OO-design

Course OOSE-OOAD   
2020-2021 Semester 2  
Robert Boudewijn  
www.robertboudewijn.nl

Inhoudsopgave

[1. GRASP 2](#_Toc67423791)

[1.1. Controller 2](#_Toc67423792)

[1.2. Creator 3](#_Toc67423793)

[1.3. Information Expert 4](#_Toc67423794)

[1.4. Low Coupling 4](#_Toc67423795)

[1.5. High Cohesion 5](#_Toc67423796)

[2. SOLID 5](#_Toc67423797)

[2.1. Single Responsibility Principle (SRP) 6](#_Toc67423798)

[2.2. Open/Closed Principle (OCP) 6](#_Toc67423799)

[2.3. Dependency Inversion Principle (DIP) 7](#_Toc67423800)

[2.3.1. Tight Coupling 7](#_Toc67423801)

[2.3.2. Dependency inversion 7](#_Toc67423802)

[2.3.3. Inversion vs Injection 8](#_Toc67423803)

[3. STUPID 8](#_Toc67423804)

[4. GoF Design Patterns 8](#_Toc67423805)

[4.1. Observer 9](#_Toc67423806)

[4.2. Strategy 10](#_Toc67423807)

[4.3. State 11](#_Toc67423808)

[4.4. Adapter 12](#_Toc67423809)

[4.5. Factory Method 14](#_Toc67423810)

[Bibliografie 16](#_Toc67423811)

[Figuren 16](#_Toc67423812)

# GRASP

GRASP staat voor ‘Generalized Responsibility Assignment Software Patterns/Principles’. Het bestaat uit 9 richtlijnen die kunnen worden gebruikt om in objectgeoriënteerde systemen verantwoordelijkheden toe te kennen aan klassen of objecten. Elke richtlijn behandelt een combinatie van een veelvoorkomend probleem en een algemeen toepasbare oplossing voor dat probleem met betrekking tot het ontwerp van softwaresystemen. (Wikipedia-bijdragers, 2016)

Voor de laatste letter van acroniem GRASP wordt zowel Patterns (patronen) als Principles (richtlijnen) gebruikt. Een pattern is een beschrijving van een probleem en een oplossing, dat kan worden toegepast in nieuwe contexten (M. Koolwaaij, 2020). Lees hierover meer in hoofdstuk 4.

De 9 richtlijnen zijn:

1. Controller
2. Creator
3. Information Expert
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

In OOSE OOAD word er gefocust op de eerste 5.

## Controller

De controller is verantwoordelijk voor het behandelen van de verzoeken van actoren. De controller is de tussenpersoon tussen uw gebruiker die op "Verzenden" klikt en uw back-end om dat te laten gebeuren (R. Fadatare, 2018).

**Voorbeeld van probleem:**  
Welke (niet-UI) class moet de system events afhandelen?

**Oplossing:**

Geef de responsibility aan een class die:

1. Het totale systeem representeert
   1. Façade controller
2. Of de use case representeert
   1. Use-case controller

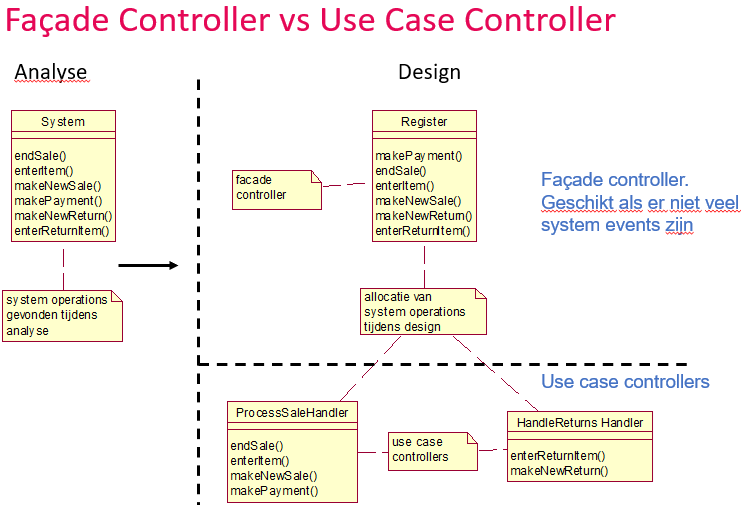
**Factoren die de keuze beïnvloeden:**

1. De mate van koppeling of cohesie

Figuur 1; Hoort het Syteem al deze events op te vangen of is dat dus een taak van een controller?

1. Hoeveel verantwoordelijkheden krijgt de façade controller? Misschien zijn nog meer controllers nodig, en moet de façade controller delegeren

**Een Controller is géén user-interface object**,system events mogen daardoor niet worden afgehandeld door de presentatielaag (UI) (M. Koolwaaij, 2020).



Figuur 2; Een Façade controller is in dit geval een algemene super controller. Dit is handig wanneer je een snel een applicatie wilt opzetten. Echter wanneer het (te) groot (-er) wordt is het handig om de meerdere kleine controllers te maken.

## Creator

Creator is een GRASP-patroon dat helpt om te beslissen welke klasse verantwoordelijk moet zijn voor het maken van een nieuwe instantie van een klasse. Het maken van objecten is een belangrijk proces en het is handig om een principe te hebben om te beslissen wie een instantie van een klasse moet maken. (R. Fadatare, 2018)

**Voorbeeld van een probleem:**

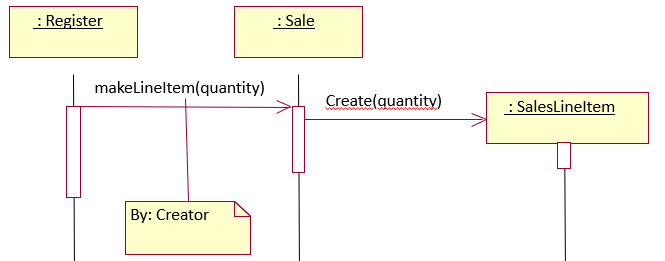
Welk object mag een instantie van een class A aanmaken?

**Oplossing:**

Geef die verantwoordelijkheid aan een class B waarvoor geldt:

1. B ‘bevat' A
2. B legt A vast
3. B gebruikt A het meest
4. B heeft de initialisatie-data voor A

(M. Koolwaaij, 2020)



Figuur 3; In dit geval maakt Sale gebruik van een SalesLineItem. Dus hebben we temaken met optie b. B legt A vast

## Information Expert

information-expert is een principe dat wordt gebruikt om te bepalen waar verantwoordelijkheden, zoals methoden, berekende velden, enzovoort, moeten worden gedelegeerd. (R. Fadatare, 2018)

**Voorbeeld van probleem:**

Ik moet responsibilities aan classes toewijzen welk algemeen principe kan ik hierbij gebruiken?

**Oplossing:**

Wijs een responsibility toe aan die class die de vereiste informatie heeft om aan die responsibility te voldoen.

**Voordeel van Expert:**

Information hiding( een principe dat stelt dat interne details van de implementatie van een klasse verborgen moeten blijven voor andere klassen) wordt dan in stand gehouden. (M. Koolwaaij, 2020)



Figuur 4; Hier is goed te zien dat Information Expert is toegepast. Omdat, een total behoort bij een Sale, een SubTotal bij een SalesLineItem en een Prijs bij een Product(Specification)

## Low Coupling

Hoe afhankelijk is een component van andere compenten (Classes, subsystems, systems)?

**Probleem:** Hoe wordt bevorderd dat componenten minder van elkaar afhankelijk zijn?

**Oplossing**: Deel responsibilities zó uit dat *koppeling* *zwak* blijft, dit:

1. Verhoogt de herbruikbaarheid.
2. Een afzonderlijke component is beter te begrijpen.
3. Minder last van wijzigingen in andere componenten.

**Type koppelingen in java of andere Object georiënteerde talen:**

1. Een object heeft een attribuut dat verwijst naar een ander object.
2. Een object roept een method aan van een ander object.
3. Een object heeft een method die verwijst naar een ander object.
4. Subclasses in een class hiërarchie.
5. Een object implementeert een interface.

Maar: Doel van OOAD is te komen tot een systeem van objecten die via messages met elkaar communiceren. Dus een (bescheiden) vorm van koppeling is normaal en noodzakelijk om goed samenwerkende objecten te krijgen. Een applicatie met een zeer zwakke koppeling kan leiden tot een systeem met:

1. Enkele ‘opgeblazen’ complexe objecten die al het werk doen: Bloated objects
2. Veel passieve objecten die dienst doen als data repositories

## High Cohesion

Hoge cohesie is een evaluatief patroon dat probeert objecten op de juiste manier gefocust, beheersbaar en begrijpelijk te houden. Hoge cohesie wordt over het algemeen gebruikt ter ondersteuning van low coupling. (Fadatare, 2018)

De term cohesie wordt gebruikt om aan te geven in welke mate een klas één enkele, gerichte verantwoordelijkheid heeft. Cohesie is een maatstaf voor hoe de methoden van een klasse of module zinvol en sterk verwant zijn en hoe gefocust ze zijn bij het verschaffen van een welomschreven doel aan het systeem. (Fadatare, 2018)

**Probleem:** Hoe wordt bevorderd dat een component niet te complex wordt?

**Oplossing:** Deel verantwoordelijkheden zó uit dat cohesie sterk blijft. De voordelen hiervan zijn:

1. Dit verhoogt de herbruikbaarheid
2. Een afzonderlijke component is beter te begrijpen
3. Minder last van wijzigingen in andere componenten
4. En vaak is low coupling het gevolg

Class met sterke cohesie:

1. Heeft een relatief klein aantal methoden die functioneel sterk samenhangen
2. Doet niet te veel werk
3. Werkt samen met andere objecten als de taak te zwaar is (Delegeert)

# SOLID

Solid zijn een aantal standaard regels die je tijdens de software ontwikkeling kan gebruiken. Je hoeft je er niet aan te houden en er kunnen goede redenen zijn om er vanaf te wijken maar de volgende redenen zijn nooit goed:

1. Ik begrijp het niet.
2. Ik weet niet hoe.
3. Ik vind dit mooier
4. Het moest van [x]

Solid staat voor:

1. Single Responsibility Principle (SRP)
2. Open-Closed Principle (OCP)
3. Liskov Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

En in OOAD zullen we alleen a, b en e behandelen

## Single Responsibility Principle (SRP)

S.R.P. in het kort. Dit principe vertelt ons:

“A class should have one and only one reason to change, meaning that a class should have only one job.”

Oftewel, een class moet maar één doel hebben en daarmee dus maar ook één reden hebben om te worden aangepast. (Karsten, onbekend)

**Waarom is SRP belangrijk:**

1. We willen dat het gemakkelijk is om code te hergebruiken
2. Grote klassen zijn moeilijker te veranderen
3. Grote klassen zijn moeilijker te lezen
4. Kleinere klassen en kleinere methoden geven je meer flexibiliteit en je hoeft niet veel extra code te schrijven om dit te doen!

**SRP is ook wel bekend als:**

* Don’t Repeat Yourself (DRY)
* Once and only once
* Single point of truth

## Open/Closed Principle (OCP)

OCP staat voor:

“Objects or entities should be open for extension, but closed for modification.”

Simpel gezegd: een class moet makkelijk uit te breiden zijn, zonder de gehele classe aan te passen.

**Om te voldoen aan OCP moeten je compenten (Classes, subsystems, systems):**

Open zijn voor verlenging:

1. Het gedrag van de module kan worden uitgebreid
2. We zijn in staat om te veranderen wat de module doet

Gesloten voor wijziging zijn:

1. Uitbreidingsgedrag resulteert niet in wijzigingen in de broncode, het binaire bestand of de code van de module

Van abstractheid gebruik maken zoals:

1. Interfaces
2. Abstracte klassen

**Waarom OCP belangrijk is:**

1. Iedere keer als je code wijzigt, heb je de mogelijkheid om de code kapot te maken.
2. Soms kun je niet van libraries veranderen (bijvoorbeeld code die niet van jou is).
3. Mogelijk moet de code op veel verschillende plaatsen worden gewijzigd om ondersteuning voor een bepaald type situatie toe te voegen.

**Hoe je OCP toepast:**

1. Maak gebruik van if of switch als de hoeveelheid van cases waarschijnlijk niet gaat veranderen.
2. Maak gebruik van een strategy pattern(zie hoofdstuk 4.2) als de hoeveelheid waarschijnlijk wel veranderd.
3. Gebruik altijd je gezonde verstand. (duhh)

## Dependency Inversion Principle (DIP)

**“**“High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.” (1996, Robert C. Martin)**”** (M. Koolwaaij, 2020)

### 2.3.1. Tight Coupling

Een klasse is “Tightly coupled” aan een andere klasse als deze aan die klasse is gekoppeld en ervan afhankelijk is. Strak gekoppelde klassen kunnen niet onafhankelijk van elkaar werken of evolueren. Daardoor zijn deze klassen slecht voor onderhoud. Een verandering in één klasse zou een golf van vereiste veranderingen in nauw gekoppelde klassen kunnen veroorzaken.

### 2.3.2. Dependency inversion

Dependency inversion verwijst naar een omkering van de afhankelijkheidsrelatie tussen componenten op hoog en laag niveau

DIP pleit ervoor om de koppeling in het algemeen te verminderen en voor de resterende koppeling om low-level componenten te laten afhangen van high-level componenten door middel van abstracties.

### 2.3.3. Inversion vs Injection

1. Dependency injection != dependency inversion

 Beide maken de code beter testbaar

**Dependency injection:** I.p.v. een object zelf aan te maken krijgen we deze mee als parameter

Niet:

1. IBier bier = new Bier();

Maar:

1. public.Kratje (IBier bier) {
2. this.bier = bier;
3. }

# STUPID

Nog een kleine toevoeging uit Thema 8. Vermeid STUPID code:

* (S)ingleton
* (T)ight coupling
* (U)ntestability
* (P)remature optimization
* (I)ndiscriptive naming
* (D)uplication

# GoF Design Patterns

Een pattern is een beschrijving van een probleem en een oplossing, dat kan worden toegepast in nieuwe contexten (M. Koolwaaij, 2020). Patterns hebben een naam, probleem, oplossing en voorbeelden en/of discussies (M. Koolwaaij, 2020).

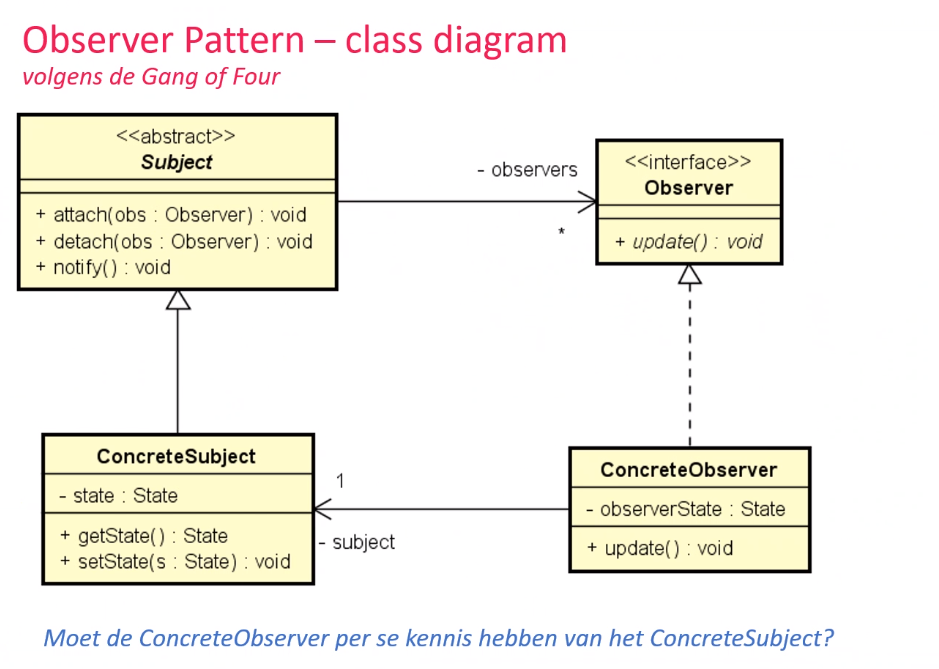
Een voorbeeld is:

**Context:**  Een man wil zijn auto in de garage zetten.  
**Problem:**  De garage is heel smal.  
**Solution:** https://www.youtube.com/watch?v=GfQlWCufkRo&ab\_channel=R0FLMAOWPIMP

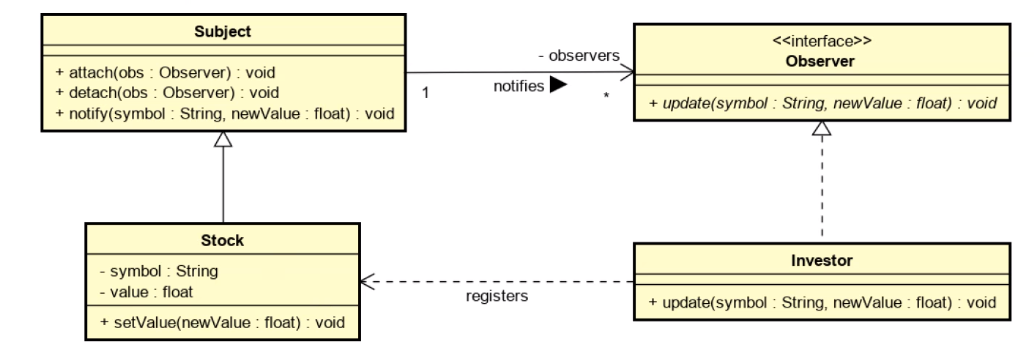
## Observer

**Context:** De toestand van objecten is afhankelijk van een ander object.

**Problem:** Hoe kunnen veranderingen in de toestand van een object gemakkelijk overgebracht worden naar de afhankelijke objecten.

**Solution:** Definieer een one-to-many depandacy tussen de objecten zodat wanneer het ene object van toestand veranderd, alle afhankelijke objecten op de hoogte gebracht en ge-update worden. 

Figuur 5; Een voorbeeld van een observer pattern.

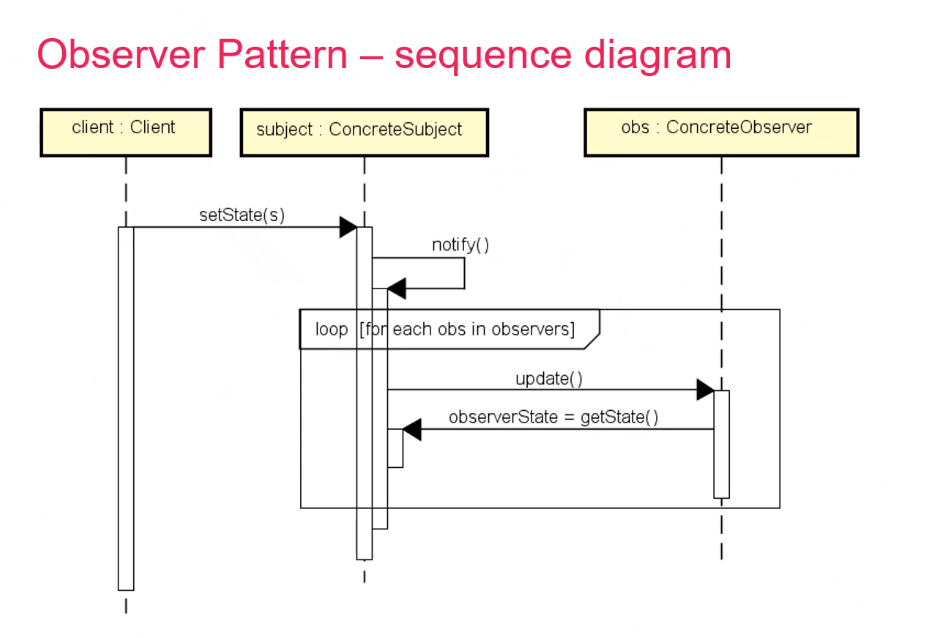


Figuur 6; een uitgewerkt toets voorbeeld

**Belangrijk** hierbij is wel: het subject weet niets bijzonders van zijn observers.

Low coupling is wenselijk omdat:

1. Observers abonneren (subscribe) zichzelf op notificatie events van het subject
2. Wanneer de observer op de hoogte wordt gebracht (notification) zal het zichzelf updaten
3. Het enige dat het subject weet van de andere objecten is dat die de rol van observer hebben.
4. Publish-Subscribe 1:\* relatie



Figuur 7; Sequence diagram van het observer pattern

## Strategy

**Context:** Een object kan een operatie volgens verschillende strategiën uitvoeren.

**Problem:**

1. Hoe kunnen we makkelijk de huidige strategie van een object veranderen?
2. Hoe kunnen we makkelijk een nieuwe strategy toevoegen?

**Solution:**

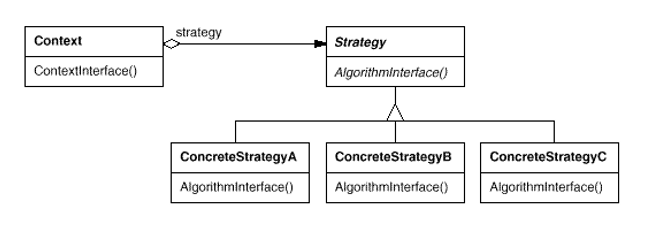
1. Creëer voor elke strategie een strategy class, die een gemeenschappelijk interface implementeert.
2. Delegeer strategy-afhankelijke operaties van het object naar het betreffende strategy object.

**Intentie:**

Definiëer een familie van algoritmen, encapsulate ze en maak ze uitwisselbaar. Strategy Pattern zorgt ervoor dat het algoritme onafhankelijk kan variëren van de clients (objecten) die het gebruiken.

**Gebruik het Strategy Pattern wanneer:**

1. Vele gerelateerde klassen die alleen qua gedrag verschillen. Strategieën geven de mogelijkheid om klassen te configureren met 1 van de vele mogelijke gedragingen.
2. Je verschillende varianten van een algoritme nodig hebt
3. Een algoritme gebruikt data waarvan je wilt dat de client die niet weet. Gebruik het pattern om te voorkomen dat complex, algoritme specifieke data structuren bekend worden gemaakt.
4. Een klasse heeft verschillende soorten gedrag welke als conditionele statements in zijn methoden voorkomen. In plaats van al die condities op te nemen geef elke gerelateerde conditionele branche zijn eigen Strategy klasse



Figuur 8; Een Class diagram die het pattern extra toelicht.

## State

Een state pattern heeft te maken met een state machine. Deze hebben we gehad in de propedeuse met EPD.

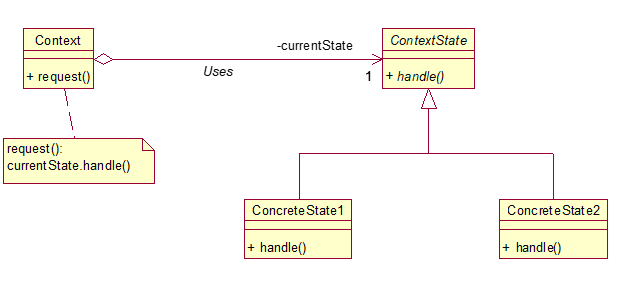
**Context:**  Het gedag van een object is afhankelijk van zijn huidige status.

**Problem:**

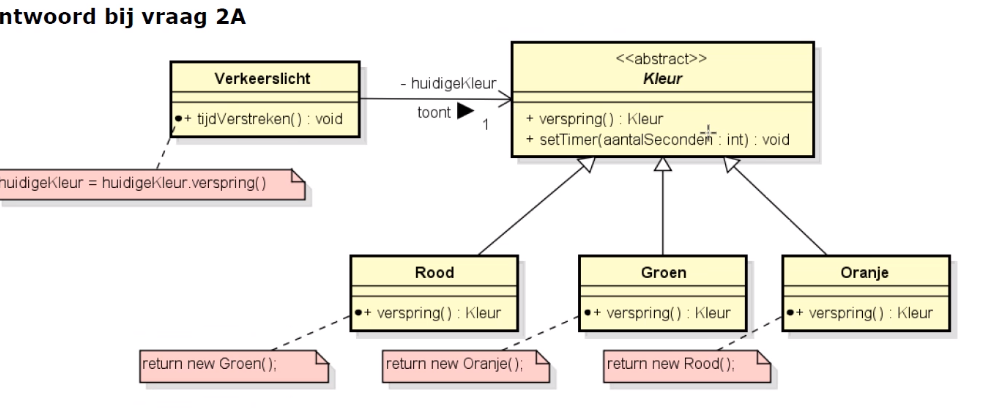
1. De methode van het object (klasse) bevatten veel voorwaardelijke logica.
2. Is er een alternatief voor de voorwaardelijke logica?

**Solution:**

1. Maak voor elke status(toestand) van het object een State class, dat een gemeenschappelijke interface implementeert.
2. Status bewerkingen van het context object overdragen (delegeren) aan het huidige status object.



Figuur 9; Een class diagram van een state machine in bijvoorbeeld java.



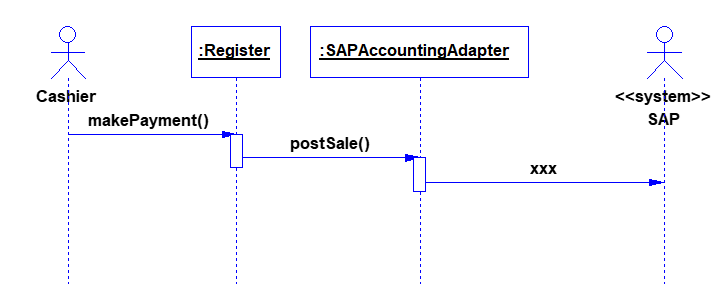
Figuur 10; Concreet voorbeeld van een state machine

## Adapter

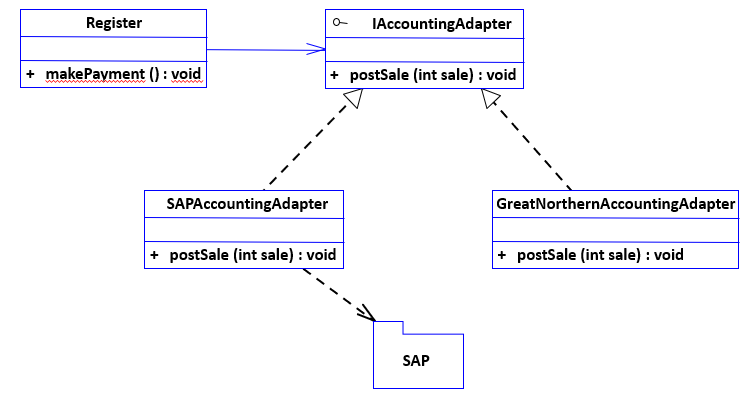
**Probleem:** Een client class verwacht een bepaalde interface voor een methode, maar de component of class die de service kan verlenen gebruikt een **andere** interface.

**Oplossing:** Gebruik een ADAPTER.

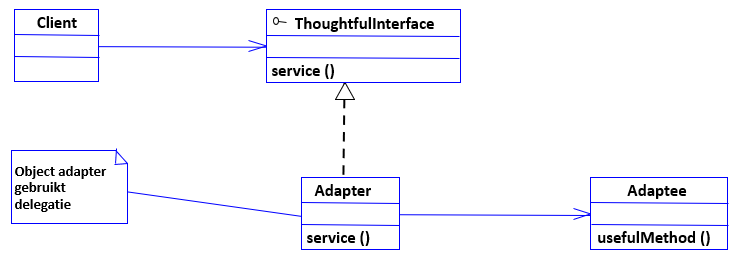
Je maak dus eigenlijk een brug tussen “ons systeem” en een extern systeem. Hierdoor kan je gemakkelijk switchen van systeem.



Figuur 11; SAPAccountingAdapter adapts to the SAP system



Figuur 12; Voorbeeld van een adapter DMV class diagram.



Figuur 13; Sjabloon voor code voorbeeld

1. public class Adapter {
2. private Adaptee adaptee;
3. public void service()
4. {
5. adaptee.usefulMethod();
6. …………
7. }
8. }

**Gebruik:**

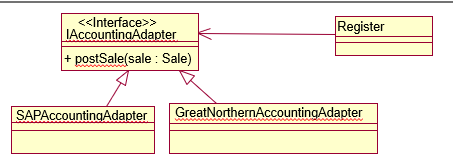
1. De meeste applicaties die gebruik maken van softwarebibliotheken van derden gebruiken adapters als middenlaag tussen de applicatie en de bibliotheek om de applicatie onafhankelijk te maken van de bibliotheek.
2. Als een andere bibliotheek wordt gebruikt heb je alleen een adapter nodig voor de nieuwe bibliotheek zonder de applicatie code te wijzigen.

## Factory Method

**Probleem:** Wie mag objecten aan maken? Wie kan dat het beste aanmaken?

**Oplossing:**  Je kan hiervoor GRASP Creator gebruiken of je kan een object Factory maken.

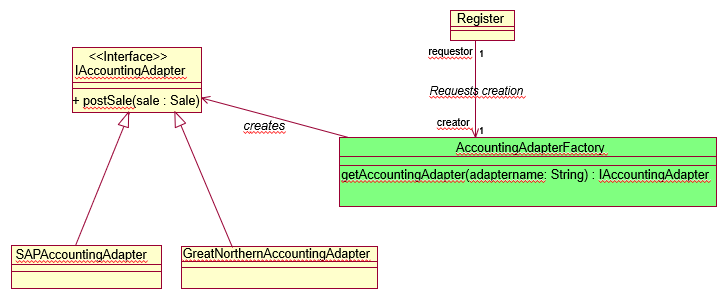
Neem bijvoorbeeld:



Volgens Creator moet Register de adapters aanmaken maar dit verlaagt de cohesie van Register.

En hoe bepaal je welke adapter je aan moet maken?

Hiervoor kan je dus een factory maken:



Figuur 14; Register heeft nu geen kennis van de concrete adapters.

Dit is simpele voorbeeldcode voor een factory:

Zie volgende pagina

1. public class AccountingAdapterFactory {
2. private static AccountingAdapterFactory instance;
3. private AccountingAdapterFactory() {
4. }
5. public static AccountingAdapterFactory getInstance(){
6. if (instance == null) {
7. instance = new AccountingAdapterFactory();
8. }
9. return instance;
10. }
11. public IAccountingAdapter getAccountingAdapter (String adaptername) {
12. …… }
13. }

# Bibliografie

Fadatare, R. (2018, 06 16). *High Cohesion GRASP Pattern*. Retrieved from sourcecodeexamples: https://www.sourcecodeexamples.net/2018/06/high-cohesion-grasp-pattern.html

Karsten, V. (onbekend). *📈 S.O.L.I.D: de eerste vijf principes van Object Oriented Design (OOD)* . Retrieved from Sigma Solutions: Online Business Partner: https://sigmasolutions.nl/blog/20150323-s.o.l.i.d-de-eerste-vijf-principes-van-object-oriented-design-ood/

M. Koolwaaij. (2020, 08 26). Thema 8 OO Design principles.

R. Fadatare. (2018, 06 15). *Controller GRASP Pattern*. Retrieved from sourcecodeexamples: https://www.sourcecodeexamples.net/2018/06/controller-grasp-pattern.html

R. Fadatare. (2018, 06 16). *Creator GRASP Pattern*. Retrieved from sourcecodeexamples: https://www.sourcecodeexamples.net/2018/06/creator-grasp-pattern.html

R. Fadatare. (2018, 06 16). *Information Expert GRASP Pattern*. Retrieved from sourcecodeexamples: https://www.sourcecodeexamples.net/2018/06/information-expert-grasp-pattern.html#:~:text=Information%20expert%20(also%20expert%20or,computed%20fields%2C%20and%20so%20on.

Wikipedia-bijdragers. (2016, 01 08). *GRASP*. Retrieved from Wikipedia: https://nl.wikipedia.org/wiki/GRASP#:%7E:text=GRASP%20is%20een%20Engels%20acroniem,kennen%20aan%20klassen%20of%20objecten.

# Figuren

[Figuur 1; Hoort het Syteem al deze events op te vangen of is dat dus een taak van een controller? 2](file:///C:\Users\Rober\Documents\1.0%20ICA\Leerjaar%202\blok%203\AIM-OOSE--\OOAD\Samenvatting%20OO%20design%20.docx#_Toc67423813)

[Figuur 2; Een Façade controller is in dit geval een algemene super controller. Dit is handig wanneer je een snel een applicatie wilt opzetten. Echter wanneer het (te) groot (-er) wordt is het handig om de meerdere kleine controllers te maken. 3](#_Toc67423814)

[Figuur 3; In dit geval maakt Sale gebruik van een SalesLineItem. Dus hebben we temaken met optie b. B legt A vast 4](#_Toc67423815)

[Figuur 4; Hier is goed te zien dat Information Expert is toegepast. Omdat, een total behoort bij een Sale, een SubTotal bij een SalesLineItem en een Prijs bij een Product(Specification) 4](#_Toc67423816)

[Figuur 5; Een voorbeeld van een observer pattern. 9](#_Toc67423817)

[Figuur 6; een uitgewerkt toets voorbeeld 9](#_Toc67423818)

[Figuur 7; Sequence diagram van het observer pattern 10](#_Toc67423819)

[Figuur 8; Een Class diagram die het pattern extra toelicht. 11](#_Toc67423820)

[Figuur 9; Een class diagram van een state machine in bijvoorbeeld java. 11](#_Toc67423821)

[Figuur 10; Concreet voorbeeld van een state machine 12](#_Toc67423822)

[Figuur 11; SAPAccountingAdapter adapts to the SAP system 12](#_Toc67423823)

[Figuur 12; Voorbeeld van een adapter DMV class diagram. 12](#_Toc67423824)

[Figuur 13; Sjabloon voor code voorbeeld 13](#_Toc67423825)

[Figuur 14; Register heeft nu geen kennis van de concrete adapters. 14](#_Toc67423826)