Rick Aerts, Tom Eisermann, Harm van Veen

Presentatie

Ondezoek naar het gebruik van design patterns binnen software engineering

Design Patterns

Proxy Design Pattern

[Design Patterns 2](#_Toc57051641)

[Wat zijn software design patterns? 2](#_Toc57051642)

[Hoe hebben design patterns hun intrede gedaan in het vak van software engineering? 2](#_Toc57051643)

[Welke software design pattern categorieën onderkennen we? 3](#_Toc57051644)

[Wat zijn de voordelen van design patterns? 4](#_Toc57051645)

[Wat zijn de nadelen van design patterns? 4](#_Toc57051646)

[Proxy Structural Design Pattern 5](#_Toc57051647)

[Virtual Proxy 5](#_Toc57051648)

[… Proxy 5](#_Toc57051649)

[… Proxy 5](#_Toc57051650)

# Design Patterns

## Wat zijn software design patterns?

* Gestandaardiseerde oplossingen voor veel voorkomende problemen in software ontwerp.
* **WEL** een blauwdruk, template of mal. **GEEN** kant en klare code.
* **Opbouw:**
  + Intent  
    Een korte samenvatting van zowel het probleem, als de oplossing
    - Algemene termen
  + Motivation  
    Een uitgebreidere uitleg van het probleem, en hoe het ontwerp daar een oplossing voor bied. Naast de oplossing biedt de design pattern ook waarom het specifiek design pattern een oplossing is voor het specifiek probleem.
  + Structure  
    Een gedetaileerdere omschrijving van de verschillende onderdelen/classes binnen het patroon, en hoe deze aan elkaar gerelateerd zijn.
    - Class diagram
    - Pseudocode
    - Entity-relation diagram
  + Code Example  
    Een uitwerking van het patroon in code om een concreet voorbeeld te geven van een probleem en hoe dit opgelost is.

## Hoe hebben design patterns hun intrede gedaan in het vak van software engineering?

Een vroege poging tot definiëren van design patterns is gedaan door Christopher Alexander in A Pattern Language: Towns, Buildings, Construction. Wat een boek over ontwerpen van steden is. Een van de eerste publicaties van gebruik in context software engineering: Design Patterns: Elements of Reusable Object-Oriented Software (1994)

## Welke software design pattern categorieën onderkennen we?

We maken bij de software design patterns onderscheid tussen drie verschillende categorieën, namelijk: creational design patterns, structural design patterns en behavioral design patterns. Iedere categorie beschrijft een ander type probleem, dat de design pattern probeert op te lossen.

Zo richt een creational design pattern zich op het aanmaken van objecten, op een manier waarop software flexibel en beheerbaar blijft. Zo proberen patronen uit deze categorie ervoor te zorgen dat er zo weinig mogelijk afhankelijkheden ontstaan, door zo weinig mogelijk instantiaties van objecten plaats te laten vinden in operationele klassen.

Een structural design pattern beschijven op welke manieren klassen en objecten grotere structuren kunnen vormen, zonder daarbij efficiëntie en flexibiliteit te verliezen. Het patroon dat we later gaan bespreken is een voorbeeld van dit type design patterns.

Een behavioral design pattern richt zich op algoritmes en de manier waarop een verdeling van verantwoordelijkheden tussen objecten tot stand kan komen.

## Wat zijn de voordelen van design patterns?

* Design patterns zijn beproefde oplossingen voor veel voorkomende problemen
  + Het wiel hoeft niet telkens opnieuw ontworpen te worden.
  + Minder kans op onverwachte verassingen.
* Beter Programmeren.
* Design patterns bieden een gedeelde vocabulaire voor communicatie over ontwerp

## Wat zijn de nadelen van design patterns?

* Bij misbruik van design patterns kunnen programma’s onnodig complex worden gemaakt.
* Niet alle programmeertalen ondersteunen de elementen, die nodig zijn om design patterns nodig te maken, daarnaast kunnen design patterns behoefte voor vermogen verhogen.

# Proxy Structural Design Pattern

Zoals eerder in de presentatie aangegeven bestaat een Design pattern uit 4 onderdelen:

* Intent
* Motivation
* Structure
* Example

De eerste drie zal ik hier bespreken, en daarna hebben we nog enkele voordelen van hoe het proxy pattern in de practijk nu doorgevoerd kan worden.

## Intent

**Een proxy behoort tot de groep structurele design patterns. Kort gezegd, bied het de mogelijkheid om een plaatsvervanger voor een object te zetten die het contact met het gekozen object beheert. Dit maakt het mogelijk om extra functionaliteit toe te voegen die voor of na het aanroepen van het oorspronkelijke object plaats vindt.**

## Motivation

### Probleem:

Wanneer gebruik je nu een proxy pattern? In de praktijk zijn er verschillende problemen die door middel van een proxy patroon opgelost kunnen worden. We zullen later op wat details in gaan, maar enkele voorbeelden zijn bijvoorbeeld:

* Je hebt een object wat veel resources vraagt, maar wat maar sporadisch gebruikt wordt.
* Je wilt de toegang tot bepaalde objecten op basis van bepaalde criteria beperken.
* Je wilt een object gebruiken wat niet lokaal beschikbaar is.
* Je wilt een overzicht van het gebruik van een object registreren
* Je wilt een cache van de resultaten van verzoeken aan het object (tijdelijk) bewaren

In sommige situaties kun of wil je het service object zelf aanpassen, maar dit is niet altijd een optie. Wellicht zijn het objecten uit een externe resource, of zijn er verschillende cliënten waar verschillende wensen voor zijn.

### Oplossing:

Een Proxy pattern kan door de zelfde interface te implementeren als het daadwerkelijke service object als tussenpersoon fungeren voor een client. Op deze manier hoeft in veel gevallen de client, noch het service object aangepast te worden, en is het toch mogelijk om de gewenste extra functionaliteit te implementeren.

## Structure

### Structure of the Proxy design patternOpbouw van een proxy design pattern

Een proxy pattern bestaat uit 3 onderdelen:

#### Service Object

Het service object levert een vorm van businesslogic.

#### Service Interface

De service interface wordt door het service object geïmplementeerd zodat een client gebruik kan maken van de service.

#### Proxy Object

Het proxy object werkt door een de zelfde Service interface te implementeren als het service object. Hierdoor kan het net als het service object zelf door de client geïmplementeerd worden.

In tegenstelling tot het serviceobject heeft het echter niet de logica om zelfstandig de geïmplementeerde methodes uit de interface uit te voeren. In plaats daarvan heeft het een composiet relatie met het daadwerkelijke service object, en roept hij de businesslogic van het oorspronkelijke service object aan als onderdeel van zijn eigen methode. Extra functionaliteit van de proxy service zelf kan vervolgens daarvoor of daarna toegepast worden.

## Voordelen & Nadelen

Het gebruik van Design Patterns heeft net als alle andere design patterns voor en nadelen.

Voordelen

Zowel het proxy object en het server object kunnen gewijzigd worden zonder dat dit effect heeft voor de klasse, die ze gebruikt.

Het proxy object blijf beschikbaar zelfs als de het server object dat niet is

Door deze pattern te gebruiken kunnen functionaliteiten toegevoegd te worden zonder de eerder aangemaakte klassen te wijzigen en de proxy verschillende functionaliteiten toevoegen waardoor dit patroon zeer flexibel is

De reactie van het service object kan trager worden doordat het dataverkeer via de proxy pattern loopt in plaats van rechtstreeks naar het object

Door toevoeging van extra klassen kan deze design pattern complex overkomen

### Vergelijking met andere design patterns

Het proxy object heeft een zekere mate van overlap met sommige andere design patterns. Toch verschillen ze in intent en functionaliteit

#### Adapter vs Proxy

Zowel een adapter als een proxy fungeren als plaatsvervanger voor een service object. Maar waar een adapter een service object met één interface beschikbaar stelt voor een client onder een andere interface, bied een proxy *extra* functionaliteit aan onder de zelfde interface.

#### Facade vs Proxy

Zowel een façade als een proxy een buffer vormen tussen een client en een service object en beide het service object initialiseren, implementeert de proxy de zelfde interface als het service object, en kan dus als plaatsvervanger fungeren.

#### Decorator vs Proxy

Zowel een decorator als een proxy implementeren de zelfde interface als het service object en voegen op deze manier extra functionaliteit aan een bestaand service object toe. Maar waar een proxy over het algemeen de levenscyclus van het service object zelf beheert, wordt dit bij een decorator altijd door de client beheerd.