OO Ontwerp: Design Patterns

Lorenz Verschingel

3 December 2014

Inhoudsopgave

1 Inleiding

1.1 Waarom?

We gebruiken design patterns omdat andere ontwikkelaars via deze weg het probleem eerder al hebben opgelost. Een pattern is geen concrete oplossing, maar eerder een sjabloon waarmee het ontwerpprobleem kan worden opgelost.

1.2 Hoe gebruiken?

De beste manier om design patterns te gebruiken is om ze vanbuiten te leren en vervolgens de plaatsen te leren herkennen waar men ze kan toepassen.

2 Strategy Pattern

2.1 Waarom?

Specificaties blijven constant veranderen.

Hergebruik van code door overerving is in veel gevallen geen zo'n goed idee. Men moet in iedere subklasse gaan kijken of een bepaalde supermethode niet moet vervangen worden via een override.

> Los van wat je maakt of in welke taal je programmeert, de enige constante die je altijd tegenkomt is VERANDERING.

2.2 Ontwerpprincipe

Bepaal de aspecten van je applicatie die variëren en scheid deze van de aspecten die hetzelfde blijven. M.a.w. bevat onze code een aspect dat verandert, dan weten we dat we een gedrag hebben dat eruit gelicht moet worden en moet worden afgezonderd van alle code die niet verandert.

Voordelen:

- Blijft flexibel
- Gedrag aan instanties toekennen die het nodig hebben.
- Het gedrag kan dynamisch verandert worden als in de superklasse een methode voorzien wordt om het gedrag te veranderen.

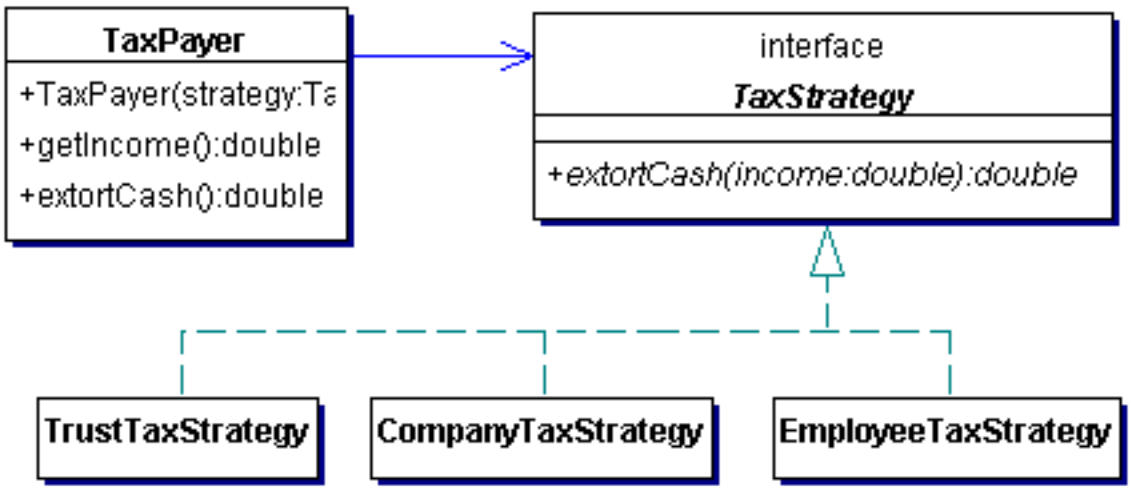
Programmeer naar een interface, niet naar een implementatie. M.a.w. we gaan voor ieder gedrag een interface gebruiken, bv. FlyBehavior en QuackBehavior, en iedere implementatie van een gedrag implementeert één van deze interfaces. M.a.w. we gaan voor ieder gedrag een interface gebruiken en iedere implementatie van een gedrag implementeert één van deze interfaces.

Geef aan compositie de voorkeur boven overerving. In plaats van het gedrag te erven, verkrijgt de klasse zijn gedrag uit het (juiste) gedragsobject waarmee hij is samengesteld. Het creëren van systemen via compositie geeft meer flexibiliteit. We kunnen een familie algoritmen inkapselen. We kunnen tevens het gedrag at runtime veranderen. Compositie wordt in veel design patterns toegepast.

2.3 Wat?

Het Strategy Pattern definieert een familie algoritmen, isoleert ze en maakt ze uitwisselbaar. het strategy pattern maakt het mogelijk om het algoritme los van de client die deze gebruikt, te veranderen.

2.4 Voorbeeld



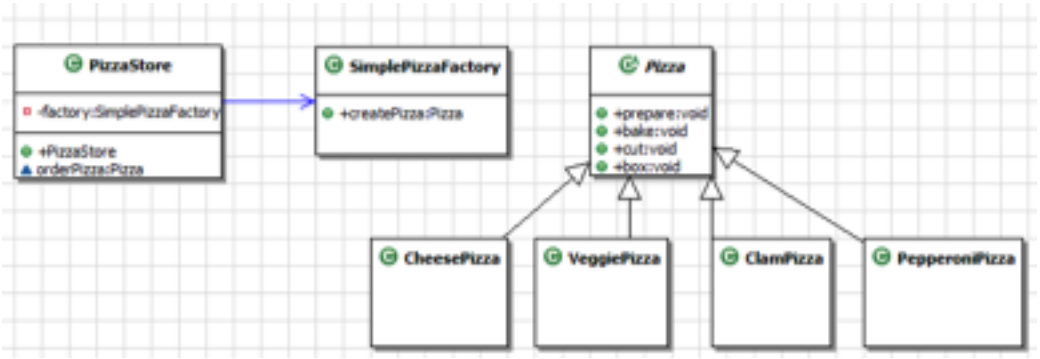
3 Simple Factory Pattern

3.1 Wat?

In objectgeoriënteerde programmeertalen is het factory-ontwerppatroon een manier om objecten te instantiëren zonder exact vast hoeven te leggen van welke klasse deze objecten zullen zijn. In dit patroon wordt een methode (de factory-methode) gebruikt die door subklassen geïmplementeerd kan worden. De klasse van het object die door die methode geïnstantieerd wordt, implementeert een bepaalde interface. Elk van de subklassen kan vervolgens bepalen van welke klasse een object wordt aangemaakt, zolang deze klasse maar die interface implementeert.

Het doel van dit ontwerppatroon is het vereenvoudigen van het onderhoud van het programma. Als er nieuwe subklassen nodig zijn dan hoeft men alleen een nieuwe factory-methode te implementeren.

3.2 Voorbeeld



4 Decorator Pattern

4.1 Wat?

Een decorator is een ontwerppatroon voor objectoriëntatie dat dynamisch extra functionaliteit toevoegt aan een object. Dit is flexibeler dan uitbreiding van functionaliteit door middel van subklassen. Decorator behoort tot de structuurpatronen.

4.2 Het open-gesloten principe

Het doel is dat klassen eenvoudig uitgebreid kunnen worden om nieuw gedrag te incorporeren zonder de bestaande code te wijzigen. Dit kan bereikt worden door ontwerpen te maken die weerstand bieden tegen verandering en flexibel genoeg zijn om nieuwe functionaliteiten op te nemen om aan veranderende eisen tegenmoet te komen.

4.3 Toepassing

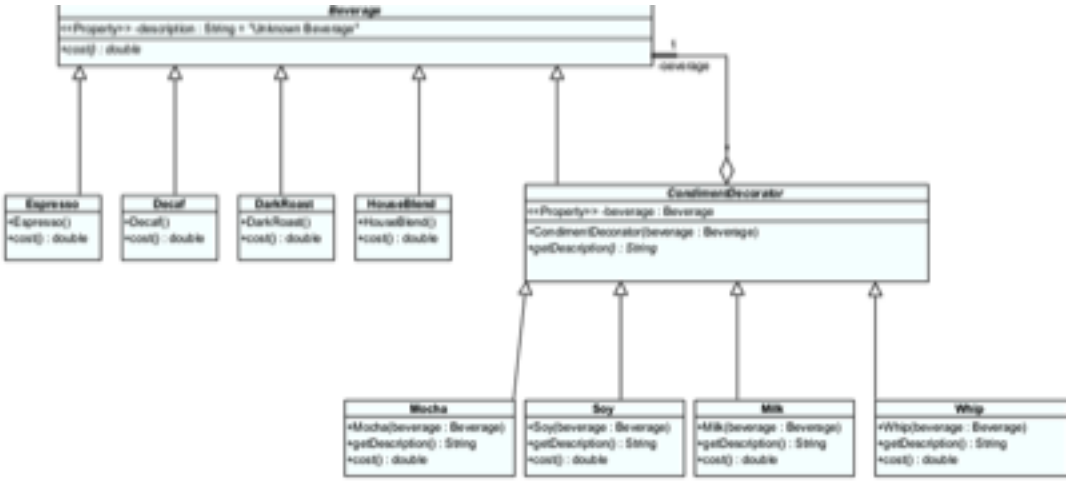
Het patroon is van toepassing in situaties waarbij aan objecten dan weer de ene en dan weer de andere functionaliteit toegevoegd wordt. Indien overerving gebruikt zou worden zou voor elke combinatie van functionaliteiten een subklasse geschreven moeten worden hetgeen al snel inefficiënt wordt in termen van onderhoudbaarheid van broncode.

Het Decorator Pattern kent dynamisch additionele verantwoordelijkheden toe aan een object. Decorators bieden een flexibel alternatief voor het gebruik van subklassen om functionaliteiten uit te breiden.

Een ander voordeel is dat de functionaliteit ter plekke weer kan worden opgeheven.

Grootste nadeel is dat er een nieuw (decorator-)object gemaakt wordt terwijl bij overerving de functionaliteit aan hetzelfde object wordt toegevoegd.

4.4 Voorbeeld



5 Observer Pattern

5.1 Wat?

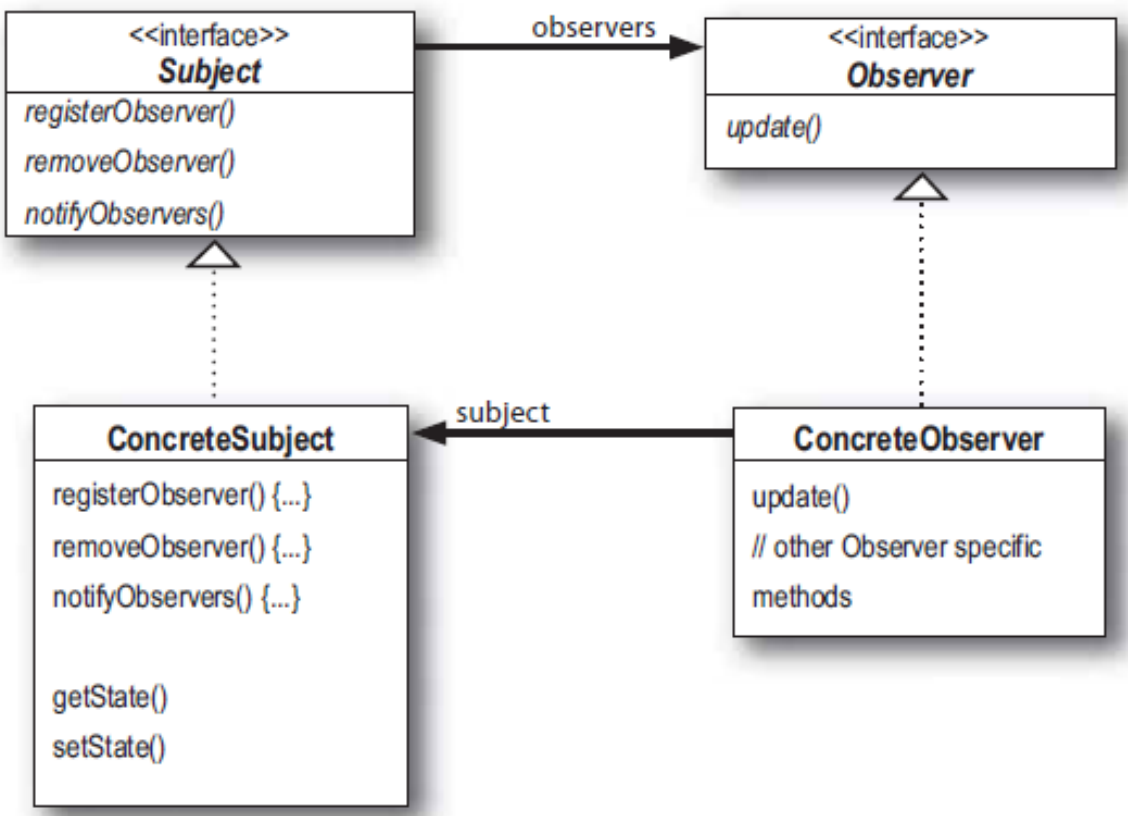
Het Observer Pattern definieert een één-op-veel-relatie tussen objecten, zodanig dat wanneer de toestand van een object verandert, alle afhankelijke objecten worden bericht en automatisch worden geüpdatet.

5.2 Kracht van zwakke koppelingen

- Wanneer twee objecten een zwakke koppeling hebben, dan kunnen ze interactie plegen, maar weten ze weinig over elkaar.
- Het Observer Pattern zorgt voor een ontwerp van objecten waarin subjecten en observers zwak gekoppeld zijn.
  - Het enige wat een subject over een observer weet is dat deze een bepaalde interface implementeert (de Observer-interface).
  - We kunnen op ieder moment nieuwe observers toevoegen.
  - We hoeven het subject nooit te veranderen om nieuwe soorten observers toe te voegen.
  - We kunnen subjecten en observers onafhankelijk van elkaar hergebruiken.
  - Veranderingen in het subject of een observer hebben geen invloed op elkaar.

Zwak gekoppelde ontwerpen maken het mogelijk om flexibele OO-systemen te bouwen die met verandering om kunnen gaan, omdat ze de wederzijdse afhankelijkheid tussen objecten erg klein maken.

5.3 Voorbeeld



6 Façade Pattern

6.1 Wat?

Het Façade Pattern zorgt voor een vereenvoudigde interface naar een verzameling interfaces in een subsysteem. De façade definieert een interface op een hoger niveau zodat het gebruik van het subsysteem vereenvoudigt.

6.2 Het principe van kennisabstractie

Met dit ontwerppatroon wordt er vermeden dat er veel klassen aan elkaar gekoppeld worden, zodat veranderingen in het ene deel van het systeem invloed hebben op andere delen van het systeem. Wanneer er veel afhankelijkheden zijn tussen verschillende klassen dan wordt een systeem ontwikkeld, dat duur is in onderhoud en ook nog eens moeilijk te begrijpen is voor anderen.

6.3 Voordelen

De façade bewerkstelligt losse koppeling tussen verzamelingen functies (of functiecomponenten) en componenten die van die functies gebruik willen maken. Bijgevolg kent een façade de volgende voordelen:

- Het blijft mogelijk om een collectie functies aan te passen of opnieuw te implementeren als de collectie al door andere componenten gebruikt wordt.
- Het is mogelijk om complexiteit bij het correcte gebruik van een collectie functies te verstoppen in of achter de façade.

6.4 Nadelen

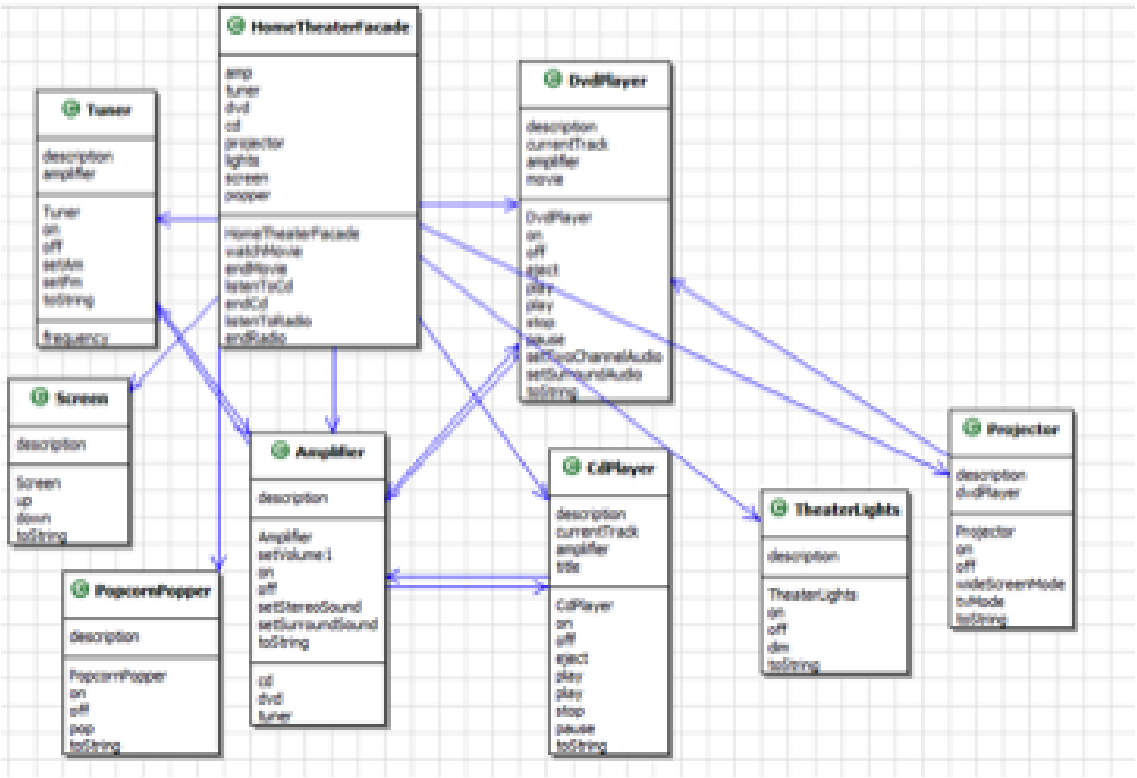
Naast voordelen kent het façadepatroon ook enige nadelen:

- Een façade is van nature een sterk gekoppeld component.
- Een façade reduceert het aantal gebruiksmogelijkheden van een collectie functies van ongebreideld tot "dat wat de façade toestaat". Als gevolg hiervan is het mogelijk dat door het gebruik van de façade niet alle mogelijkheden van een collectie functies aan de buitenwereld aangeboden worden (of kunnen worden).

6.5 Overwegingen

De nadelen van de façade maken dat over het gebruik ervan altijd een afweging plaats moet vinden. Aangezien een façade altijd sterk gekoppeld is en ook nog eens een extra component is in het totale systeem, moet bijvoorbeeld de afweging gemaakt worden of toepassing van de façade de moeite waard is. Een façade voor een klein aantal functiecomponenten plaatsen, kan bijvoorbeeld het systeem onnodig verzwaren. En als blijkt dat een façade het moeilijk maakt om een complexe combinatie van achterliggende functies te gebruiken die wel gewenst is, moet overwogen worden of de façade de moeite is of wellicht ook om het mogelijk te maken voor andere componenten om (gedeeltelijk) om de façade heen te gaan.

6.6 Voorbeeld



7 State pattern

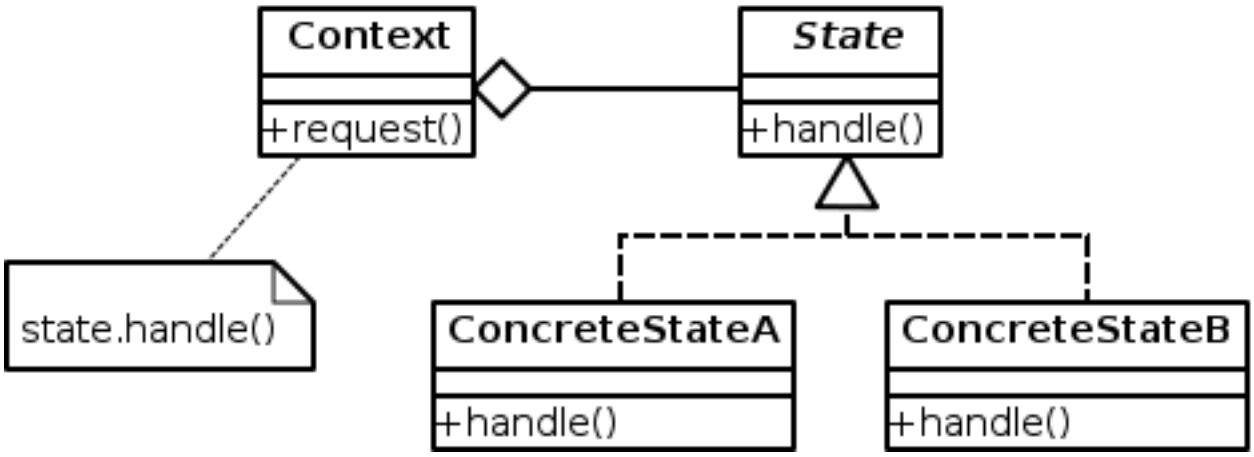
7.1 Wat?

Het State Pattern maakt het voor een object mogelijk zijn gedrag te veranderen wanneer zijn interne toestand verandert. Het object lijkt van klasse te veranderen.

Het State Pattern wordt gebruikt om verschillende toestanden van een object op een nette manier in te kapselen.

Het patroon bestaat uit een tweetal klassen: de 'context' en de 'toestand' (state). De eerste handelt verzoeken af en geeft deze door aan de tweede. De verschillende interne toestanden worden geïmplementeerd als subklassen van de state-klasse. Iedere subklasse kan anders reageren op verzoeken, zodat we gedrag verkrijgen dat kan afhangen van de interne toestand van een object zonder allerlei globale variabelen en een batterij if-statements.

7.2 Voorbeeld



7.3 Verschil tussen state en strategy pattern

Het state pattern definiëert wat het object is. Het encapsuleert staat-afhankelijk gedrag.

Bijvoorbeeld:

Een kauwgomautomaat: afhankelijk of er een munt in de automaat zit of niet zal de machine anders reageren.

Het strategy pattern bepaalt hoe een object een bepaalde taak uitvoert, het encapsuleert een algoritme.

Bijvoorbeeld:

De sort algoritmes in java. Via een parameter kan meegegeven worden hoe de collection gesorteerd moet worden. Onafhankelijk van welke parameter je meegeeft blijft het doel hetzelfde: de collection moet gesorteerd worden.