Begeleidend verslag Design Patterns

Klaus, Tobias de Bildt, Kristiaan Cramer, Ouassim Rafiq, Eliam Traas

Ontwerp

Ons project omvat een model van een auto-dealership systeem waarin we verschillende design patterns hebben toegepast met het oog op schaalbaarheid, onderhoudbaarheid en flexibiliteit. Het systeem simuleert een client (`app.java`) die via een dealership-interface auto's bestelt bij verschillende fabrieken. Deze auto's bevatten allemaal een motor en een remsysteem en bovendien is de auto uit te breiden met upgrades. In ons ontwerp hebben we de Factory-, Decorator-, Bridge- en Facade-pattern toegepast, om ons model te ontwerpen in lijn met de software eisen die gesteld waren.

Toelichting

Voordat we beginnen met het verdedigen van onze keuzes voor welke patronen er gebruikt zijn, lichten we eerst een aantal woorden toe die we zullen gebruiken voor onze verdediging. Voor dit vak hebben we het boek “Design patterns explained” van Alan Shalloway gelezen en daarin wordt uitgelicht dat ‘goede’ software “high in coupling” en “low in cohesion” dient te zijn. Hiermee wordt bedoeld dat klassen in een goed ontworpen systeem een hoge interne samenhang (cohesie) hebben, terwijl afhankelijkheden tussen klassen zo beperkt mogelijk zijn (lage koppeling). Hoge cohesie betekent dat functies binnen een klasse nauw met elkaar verbonden zijn en samen één duidelijke verantwoordelijkheid hebben, wat het systeem overzichtelijk en makkelijk te onderhouden maakt. Lage koppeling zorgt ervoor dat klassen weinig afhankelijk zijn van andere klassen, waardoor wijzigingen minder snel invloed hebben op andere delen van de code.

Factory pattern

In ons ontwerp maken we gebruik van de factory pattern om de verschillende typen autos aan te maken. We hebben een abstracte fabriek genaamd `AutoFabriek` en deze wordt afgeleid naar de verschillende autofabrieken voor de verschillende auto’s. Door het maken van afzonderlijke fabrieken, zoals `BestelBusAutoFabriek`, `PersonenAutoFabriek` en `SportAutoFabriek`, wordt de verantwoordelijkheid voor het creëren van specifieke typen auto’s overgedragen aan dedicated factory-klassen. Dit leidt tot een hogere cohesie omdat de fabrieken enkel verantwoordelijk zijn voor het produceren van een specifiek type auto. Als alle verschillende auto typen (die op verschillende manieren worden opgebouwd) allemaal uit dezelfde factory zouden komen zou dit lijden tot low cohesion omdat er allemaal ongerelateerde zaken door dezelfde klasse worden afgehandeld.

Bovendien behalen we met de factory pattern dat de code voor het maken van verschillende objecten gescheiden is van de code voor het gebruiken van een object. Hierdoor kan de

code om een object aan te maken veranderd worden, zonder dat de code waarin het object gebruikt wordt, hoeft te worden aangepast.
Daarbij, omdat verschillende soorten auto's anders gebouwd kunnen worden, geeft de Autofabriek een universeel interface voor elk type auto. Hierdoor wordt het makkelijker om tussen verschillende soorten auto's te kiezen door de dealer.

Decorator pattern

We hebben het Decorator Pattern geïmplementeerd in de klasse `AutoDecorator.java`. Dit patroon maakt het mogelijk om extra "decoraties" of upgrades aan een auto-object toe te voegen zonder de code onnodig complex te maken. Zonder dit patroon zouden we namelijk te maken krijgen met een "state explosion" door de vele mogelijke combinaties van opties. De huidige decoraties zijn een parkeersensor, een Bose-geluidssysteem en verwarmde stoelen. Deze upgrades zijn optioneel en bieden extra functionaliteit zonder essentieel te zijn voor het functioneren van de auto. Dit is ook zichtbaar in de aggregatie-relatie in het diagram: de auto is niet afhankelijk van de decoraties, maar kan er wel flexibel mee worden uitgebreid.

De opzet is schaalbaar; er kunnen onbeperkt decoraties worden toegevoegd, elk met eigen prijs en functie. Wanneer er een nieuwe decoratie op de markt komt, kan deze eenvoudig worden afgeleid van de `AutoDecorator.java`, zodat elke auto deze nieuwe feature kan toevoegen.

Bridge pattern

De keuze voor het Bridge Pattern maakt het mogelijk om onderdelen zoals motoren (`Motor.java`) en remsystemen (`RemSysteem.java`) modulair en onafhankelijk aan `Auto`-instanties toe te voegen. Door deze loskoppeling van de implementatie van motoren en remsystemen van de `Auto`-klasse realiseren we *low coupling* tussen de componenten, wat resulteert in een flexibel en onderhoudbaar ontwerp. Het Bridge Pattern zorgt ervoor dat motoren en remsystemen als afzonderlijke modules met *high cohesion* kunnen worden ontwikkeld en beheerd, zonder afhankelijkheid van de `Auto`-klasse. Hierdoor kunnen deze componenten eenvoudig worden hergebruikt in verschillende `Auto`-instanties, zonder dat aanpassingen aan de `Auto`-klasse zelf nodig zijn. Bovendien voorkomt het Bridge Pattern een explosie van subklassen voor elke mogelijke combinatie van onderdelen, wat het systeem overzichtelijk en schaalbaar houdt.

Facade

In ons project hebben we het Facade Pattern geïmplementeerd in de `Dealership`-klasse, waarmee het autodealership fungeert als een eenvoudige interface voor de gebruiker. Met het Facade Pattern bieden we een overzichtelijk en gebruiksvriendelijk systeem, waarbij de gebruiker geen directe interactie heeft met de complexe logica achter de verschillende autofabrieken (`SportAutoFabriek`, `PersonenAutoFabriek`, `BestelBusAutoFabriek`). In plaats daarvan kunnen gebruikers simpelweg via de `Dealership`-klasse een auto bestellen of uit de huidige voorraad kiezen, zonder gedetailleerde kennis over auto-onderdelen of fabriekstypen nodig te hebben.

Het Facade Pattern maakt de interactie laagdrempelig, doordat de gebruiker kan kiezen uit voorgedefinieerde opties die de dealer heeft samengesteld, zoals met `getStock()` of `getOrderPresets()`. Hierdoor worden complexe beslissingen gereduceerd tot eenvoudige keuzes, wat vooral nuttig is voor klanten met beperkte kennis van auto's. Dit patroon voorkomt ook *tight coupling* door de details van de verschillende fabrieken af te schermen en zorgt zo voor een eenvoudiger, onderhoudbaar systeem. De gebruiker werkt uitsluitend met de `Dealership`-klasse, wat het geheel overzichtelijk houdt en het systeem flexibel uitbreidbaar maakt zonder extra complexiteit voor de gebruiker.