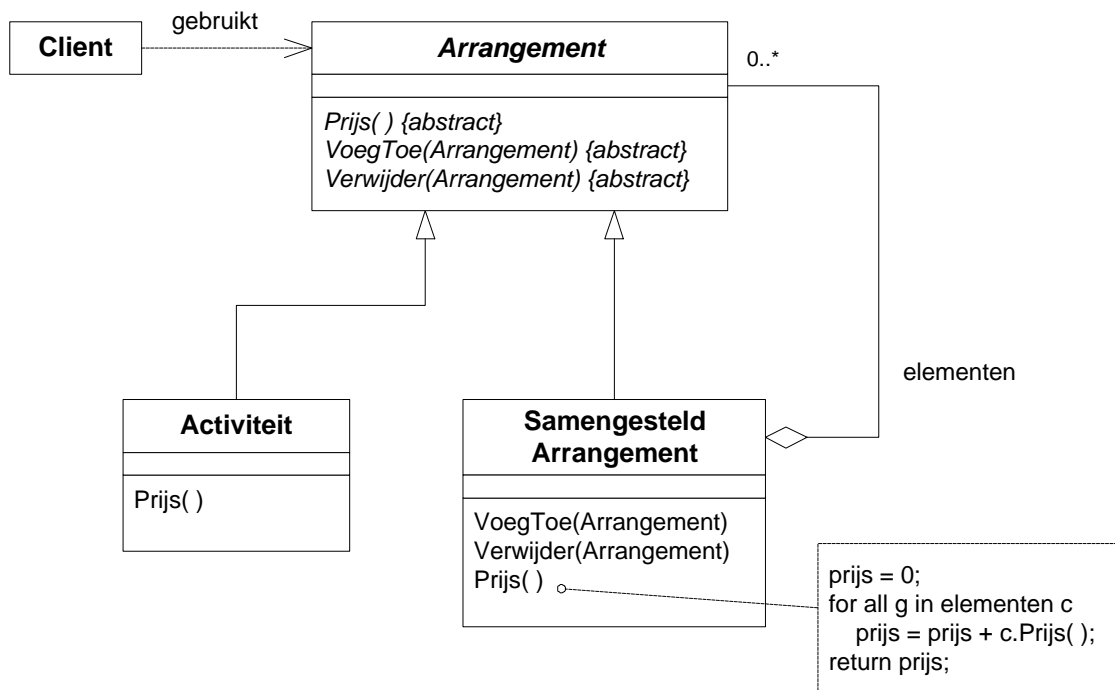
**Figuur 10 - Structuur van het Composite-patroon**

De abstracte klasse **Component** beschrijft de interface die enkelvoudige en samengestelde objecten gemeen hebben. De klasse **Leaf** definieert de structuur en het gedrag van de enkelvoudige objecten. Bij toepassing van het patroon kunnen verschillende klassen de rol van 'Leaf' spelen. Objecten van de klasse **Composite** bevatten kinderen, die zelf weer instanties van **Component** zijn. Op deze manier kan een willekeurige hiërarchie van objecten worden gemaakt, waarbij de samenstelling als geheel op dezelfde manier behandeld kan worden als elk van de onderdelen. Het onderstaande voorbeeld toont een mogelijke structuur van een object na het toepassen van het bovenstaande patroon en een voorbeeld van een dergelijke structuur voor de casus.



**Figuur 11 - Voorbeeldstructuur na het toepassen van Composite**

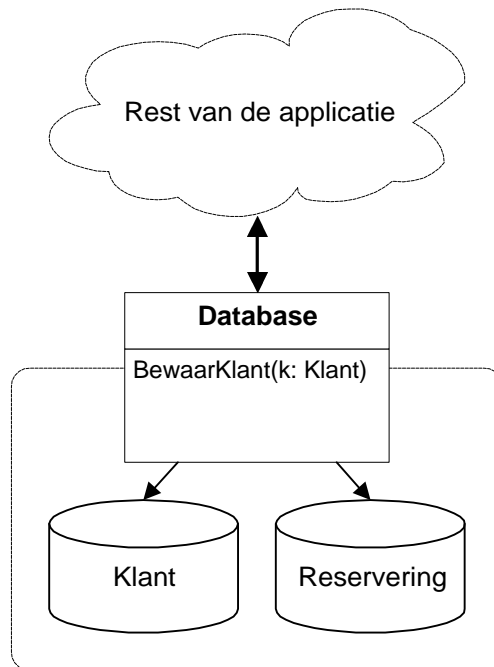
Als we het patroon toepassen in de casus krijgen we het onderstaande modelfragment. De prijs van een samengesteld arrangement wordt gedefinieerd als de som van prijzen van de onderdelen.



**Figuur 12 - Toepassing van het Composite-patroon**

Volgens de specificatie moeten de bestaande systemen voor klant- en reserveringsadministratie gebruikt worden. Tijdens het ontwerp wordt besloten dat klanten en hun reserveringen bij elkaar horende begrippen zijn; transacties moeten dus alle wijzigingen naar klanten en reserveringen atomair verwerken. Het Facade-patroon kan hierbij een oplossing bieden.

Het doel van het Facade-patroon is het maken van één enkele interface naar een aantal interfaces van subsystemen. Deze enkele interface biedt meestal een hoger abstractieniveau dan de onderliggende interfaces.



**Figuur 13 - Voorbeeldstructuur na het toepassen van Facade**

Er wordt een interface gedefinieerd met als één van de operaties het wegschrijven van een klant en bijbehorende reserveringen; dit is de operatie *BewaarKlant*. Die operatie wordt volgens Two-Phase Commit-principes aan de rest van het systeem aangeboden; de componenten die gebruik maken van deze operatie weten niets van de fysieke representatie van de gegevens.

In de casus is de structuur van de database gegeven. Als die structuur echter niet bekend is, zal deze moeten worden ontworpen aan de hand van de klassendefinities uit het programma-ontwerp. Hiervoor is een pattern language beschikbaar genaamd “Crossing Chasms” [Brown96]. Met behulp van deze patronen kan stapsgewijs de tabelstructuur worden bepaald. De patronen geven oplossingen voor het modelleren van objecten, relaties en overerving in een relationele database.

## 4.5 Idiomen

### Algemeen

Idiomen zijn patronen die programmeertaal-specifieke oplossingen beschrijven. Inmiddels zijn er patronen verschenen voor C++ [Coplien92], Smalltalk [Beck97] en Java [Lea96]. Deze patronen zijn grofweg te verdelen in twee groepen. De eerste groep patronen beschrijft implementatie-oplossingen, meestal voor een probleem van een laag abstractieniveau of voor een specifieke implementatie van een ontwerppatroon. Voorbeelden hiervan zijn patronen voor persistentie, garbage collection en de implementatie van het Observer-patroon. Zij geven dus structuur aan de uiteindelijke software.

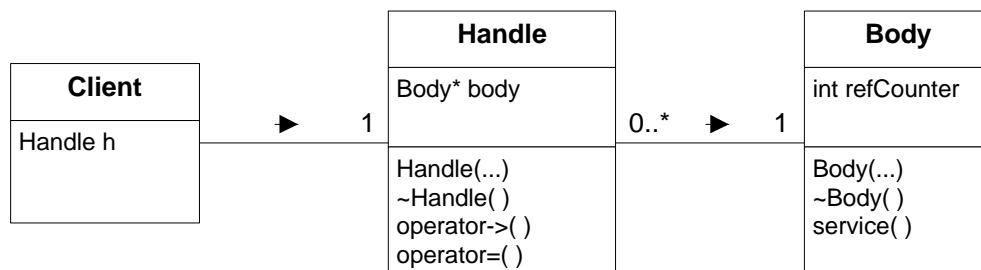
De andere groep patronen geeft aanwijzingen voor het helder en leesbaar programmeren en schrijven bijvoorbeeld een naamconventie voor variabelen en methoden voor. Deze patronen geven structuur aan de broncode en een verzameling ervan wordt ook wel een *style guide* genoemd.

Om idiomen te kunnen begrijpen, is kennis van de bijbehorende programmeertaal onontbeerlijk. Als voorbeeld is gekozen voor een C++-idiom.

### Toepassing

De Smalltalk- en Java-omgeving voorzien in automatische garbage collection (het opruimen van objecten waarnaar niet meer verwezen wordt door andere objecten). In C++ is dit echter niet geregeld. Het Counted Pointer-patroon [Coplien92, Buschmann96] biedt een oplossing voor dit probleem.

Het idee is dat het aantal referenties naar een object geteld wordt. Als dit aantal nul wordt, zijn er geen referenties meer en zullen er ook geen referenties meer komen. Het object kan dan ‘weggegooid’ worden – het geheugen dat het object gebruikt kan worden teruggegeven aan het systeem. Referenties gaan niet meer direct naar het betreffende object, maar via een zogenaamde *handle*. In Figuur 14 staat de structuur in klassen weergegeven.



**Figuur 14 - Structuur van het Counted Pointer-patroon**

Het patroon maakt gebruik van een aantal specifieke C++-eigenschappen, zoals *friend* klassen en het herdefiniëren van operatoren (in dit geval de operator `->`). Hieronder staat het patroon in C++-code uitgewerkt. Voor de duidelijkheid zijn enkel de relevante onderdelen getoond.

```

class Body {
public:
    void service();
    // de diensten die beschikbaar zijn
private:
    friend class Handle;
    :
    :
    int refCounter;
    // het aantal referenties naar de instantie
};
  
```

De instanties waarvan de referentie geteld moeten worden, zijn van de klasse `Body`. Het aantal referenties wordt bijgehouden in het attribuut `refCounter`.

```

class Handle {
public:
    Handle(...) {
        // als er een nieuw object wordt aangemaakt is er één
        // referentie
        body = new Body(...);
        body->refCounter = 1;
    }

    Handle& operator=(const Handle &h) {
        // als een variabele van klasse Handle een nieuwe waarde
        // krijgt moet het aantal referenties naar de oude 'body'
        // omlaag en naar de nieuwe omhoog
        h.body->refCounter++;
        if(--body->refCounter <= 0) delete body;
        body = h.body;
    }

    ~Handle() {
        // als de Handle wordt weggegooid moet het aantal
        // referenties omlaag
        if(--body->refCounter <= 0) delete body;
    }

    Body* operator->() {
        // door de operator -> te herdefiniëren, kan de 'body'
        // eenvoudig worden benaderd
        return body;
    }

private:
    Body* body;
    // verwijzing naar de feitelijke instantie
};

```

De klasse Handle verzorgt het verhogen en verlagen van de referentie-teller, alsmede het creëren van nieuwe Body-objecten en het weggooien ervan. Zoals vaak bij idiomen draait de oplossing om taalspecifieke constructies. Doordat de operator -> opnieuw gedefinieerd kan worden is de volgende constructie mogelijk:

```

Handle h(...);
h->service();

```

De programmeur merkt dus niets van het extra niveau van indirectie dat is aangebracht.

Het idioom maakt gebruik van typische C++-constructies. De oplossing is, na aanpassing, ook voor sommige andere programmeertalen te gebruiken. Als de taalafhankelijkheid wordt verwijderd, spreken we echter van een ontwerp patroon.

De enige manier om de bruikbaarheid van idiomen, en software-patronen in het algemeen, te kunnen bepalen, is door ze toe te passen (*the proof of the pudding is in the eating*). In het volgende hoofdstuk worden de ervaringen met het toepassen van patronen samengevat.

## 5. Patronen in de praktijk

### 5.1 Inleiding

De interesse naar software-patronen bestaat nog niet zo lang. Om de bruikbaarheid ervan te kunnen toetsen is toepassing nodig in de praktijk. Een af en toe gehoord bezwaar is dat er veel over patronen wordt geschreven en gepubliceerd, maar dat ze nog te weinig structureel worden toegepast. Toch zijn er al kleine en grote software-projecten uitgevoerd met behulp van deze patronen.

In dit hoofdstuk wordt aandacht besteed aan de opgedane ervaringen bij het gebruiken van software-patronen in de praktijk.

### 5.2 Ervaringen

Doordat patronen een vrij nieuw begrip zijn en ze daarnaast veelal tot doel hebben de software aanpasbaar en onderhoudbaar te maken, ligt het toepassingsgebied van de patronen ligt typisch in de academische wereld en binnen bedrijfstakken die snel veranderen qua informatievoorziening. Systemen waarbij patronen met succes zijn toegepast omvatten communicatiesoftware, financiële systemen, gedistribueerde systemen en grafische toepassingen. Deze successen worden onder andere gemeld door Motorola, Ericsson, Kodak, Siemens en AT&T.

De meest in het oog springende ervaringen zijn in de volgende lijst opgenomen:

- **Patronen verrijken de woordenschat van de ontwerpers en vergemakkelijken de communicatie** – Er wordt niet alleen meer gesproken in termen van objecten en klassen, maar in termen van patronen: “Voor dit probleem nemen we een Composite en voor dat probleem het Quantity-patroon.” Voorwaarde hiervoor is natuurlijk dat ontwerpers dezelfde patronen (goed) moeten kennen. Patronen worden hierdoor logische bouwstenen om over software te praten. Dit wordt door John Vlissides, consultant bij IBM, als het grootste voordeel van patronen beschouwd [Beck96].
- **Patronen zijn niet per se object-georiënteerd** – Hoewel de meeste patronen zijn ontstaan in de OO-wereld is er niets in het patroonidee dat de beschreven oplossingen beperkt tot object-georiënteerde oplossingen. Elke programmeertaal, elk paradigma en elk domein heeft zijn eigen patronen.
- **Patronen zijn eenvoudig toe te passen** – Dit is deels te danken aan het feit dat ze ‘kleine’ problemen beschrijven die vaak voorkomen. Anderzijds zijn de beschrijvingen niet formeel en niet bijzonder gedetailleerd, zodat ze ruim geïnterpreteerd kunnen worden; desondanks lossen ze concrete problemen op. Een bijkomend voordeel hiervan is dat ontwikkelaars minder snel last hebben van het ‘*not-invented-here*’-syndroom: de toepassing van patronen geeft de ontwikkelaar voldoende keuzevrijheid.
- **Patronen verdwijnen in de programma-code** – Als na analyse en ontwerp de implementatie wordt gemaakt en de programma-code wordt geschreven, is uit deze code vaak niet meer precies af te leiden welke patronen gebruikt zijn. Ontwerpen met patronen levert weliswaar onderhoudbare modellen op, maar om de uiteindelijke software ook (op dezelfde wijze) onderhoudbaar te maken, is extra inspanning nodig. In de toekomst zal dit probleem mogelijk worden opgelost door het inzetten van tools, waarmee software-patronen en programma-code geïntegreerd kunnen worden. Zie [Florijn97] voor een beschrijving van een dergelijke tool.

Het is niet algemeen te specificeren hoeveel programmacode direct ontstaat uit het toepassen van patronen. Maar hoewel het sterk afhangt van de soort applicatie die gebouwd wordt, is een verhouding van 80%-20% goed mogelijk, waarbij 80% van de code direct is afgeleid uit patronen. Er is een geval bekend van een electronic mail-applicatie die voor 90% uit ‘patrooncode’ bestaat.

- **Patronen leveren soms meer op dan bedoeld** – Deze meerwaarde kan een positief en een negatief karakter hebben. Software-patronen zijn veelal bedoeld om de flexibiliteit en de herbruikbaarheid van ontwerpen te vergroten. Het komt geregeld voor dat bij het toepassen van patronen de mate van flexibiliteit meer toeneemt dan bedoeld of verwacht. Een nadelig effect hiervan kan zijn dat het ontwerp complexer wordt en minder begrijpelijk; dit wordt grotendeels veroorzaakt door het toenemen van wederzijdse afhankelijkheden tussen onderdelen van het ontwerp.
- **Patronen kunnen leiden tot overenthousiasme** – Het gevaar bestaat dat ontwerpers proberen voor al hun problemen patronen te vinden en toe te passen. Dit kan leiden tot onleesbare en niet-implementeerbare ontwerpen.

### 5.3 Gebruik

Om patronen effectief in te kunnen zetten is de aanwezigheid van één of meer schriftelijke of elektronische catalogi onontbeerlijk. Een dergelijke catalogus moet lezers in staat stellen snel het juiste patroon voor een probleem te vinden. Classificatie van patronen (zie paragraaf 3.4) kan het zoeken daarbij versnellen.

Het zoeken en gebruiken van patronen bestaat vrijwel altijd uit de volgende stappen:

1. De eerste stap bij het kiezen van een patroon, gegeven een specifiek probleem, is het classificeren van het probleem volgens de criteria van de catalogus. Dit betekent dat het duidelijk moet zijn wat het probleem is.
2. De volgende stap is het vinden van patronen die volgens de beschrijving het probleem helemaal of gedeeltelijk oplossen. De meeste catalogi voorzien in een samenvatting per patroon om dit selecteren te vergemakkelijken. Patronen die het probleem volledig oplossen verdienen de voorkeur boven gedeeltelijke oplossingen.
3. Als één of meer patronen gevonden zijn, moeten de consequenties van het gebruik ervan bestudeerd worden. Sommige patronen kunnen ongewenste neveneffecten hebben en kunnen daarom afvallen.
4. Wanneer het juiste patroon gevonden is, kan het in het model gebruikt worden. Hiervoor is het nodig de structuur en de dynamiek van het patroon goed te begrijpen. De rollen die worden gedefinieerd moeten worden toegekend aan nieuw te definiëren of bestaande klassen. Mogelijke varianten van het patroon moeten hierbij bestudeerd worden. Deze varianten zijn meestal opgenomen als onderdeel van de patroonbeschrijving.

De tijd die de bovenstaande stappen kosten, hangen af van de ervaring van de ontwerper. Een ontwerper die goed zijn weg weet in de catalogus en de patronen kent, zal de eerste drie stappen snel kunnen uitvoeren.

Bij Siemens werken ontwikkelaars al enige tijd met allerlei soorten software-patronen. Het vinden van het juiste patroon uit een grote voorraad bleek, vooral in het begin, niet eenvoudig; de patroongebruikers vroegen om elektronische ondersteuning. Alle patronen zijn daar online beschikbaar gemaakt met behulp van HTML-pagina's en voor alle medewerkers toegankelijk.

### 5.4 Ontdekken van patronen

Patronen leggen oplossingen vast waarvan de werking bewezen is in de praktijk. 'Officiële' vuistregel hierbij is dat er minstens drie verschillende voorkomens van het patroon in bestaande software moeten bestaan. Patronen worden dus niet uitgevonden, maar ontdekt!

Het zoeken naar patronen in bestaande software wordt *pattern mining* genoemd. Hierbij wordt gekeken naar minimaal drie verschillende bestaande software-systemen die ontwikkeld zijn door verschillende personen of teams. Als deze patronen gevonden zijn, moeten ze beschreven en geclassificeerd worden op één of meer van de eerder beschreven manieren.

Naast het herbruiken van persoonlijke ervaring of die van collega's kan gebruik gemaakt worden van de ervaring van vele anderen. Patronen kunnen uiteraard gevonden worden in boeken (bijvoorbeeld [Gamma94], [Buschmann96], [Fowler97], [PLOP1], [PLOP2] en [PLOP3]), maar ook op Internet. De *Patterns Homepage* (<http://hillside.net/patterns/patterns.html>) bevat vele verwijzingen naar patroonbeschrijvingen, onderzoek op het gebied en literatuurreferenties. Er wordt veel gecommuniceerd via Internet over (nieuwe) patronen; de meeste boeken over patronen zijn geschreven en verbeterd aan de hand van elektronische discussies, bijvoorbeeld via een mailing list.

Een voorbeeld van het zoeken naar patronen is te vinden bij AT&T. Dit telecommunicatie heeft betrouwbaarheid en fouttolerantie van systemen hoog in het vaandel. Het is gebleken dat deze eigenschappen voor een groot deel te vinden zijn in patronen. Deze patronen zijn niet alleen verzameld bij software-ontwikkelaars, maar ook bij onderhouds- en beheerpersoneel. Een groot aantal algemene patronen is vastgelegd in [Coplien92].

## 6. Tot besluit

Software-patronen stellen analisten, ontwerpers en programmeurs in staat om gebruik te maken van ervaring van anderen. De patronen bieden bewezen oplossingen voor specifieke problemen en parate kennis ervan kan het ontwikkelproces versnellen en de uiteindelijke producten verbeteren. De oplossingen die de patronen beschrijven zijn niet nieuw, maar het feit dat ze systematisch benoemd en beschreven worden is dat wel.

Er wordt veel gezegd en geschreven over patronen – er is zelfs sprake van enige hype in de OO-gemeenschap. Op veel plaatsen worden patronen toegepast en worden meer ervaringen verzameld. In de toekomst is verdere ondersteuning door tools en methoden te verwachten, zeker nu UML, die een notatie voor patronen voorschrijft, tot standaard is verheven. Het enthousiasme over patronen slaat ook over naar domeinen anders dan het ontwerpen en implementeren van object-georiënteerde software, zoals bijvoorbeeld datamodellering [Hay95], organisatiekunde en project management. Het ziet er naar uit dat patronen geen eendagsvlieg zijn.

## 7. Literatuur

- [Aarsten96] A. Aarsten, D. Brugali, G. Menga, *Patterns for Three-Tier Client/Server Applications*, 3<sup>rd</sup> Conference on Pattern Languages of Programs, 1996
- [Alexander77] C. Alexander, S. Ishikawa, M. Silverstein, *A Pattern Language*, Oxford University Press 1977
- [Beck96] K. Beck et al, *Industrial Experience with Design Patterns*, ICSE 1996, IEEE Software, 1996
- [Beck97] K. Beck, *Smalltalk Best Practices – 1 Coding*, Prentice Hall, 1997
- [Brown96] K. Brown, *Crossing Chasms: A pattern language for Object-RDBMS integration*, Pattern Languages of Program Design 2, Addison-Wesley, 1996
- [Buschmann96] F. Buschmann et al., *Pattern-Oriented Software Architecture – A System of Patronen*, John Wiley & Sons, 1996
- [Coad92] P. Coad, *Object-Oriented Patterns*, Communications of the ACM, Vol. 35, No. 9, September 1992
- [Coplien92] J. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992
- [Florijn95] G. Florijn, N. van Oosterom, *Objectoriëntatie - Klaar voor gebruik?!*, Kluwer Bedrijfswetenschappen, 1995
- [Florijn97] G. Florijn, M. Meijers, P. van Winsen, *Tool support for Object-Oriented Patterns*, ECOOP97
- [Foster97] T. Foster, L. Zhao, *Modelling Transport Objects with Patterns*, Object Expert, juli/augustus 1997
- [Fowler97] M. Fowler, *Analysis Patterns – Reusable Object Models*, Addison-Wesley, 1997
- [Gamma91] E. Gamma, *Objectorientierte Software-Entwicklung am Beispiel von ET++: Klassenbibliothek, Werkzeuge, Design*, Thesis, Universiteit Zürich, 1991
- [Gamma94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994
- [Greefhorst98] D. Greefhorst, M. Maat, ###
- [Hay95] D. C. Hay, *Data Model Patterns: Convention of Thought*, Dorset House Publishing, 1995
- [Jackson95] M. Jackson, *Software Requirements & Specifications – A Lexicon of Practice, Principles, and Prejudices*, Addison-Wesley, 1995
- [Lea96] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996
- [Molin97] P. Molin, L. Ohlsson, *The Points and Deviations Pattern Language of Fire Alarm Systems*, Pattern Languages of Program Design 3, Addison-Wesley, 1997
- [PLOP1] J. Coplien, D. Schmidt (eds), *Pattern Languages of Program Design 1*, Addison-Wesley, 1995
- [PLOP2] J. Vlissides, J. Coplien, N. Kerth (eds), *Pattern Languages of Program Design 2*, Addison-Wesley, 1996

- [PLOP3] R. Martin, D. Riehle, F. Buschmann (eds), *Pattern Languages of Program Design 3*, Addison-Wesley, 1997
- [UML97] Rational Software Corporation, *UML Notation Guide 1.1*, 1997
- [Schmidt95] D. Schmidt, *Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software*, Communications of the ACM, october 1995
- [Soukup94] J. Soukup, *Taming C++: Pattern Classes and Persistence for Large Projects*, Addison-Wesley, 1994
- [Tanenbaum92] A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 1992
- [Troyer93] O. de Troyer, *Object-georiënteerde database-modellen*, Handboek Database Systemen, november 1993
- [Woolf95] B. Woolf, *A Sample Pattern Language - Concatenating with Streams*, The Smalltalk Report, februari, 1995