Programmeeropdracht Patterns

Maak gedurende maximaal twee uur een open boek ("open internet") programmeeropdracht:

* Bekijk de werking van een bestaande Java applicatie van ca. 100 regels. Er zijn voor de classes/methoden die moeten worden aangepast unit-tests beschikbaar die ook na de aanpassingen/refactorings moeten slagen.
* Voer verschillende refactorings uit zoals: Extract Constant, Extract Method, Move Method en Extract Class.
* Pas een design pattern toe om een design probleem op te lossen: Observer.
* Pas twee design patterns gecombineerd toe om een design probleem op te lossen: Adapter en Factory Method.
* Pas de SOLID-principes toe om een design probleem op te lossen.

Organiseer al je leeractiviteiten zo dat je na een sprint deze opdracht onder tijdsdruk kunt maken.

-refactor code

-magic numbers

-long methodes

-dup. Code

-apply patterns

-SOLID

[Refactoring](#_vcssxlwwrcad)

[Kent de refactoring principes en weet wanneer en waarom code gerefactord moet worden.](#_waa1nhxzb70a)

[Kent de rol van refactoring in de TDD-cyclus.](#_4k6352qudln0)

[Kent de volgende bad smells en herkent deze in broncode: Duplicated Code, Long Method and Large Class, Magic Number, Comments, Message Chains.](#_m0vhm3yrsqzg)

[Kent de volgende refactorings en kan deze in broncode toepassen om de bovenstaande bad smells te verwijderen](#_jfodgiw2glrc)

[Kan de refactoring opties van een IDE (IntelliJ, Eclipse) gebruiken.](#_khr6t8x8ptzg)

[Kent de volgende refactorings van naam maar hoeft deze niet toe te passen in de Broncode](#_8v6lzuhp923z)

[Observer](#_2lqbugrqek2d)

[Kent minimaal de volgende elementen van een generiek Design Pattern: Pattern Name and Classification, Intent, Motivation, Applicability, Structure](#_slj5g7f41sat)

[Kent de begrippen koppeling (coupling) en cohesie (cohesion) en kan deze gebruiken om ontwerp- of implementatiekeuzes toe te lichten.](#_jsvqys5ecaze)

[Kent van het Observer-pattern de Classification, Intent, Motivation, Applicability en Structure.](#_kkkfysp8hegu)

[Kan het Observer pattern in Java toepassen door gebruik van de API-onderdelen Observable en Observer.](#_j06kgr42361v)

[VoorBeeld](#_1jxfqf34favz)

[Uitleg:](#_9w3b9c9v7imi)

[Adapter](#_7lkh25qr2j0y)

[Kent de OO-begrippen delegatie, overerving, polymorfisme en interfaces.](#_htealtkfmutr)

[Kan zelfstandig interfaces definieren en implementeren.](#_qj72vn6g13jr)

[Kan het Adapter-pattern toepassen door gebruik te maken van delegatie en polymorfisme.](#_4giyh09c4xcf)

[Kent van het Adapter-pattern de Classification, Intent, Motivation, Applicability en Structure.](#_fy90lndar1so)

[Factory Method](#_c0zllu1fk8a)

[Kent van het Factory Method-pattern de Classification, Intent, Motivation, Applicability en Structure.](#_lw0njj6qujc)

[Kan het Factory Method-pattern op minimaal twee verschillende manieren implementeren in Java waarvan in elk geval een if/else-constructie op basis van Strings.](#_iv6llfkhnis9)

[Consider a high-class grocery store in London that stocks avocados all year round. It relies on a buyer to ensure that avocados arrive regularly, no matter what the time of year. The buyer sources the best avocados and supplies them to the store. The buyer is operating as a Factory Method, returning Kenyan, South African, or Spanish avocados depending on the time of year. Although the produce is labeled, the storekeeper is not particularly interested in the source of the products. Figure 5.2, "Factory Method pattern illustration-avocado sourcing" shows what might happen at different times of the year.](#_frie5rlkaba3)

[Kan het Factory Method-pattern gebruiken in combinatie met het Adapter-pattern.](#_u21k3hu4pdwv)

[Kent de kracht en beperkingen van het Singleton-pattern.](#_7lufhxkk09wq)

[SOLID](#_pv0bmn16058r)

[Kent de namen van de vijf principes waarvan SOLID de afkorting is.](#_zght1uknswj4)

[Single responsibility](#_su1po9lge8zz)

[Betekenis](#_cf1vgjglv8xz)

[Voordelen](#_wyybzbyl1aen)

[Nadelen](#_s2ig25qw6chx)

[Voorbeeld](#_qel312a5imlm)

[Open/closed](#_1p7c5qckx9g7)

[Betekenis](#_wkl0ho8dglyk)

[Voordelen](#_jdyvx4323btb)

[Nadelen](#_ie3bfukm5xy)

[Voorbeeld](#_scdi8j12a4vo)

[Liskov substitution](#_b7erhm2rzeyg)

[Betekenis](#_klmzwzn0b5nm)

[Voordelen](#_mjkh9x7s0mht)

[Nadelen](#_z4h8x1joa9bn)

[Voorbeeld](#_c94cdacld60b)

[Interface segregation](#_8sw8a6k8omxx)

[Betekenis](#_sl5ryckcbe0z)

[Voordelen](#_g2f1f6s2ovej)

[Nadelen](#_55hmd64833z0)

[Voorbeeld](#_c94uhgh4go5x)

[Dependency inversion](#_g1go5wpyaenp)

[Betekenis](#_98o5i1q5awr1)

[Voordelen](#_t2ecs8kv53at)

[Nadelen](#_b6cfmjsniw87)

[Voorbeeld](#_c566uoxgbfbw)

# 

# 

# Refactoring

**Beschrijving:** *As a student, I need to learn refactoring so I can improve the design, readability and testability of my code.*

**Inleiding:**

**Refactoren** (Engels: **refactoring**) is het herstructureren van de [broncode](https://nl.wikipedia.org/wiki/Broncode) van een [computerprogramma](https://nl.wikipedia.org/wiki/Computerprogramma) met als doel de leesbaarheid en onderhoudbaarheid te verbeteren of het stuk code te vereenvoudigen. Het refactoren van broncode verandert de werking van de software niet: elke refactorstap is een kleine, ongedaan te maken stap die de leesbaarheid verhoogt zonder de werking aan te passen. Het wordt met name gebruikt bij [extreme programming](https://nl.wikipedia.org/wiki/Extreme_programming), een vorm van [agile-softwareontwikkeling](https://nl.wikipedia.org/wiki/Agile-softwareontwikkeling).

https://nl.wikipedia.org/wiki/Refactoren

## Kent de refactoring principes en weet wanneer en waarom code gerefactord moet worden.

Refactoring principes

1.Code verminderen. Dit is een doel van refactoring

2.Slimme code vermijden. Hou je code simpel.

3.Single responsibility. Alles doet maar één ding.

4.Duplicatie verminderen/verwijderen.

5.Dependency verminderen/verwijderen

6.Self documenting code. Dus gebruik methode namen als telBijElkaar(int getal1, int getal2).Geen comments nodig.

7.Code moet in een paar seconden te begrijpen zijn. Refactoren is ten slotte om code duidelijker te maken.

8.Focus je niet op te kleine dingen. Pak grote dingen eerst aan.

9.Vaker kleine stapjes ipv soms grote stukken.

10.Methodes moeten maar één ding doen en voor andere dingen andere functies aanroepen.

https://myadventuresincoding.wordpress.com/2010/07/03/refactoring-principles/

**Waarom refactoren?**

Je refactort zodat je code duidelijker en makkelijker te lezen is. Als je aan het coderen bent dan blijkt het meestal dat je dingen niet precies genoeg hebt uitgedacht. Dit leidt tot lange onduidelijke code en dit moet je dan refactoren. Refactoren doe je meestal na je klaar bent met een blok code schrijven, niet tijdens want dan blijf je refactoren en duurt het schrijven te lang.

## Kent de rol van refactoring in de TDD-cyclus.

In de TDD-cyclus is er een red/green/refactor moment. Hierin maak je een unit test die faalt, zorg je er in de code voor dat de unit test succeed, en daarna schoon je de code op. Het opschonen is het refactor deel, en dit zorgt ervoor dat je elk stukje gemaakte code refactort, en dus opschoont voordat men doorgaat naar de volgende stuk code.

## Kent de volgende bad smells en herkent deze in broncode: Duplicated Code, Long Method and Large Class, Magic Number, Comments, Message Chains.

**Bad smells**

Bad smells zijn vrijwel vaak voorkomende stukken code welke ervoor zorgen dat de code niet overzichtelijk, onleesbaar minder onderhoudbaar wordt.

https://en.wikipedia.org/wiki/Code\_smell

**Duplicated Code**

Duplicated Code is, zoals de naam al suggereert, gedupliceerde code. Hierbij is het zo dat een stuk code twee, of meerdere keren voorkomt.

https://sourcemaking.com/refactoring/smells/duplicate-code

**Long Method**

Long Method is, zoals de naam al suggereert, een te lange methode. Hierbij is het zo dat een methode te veel lines heeft, waardoor hij te groot is.

https://sourcemaking.com/refactoring/smells/long-method

**Large Class**

Large Class is, zoals de naam al suggereert, een te grote class. Hierbij kan het zijn dat een class te veel doet, of dingen die niet in die class thuishoren.

https://sourcemaking.com/refactoring/smells/large-class

**Magic Number**

Een magic number is een nummer of constante welk weinig voorkomt, maar toch onduidelijk is wat het betekent. Hierbij moet je denken aan een nummer in een stuk code welk beter kan vervangen worden door een constante met een naam, wat uitlegt wat het nummer is of doet.

https://en.wikipedia.org/wiki/Magic\_number\_(programming)

**Comments**

Comments zijn, wonderbaarlijk, ook bad smells. De reden hiervoor is dat methode namen en dergelijke voor zichzelf moeten spreken. Ze moeten namen hebben die zelf uitleggen wat ze doen. Wanneer er comments bijstaan is het vaak zo dat dit niet het geval is.

https://sourcemaking.com/refactoring/smells/comments

**Message Chains**

Message Chains zijn aanroepingen op bepaalde waardes of objecten. Als bijvoorbeeld in class A iets wordt aangeroepen wat in B is, maar B dit moet aanroepen in C, en C weer in D, enzovoort, dan heb je te maken met een Message Chain.

https://sourcemaking.com/refactoring/smells/message-chains

## 

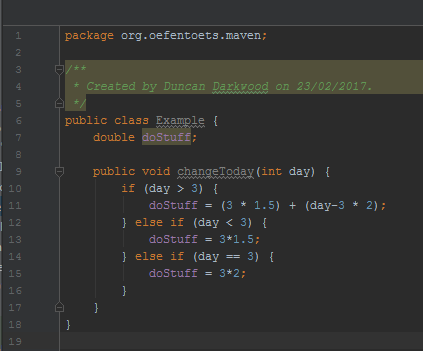
## 

## Kent de volgende refactorings en kan deze in broncode toepassen om de bovenstaande bad smells te verwijderen

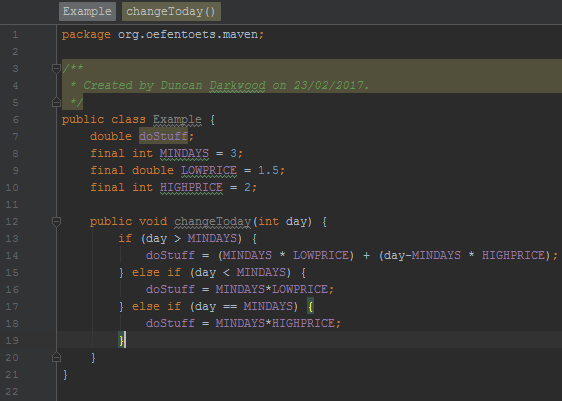
**Extract Constant**

Extract Constant is het maken van een constante voor een iets wat meerdere malen in de code voorkomt. Bij het iets moet je denken aan int, double, string, boolean, etc.

Een voorbeeld hiervan zou zijn:

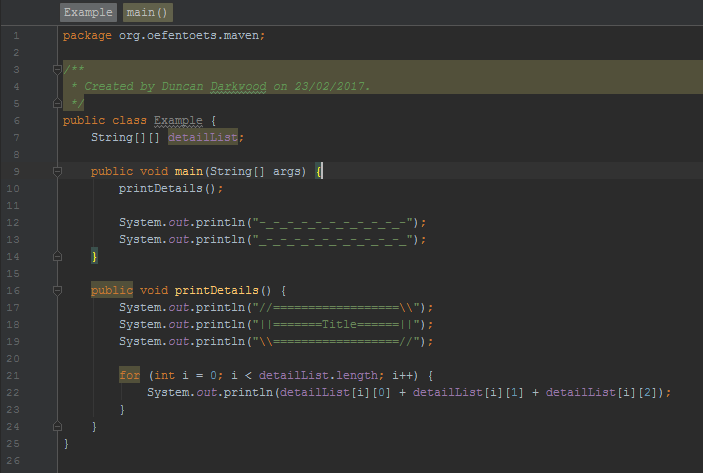


Wordt dan

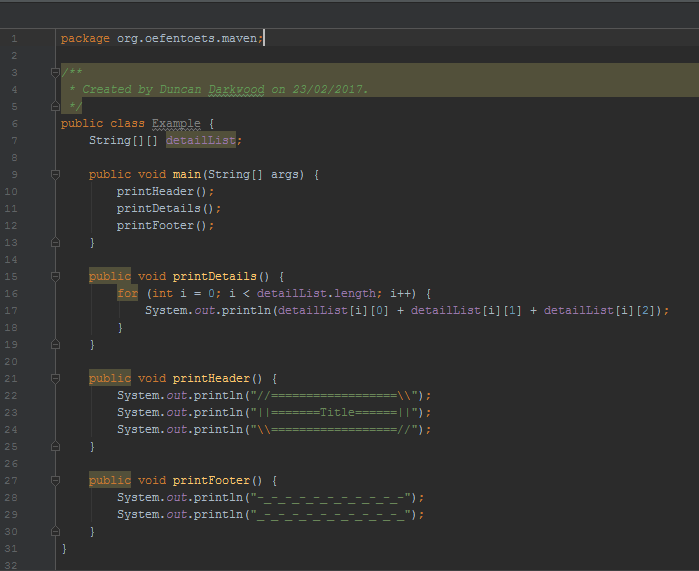


**Extract Method**

Extract Method is een techniek wat ervoor zorgt dat wanneer je meerdere lijnen code hebt die tot hetzelfde doeleinde worden gebruikt, worden samengevoegd in 1 methode. Of wanneer er 1 methode is welk meerdere dingen doet, dat dat dan wordt opgebroken in meerdere kleinere methodes. Bijvoorbeeld

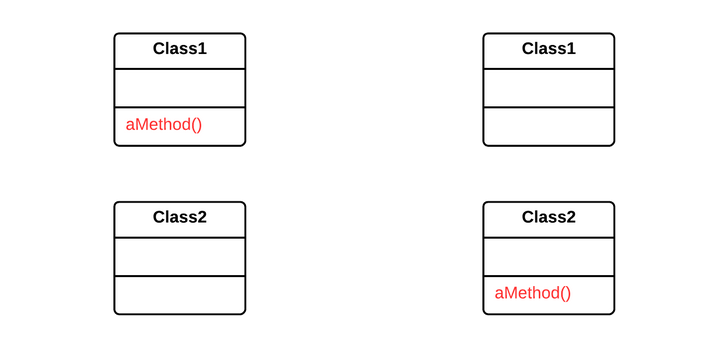


Wordt dan



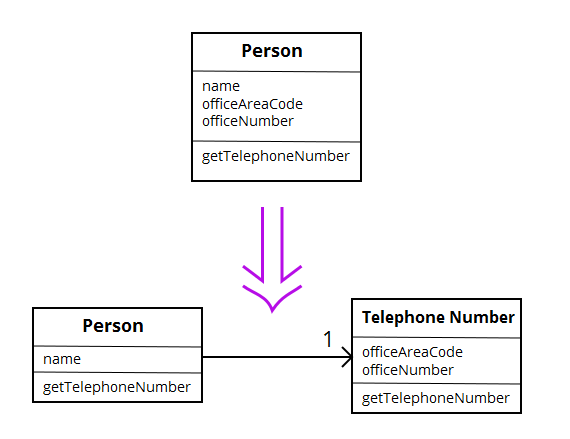
**Move Method**

Wanneer er een methode is in een class die vaker dingen moet halen uit een andere class, of vanuit daar vaker wordt opgeroepen, kan je Move Method toepassen. Hierbij verplaats je de methode naar de andere class waar hij dat vaker gebruikt wordt.

****

**Extract Class**

Wanneer je een class hebt die te veel dingen doet, of dingen doet die eigenlijk niet in die class thuishoren, kan je Extract Class gebruiken. Hierbij breek je de te groote class op in meerdere kleine classes. Een goed voorbeeld zou bijvoorbeeld deze afbeelding kunnen zijn



## 

## 

## Kan de refactoring opties van een IDE (IntelliJ, Eclipse) gebruiken.

Rechterklik op de naam van je methode/functie/klasse etc, kies refactor.

## Kent de volgende refactorings van naam maar hoeft deze niet toe te passen in de Broncode

Replace conditional (if else, switch cases) with Polymorphism: <https://refactoring.guru/replace-conditional-with-polymorphism>

Replace temp with query:

<https://refactoring.com/catalog/replaceTempWithQuery.html>

# 

# Observer

BRONNENLIJST:

<http://www.software-innovators.nl/2009/01/02/een-aspect-van-het-observer-patroon/>

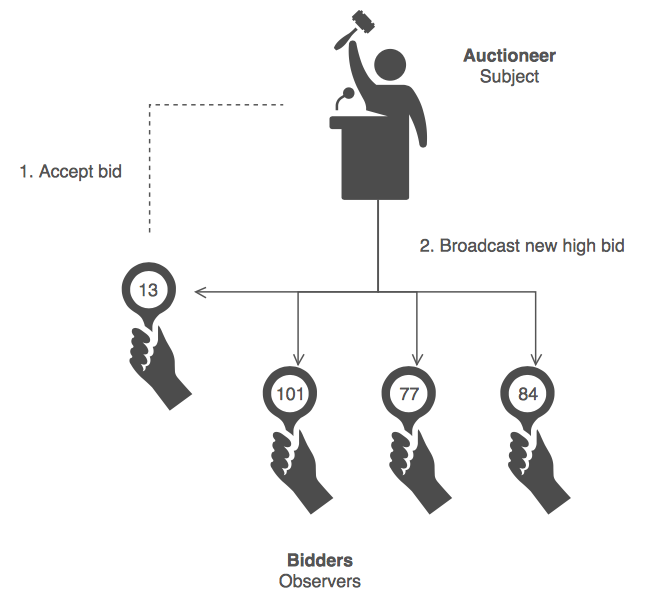
<https://nl.wikipedia.org/wiki/Observer_(ontwerppatroon)>

**Beschrijving:** *As a student, I need to learn the Observer Design Pattern so I can lower the coupling between classes using a standardized interface.*

**Inleiding:** Het observer patroon is een software ontwerp patroon waar een object, het onderwerp, een lijst bijhoudt van afhankelijken, bekend als observers. Het object/onderwerp stelt de observers op de hoogte van veranderingen, meestal door het aanroepen van methodes.

Het observer patroon wordt voornamelijk gebruikt voor gedistribueerde event handling (computer evenementen die door software/systeem worden getriggerd) systemen.

De observer definieert een één-op-veel relatie relatie zodat wanneer een object van staat verandert, de observers daarvan (automatisch) op hoogte worden gebracht.



In bovenstaande afbeelding is een voorbeeld van een observer ontwerp. Hier is het onderwerp de veiler; na elk bod dat hij accepteert communiceert de veiler het hoogste bod weer door aan de bieders (observer).

<https://sourcemaking.com/design_patterns/observer>

## Kent minimaal de volgende elementen van een generiek Design Pattern: Pattern Name and Classification, Intent, Motivation, Applicability, Structure

<https://en.wikipedia.org/wiki/Software_design_pattern>

Een Design Pattern kent documentatie; hierin staat beschreven in welke context het ontwerp patroon gebruikt wordt, wat het op te lossen probleem is en hoe dit opgelost kan worden.

Voor deze ontwerppatronen is er geen standaard formaat voor de documentatie. Een veel gebruikte documentatie is die de documentatie uit het boek Design Patterns, van de ‘Gang of Four’ (Gamma, Helm, Johnson, Vlissides). In het boek zijn de volgende secties beschreven:

* **Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.
* **Intent:** A description of the goal behind the pattern and the reason for using it.
* **Also Known As:** Other names for the pattern.
* **Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.
* **Applicability:** Situations in which this pattern is usable; the context for the pattern.
* **Structure:** A graphical representation of the pattern. [Class diagrams](https://en.wikipedia.org/wiki/Unified_Modeling_Language#UML_Class_Diagram) and [Interaction diagrams](https://en.wikipedia.org/wiki/Interaction_diagram) may be used for this purpose.
* **Participants:** A listing of the classes and objects used in the pattern and their roles in the design.
* **Collaboration:** A description of how classes and objects used in the pattern interact with each other.
* **Consequences:** A description of the results, side effects, and trade offs caused by using the pattern.
* **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
* **Sample Code:** An illustration of how the pattern can be used in a programming language.
* **Known Uses:** Examples of real usages of the pattern.
* **Related Patterns:** Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

Pattern name & classification:

*The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary. The pattern's classification reflects the scheme we introduce in Section 1.5.*

Intent:

A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Motivation:

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability:

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure:

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [RBP+91]. We also use interaction diagrams [JCJO92, Boo94] to illustrate sequences of requests and collaborations between objects. Appendix B describes these notations in detail.

## 

## Kent de begrippen koppeling (*coupling*) en cohesie (*cohesion*) en kan deze gebruiken om ontwerp- of implementatiekeuzes toe te lichten.

Het observer patroon zorgt ervoor dat objecten aan kunnen geven dat hun status is veranderd zonder harde koppelingen te maken. Het verlagen van koppelingen zorgt ervoor dat het aanmaken van nieuwe classes en het veranderen van bestaande classes minder werk kost.

Stel je voor je hebt 10 classes; een die TV heet en 9 verschillende kijkers (denk aan moeder,vader,oma,hond,kinderen) als je zonder observable zou werken dan zou de tv naar elke persoon (elke class dus) apart een koppeling moeten maken met de tv. De tv stuurt dan alle informatie naar iedereen die ermee gekoppeld is. Als de kat dan de kamer binnenloopt en ook naar de tv wilt kijken dan zou de tv eerst weer een koppeling met de kat moeten maken, in plaats van dat de kat gewoon kan zeggen ik wil nu alle tv beelden ontvangen. Als je de tv observable maakt dan kan iedereen gewoon zeggen dat ze een observer willen zijn van de tv en dan zien ze alles wat de tv laat zien.

Cohesie betekent samenhang. Als je veel samenhang hebt tussen 2 classes dan kan je hier een koppeling tussen maken; bijvoorbeeld een deurklink die een deur openmaakt. Die deurklink hoeft niet observable gemaakt te worden want er is maar een andere class die iets doet als de deurklink gebruikt wordt. Als je een spel hebt waar de actieknop ingedrukt wordt, dan zijn er veel dingen die kunnen gebeuren. Je kan de actieknop overal mee koppelen maar het is veel beter om de acties die kunnen gebeuren observers te maken en de actieknop observable.

TL;DR

Koppeling heeft met observer te maken omdat je het gebruik van sterke koppelingen kan voorkomen door gebruik te maken van het observer patroon,

Cohesie heeft met observer te maken omdat cohesie vaak bepaalt of het gebruik van het observer patroon nodig of nuttig is.

## 

## 

## Kent van het Observer-pattern de Classification, Intent, Motivation, Applicability en Structure.

Uit het ‘Gang of Four’ (GoF) boek:

**Pattern name & classification:**

*The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary. The pattern's classification reflects the scheme we introduce in Section 1.5.*

**Intent:**

*A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?*

Het definiëren van een een-op-veel afhankelijkheid tussen objecten. Wanneer één object van staat veranderd, worden de afhankelijken (observers) genotificeerd en automatisch geupdate.

(In deze update methode kunnen weer acties uitgevoerd worden).

De observer staat ook bekend al Publish-Subscribe.

**Motivation:**

*A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.*

Om classes samen te laten werken heb je consistentie nodig, wat je kunt doen door de klassen op een bepaalde manier te koppelen. Dit heeft als nadeel dat de herbruikbaarheid van een klasse verminderd.

Met de implementatie van het Observer patroon kun je classes samen laten werken zonder (veel) afhankelijkheid in te bouwen en daarnaast kunnen er een heel aantal observers per onderwerp zijn.

*Voorbeeld: grafieken bij een data-source. Je wilt niet dat de grafieken al te afhankelijk zijn van de data-source. Wanneer je iets aanpast dan moet het allemaal wel nog werken.*

*Met de Observer techniek wordt een data-source geupdate, en notificeert de source de observers (grafieken). Deze kunnen dan automatisch geupdate worden. Er kunnen nu ook aanpassingen in de classes van grafieken en source plaatsvinden zonder dat dit effect heeft/hoeft te hebben op andere classes.*

**Applicability:**

*What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?*

De observer pattern kan gebruikt worden in de volgende gevallen:

* Wanneer een abstractie twee aspecten kent, en het ene aspect afhankelijk is van de andere. Deze aspecten opdelen in aparte objecten maakt het mogelijk verschillen te implementeren en de objecten onafhankelijk te hergebruiken.
* Wanneer het veranderen van een object ook vereist dat andere objecten mee veranderen. Vooral handig als je niet weet hoeveel objecten dit betreft.
* Wanneer een object andere objecten moet notificeren, zonder kennis te hebben van wat deze objecten zijn. De objecten kennen dan low-coupling, maar toch kan er interactie plaats vinden.

**Structure:**

*A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [RBP+91]. We also use interaction diagrams [JCJO92, Boo94] to illustrate sequences of requests and collaborations between objects. Appendix B describes these notations in detail.*

Een diagram die het patroon weergeeft. Lijkt heel erg op / is in de basis een domein model / class diagram.

## 

## Kan het Observer pattern in Java toepassen door gebruik van de API-onderdelen *Observable* en *Observer*.

### 

### VoorBeeld

Hier is een voorbeeld waarin observable en observer gebruikt worden.

In App.java staat:

**package** com.oose;

**import** java.util.Observable;

**import** java.util.Observer;

*// This is the class being observed.*

**class** News **extends** Observable {

**void** news() {

String[] news = {**"News 1"**, **"News 2"**, **"News 3"**};

**for**(String s: news){

*//set change*

setChanged();

*//notify observers for change*

notifyObservers(s);

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

System.***out***.println(**"Error Occurred."**);

}

}

}

}

*//Run Observer and Observable*

**class** ObserverObservableDemo {

**public static void** main(String args[]) {

News observedNews = **new** News();

NewsReader reader1 = **new** NewsReader(**"firstNewsReader"**);

NewsReader reader2 = **new** NewsReader(**"secondNewsReader"**);

observedNews.addObserver(reader1);

observedNews.addObserver(reader2);

observedNews.news();

}

}

En in NewsReader.java staat:

**package** com.oose;

**import** java.util.Observable;

**import** java.util.Observer;

*//*

*///\*\**

*// \* Created by allard on 09/02/2017.*

*// \*/*

**public class** NewsReader **implements** Observer{

String **newsReaderName**;

**public void** update(Observable obj, Object arg) {

System.***out***.println(**newsReaderName**+**" got The news:"**+(String)arg);

}

**public** NewsReader(String name) {

**newsReaderName**=name;

}

}

### Uitleg:

je moet gebruik maken van extend Observable en implement Observer in de classes die je observable moet maken of wilt observen.

Dit zorgt ervoor dat je gebruik kan maken van de addObserver() methode.

Die gebruik je op deze manier;

observableObject.addObserver(observerObject);

In dit geval observableObject een instantie van een class die observable extends en observerObject een instantie van een class de observer implements.

Deze methode voegt observerObject toe aan de lijst met instanties die genotificeerd worden als de observableObject instantie update.

Dit updaten gebeurt als je notifyObservers(meeGeefWaarde) gebruikt, wel gebeurt dat alleen als je eerst setChanged(); gebruikt. Zonder dit doet notifyObservers() niks. Na het gebruiken van notifyObservers moet je weer setChanged gebruiken als je weer notifyObservers wilt gebruiken.

In de class die Observer extends staat een functie update(Observable obj, Object arg) {}

Deze functie wordt aangeroepen als notifyObservers() uitgevoerd wordt in een functie die de observer aan het observen is.

Obj is het object dat update aanroept en arg is de waarde die je mee geeft in notifyObservers.

# 

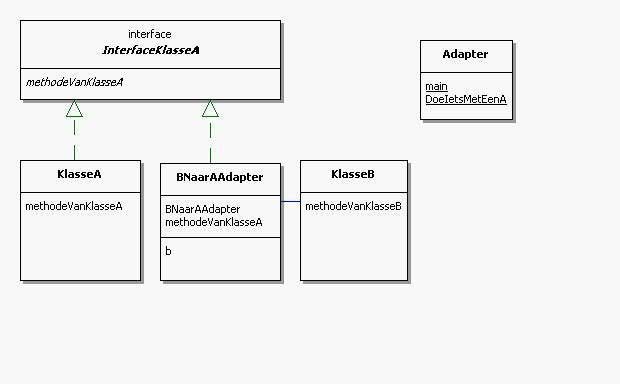
# 

# Adapter

**Beschrijving:** *As a student, I need to learn the Adapter Design Pattern so I can use classes with a different set of functions through a shared interface.*

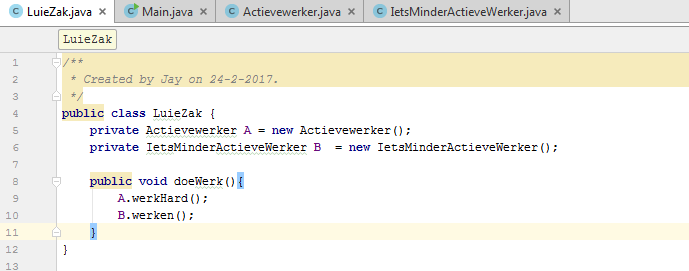
**Inleiding:**

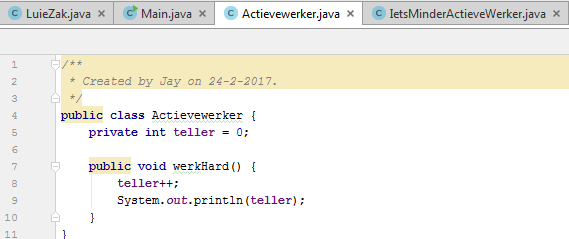
De Adapter Design Pattern is een design methode waarbij je een adapter class maakt om 2 classes met verschillende interfaces met elkaar te laten samenwerken. In de afbeelding hieronder is te zien dat KlasseA en KlasseB niet dezelfde interface hebben, maar met dank aan de adapter BNaarAAdapter kunnen deze twee classes toch samenwerken

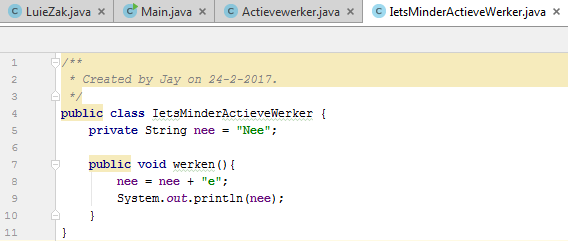


https://nl.wikipedia.org/wiki/Adapter\_(ontwerppatroon)

## Kent de OO-begrippen delegatie, overerving, polymorfisme en interfaces.

Delegatie is het doorschuiven naar iets anders. In het onderstaand voorbeeld is te zien hoe dit in zijn werk gaat.





Hierboven is dus te zien dat de klasse LuieZak zelf geen werk uitvoert. Het enige wat deze doet is tegen de andere klassen zeggen dat zij werk moeten doen. Het is een klasse die methoden aanroept van andere klassen. Dit is delegeren.

Polymorfie

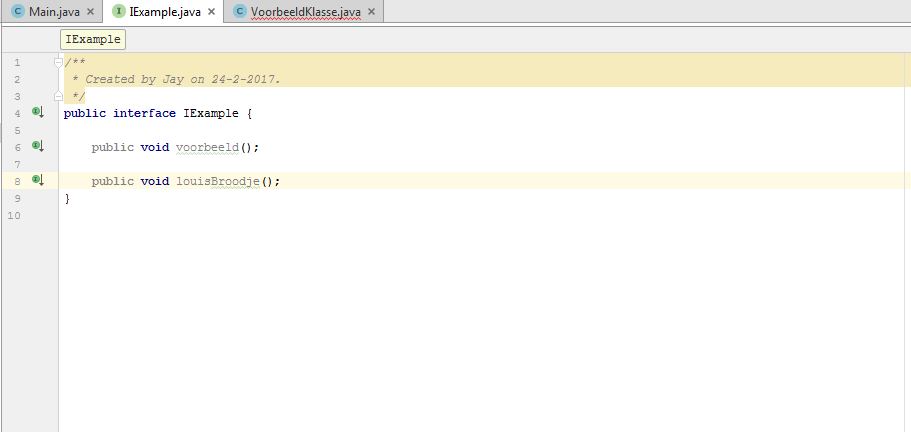
Polymorfisme is het gebruik van super- en sub-classes. Hierbij heb je bijvoorbeeld een hoofdclasses “Animal” die de standaard methoden heeft die alle dieren hebben, en dan de sub-classes daarvan kunnen alle verschillende dieren zijn. Dat is Polymorfisme.

Interface

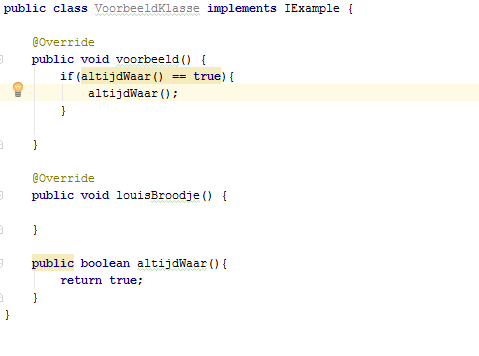
Ik hoop dat je weet wat een interface is. Zo niet zie plaatjes in het volgende hoofdstuk.

## Kan zelfstandig interfaces definieren en implementeren.

Ik neem aan dat iedereen dit al weet maar hier even een voorbeeld van hoe je een interface implementeert met twee methoden. Dus ipv public class IExample doe je public interface IExample.



En dan hieronder een voorbeeld van hoe je die implementeert, houd er rekening mee dat je ook alle methoden van je Interface moet gebruiken in de klasse.



## 

## 

## Kan het Adapter-pattern toepassen door gebruik te maken van delegatie en polymorfisme.

**Delegatie**

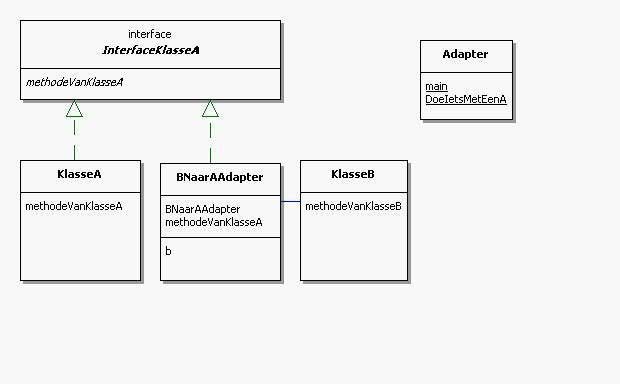
Zie onderdeel OO-begrippen

**Polymorfisme**

Zie onderdeel OO-begrippen

**Delegatie en polymorfisme in Adapter-patterns**

Om dit uit te leggen is het handig om te kijken naar de volgende afbeelding



Als je hierin kijkt zie je dat er in de interface een methode staat genaamd “methodeVanKlasseA”. Deze staat hierom dus ook in de BNaarAAdapter class, welk onze adapter is. Deze class moet dus een methode aanroeper welk in KlasseA staat. Hierbij kan je je voorstellen dat in de methode “methodeVanKlasseA” van class “BNaarAAdapter”, iets staat zoals:

public void methodeVanKlasseA() {

KlasseA.methodeVanKlasseA();

}

Hiermee kan KlasseB de methode van KlasseA oproepen via BNaarAAdapter, terwijl BNaarAAdapter eigenlijk het werk doorgeeft naar KlasseA (wat Delegatie is).

Wat heeft polymorfisme hiermee te maken? Nou, zoals je kan zien is InterfaceKlasseA een grote interface die wordt geëxtend door beide KlasseA en BNaarAAdapter. Hierbij moet in de adapter de methode “methodeVanKlasseA” gebruikt worden.

## Kent van het Adapter-pattern de Classification, Intent, Motivation, Applicability en Structure.

**Classification**

* **Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.
* Staat ook bekend als ‘Wrapper’.

**Intent**

* **Intent:** A description of the goal behind the pattern and the reason for using it.
* De interface van een klasse omzetten zodat de klasse gebruikt kan worden door andere klassen die anders geen compatibele interface hebben.

**Motivation**

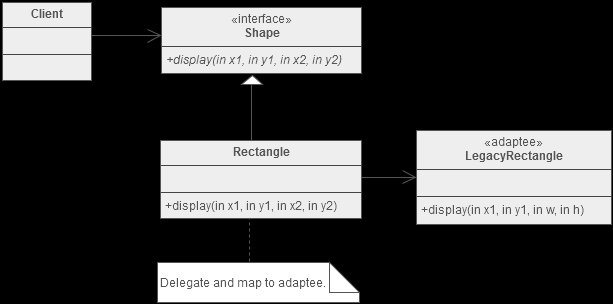
* **Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.
* Classes kunnen gebruiken die niet overeenkomstige methodes (namen) hebben via een interface toch aan te kunnen roepen. Bijvoorbeeld methodes als ‘tekenLijn’ en ‘tekenPolygon’. Deze zou je normaal gesproken niet via een interface aan kunnen roepen; ze hebben namelijk andere namen. Door tussen de interface en klassen een adapter te zetten, wordt de functie doorgelust als het ware, en kun je voor elk object/instantie dezelfde functie aanroepen. In deze adapter interface heb je dan iets staan als public void tekenFiguur(){ tekenLijn() }

**Applicability**

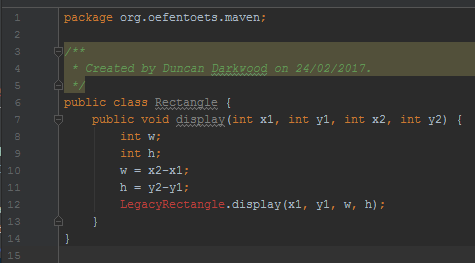
* **Applicability:** Situations in which this pattern is usable; the context for the pattern.

**Structure**

Om de structuur van een adapter goed te bekijken, nemen we het volgende ontwerp als voorbeeld.



Zoals hierboven te zien is, vraagt de Client bij de Interface “Shape” een display op en geeft hierbij de x en y positie van de hoek links-boven en recht-onder, tewijl de class “LegacyRectangle” vraagt om een x en y positie van de hoek links boven, en daarna voor een breedte en hoogte. Om ervoor te zorgen dat de Client alsnog de juiste display krijgt is de adapter Rectangle gemaakt waarin de methode van “Shape” wordt omgezet naar de methode van “LegacyRectangle”. Dit kan er bijvoorbeeld zo uitzien:



# 

# 

# Factory Method

**Beschrijving:** *As a student, I need to learn the Factory Method Design Pattern so I can centralize the creation of objects with a shared interface.*

**Inleiding:** <<tekst over factory method>>

## Kent van het Factory Method-pattern de Classification, Intent, Motivation, Applicability en Structure.

**Classification**

Een factory is een class met de naam XFactory, waarbij X de naam is van de class waarvan je een factory maakt. Bijvoorbeeld, je hebt een class Person, dan heet de factory PersonFactory.

In een Factory heb je methode waarbij de class als een object wordt aangemaakt, bijvoorbeeld de methode getObjectX in het voorbeeld hieronder. Dan is er een private constructor en een public static getFactory. In getFactory wordt de factory aangemaakt en gereturned naar degene die het opvroeg.

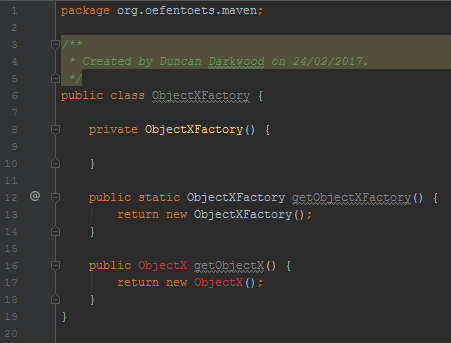
**Intent**

Factory Method-patterns worden gebruikt om te voorkomen dat een object op verschillende locaties wordt aangemaakt. Dit wordt hiermee centraal op 1 locatie gezet. Dit wordt met name gebruikt om het keyword ‘new’ uit te code te schrappen.

**Motivation & Applicablility**

Stel, je hebt een programma waarbij je in 5 verschillende classes op 5 verschillende punten een classe X moet aanmaken als een object. Op het moment dat er iets moet worden aangepast in het aanmaken van object X, moet er op 5 verschillende plekken iets worden aangepast. Wanneer er een Factory wordt gemaakt voor X, dan worden deze 5 punten gecentraliseerd op 1 locatie, waardoor je steeds maar op 1 locatie iets hoeft aan te passen.

**Structure**

****

Zo kan een factory eruit zien. Zoals je ziet is er methode getObjectX. Deze maakt het object aan en returned die aan degene die het opvraagt. Met de static getObjectXFactory kan er een ObjectXFactory worden aangemaakt en returned zodat getObjectX gebruikt kan worden.

## Kan het Factory Method-pattern op minimaal twee verschillende manieren implementeren in Java waarvan in elk geval een if/else-constructie op basis van Strings.

## Consider a high-class grocery store in London that stocks avocados all year round. It relies on a buyer to ensure that avocados arrive regularly, no matter what the time of year. The buyer sources the best avocados and supplies them to the store. The buyer is operating as a Factory Method, returning Kenyan, South African, or Spanish avocados depending on the time of year. Although the produce is labeled, the storekeeper is not particularly interested in the source of the products. Figure 5.2, "Factory Method pattern illustration-avocado sourcing" shows what might happen at different times of the year.

## Kan het Factory Method-pattern gebruiken in combinatie met het Adapter-pattern.

## Kent de kracht en beperkingen van het Singleton-pattern.

De singleton heeft als voordeel dat je maar een enkele instantie van die klasse hebt. En als nadeel dat Singletons meestal globaal zijn en dat kan leiden tot locking bij threading.

(we hoeven dit voor de rest niet te kunnen toepassen)

# SOLID

**Beschrijving:** *As a student, I need to learn the SOLID principles so I can apply general design guidelines to decrease coupling between classes and increase cohesion within classes.*

**Inleiding:**

SOLID is een afkorting die gebruikt wordt in verband met het programmeren van computers. De afkorting staat voor de vijf belangrijkste principes in het object-georiënteerd programmeren. Deze principes samen maken het waarschijnlijker dat een programmeur een systeem zal maken dat duurzamer is en gemakkelijk te onderhouden of uit te breiden. De SOLID-principes maken deel uit van een meer globale strategie van agile ontwikkeling en adaptief programmeren.

<https://nl.wikipedia.org/wiki/SOLID>

# Kent de namen van de vijf principes waarvan SOLID de afkorting is.

## **S**ingle responsibility

### Betekenis

Een klasse mag maar één verantwoordelijkheid hebben.

De single verantwoordelijkheid principe stelt dat elke module of klasse verantwoordelijkheid voor een enkel onderdeel van de functionaliteit van de software moet hebben, en dat de verantwoordelijkheid moet volledig worden ingekapseld door de klasse. Al haar diensten strikt moet worden aangepast aan die verantwoordelijkheid. Robert C. Martin drukt het principe als volgt:

"Een klasse mag maar één reden om te veranderen. "

### Voordelen

* Zorgt voor duidelijke klassen met maar één doeleinde waardoor je gemakkelijk weet waar alles zou staan.

### Nadelen

Kan ervoor zorgen dat het extra ongeordend wordt, omdat simpele doeleinde ineens een grote klasse hebben.

### Voorbeeld

Een klasse genaamd ‘Person’ is een single responsibility, want dat heeft één doeleinde. Als de intentie is om twee dingen te veranderen is dus de bedoeling om daarvan ook twee aparte klassen te maken.

## **O**pen/closed

### Betekenis

software-entiteiten moeten open zijn voor uitbreiding, maar gesloten voor wijziging. Dat betekent dat bepaalde gedrag uitbreidbaar moet zijn zonder dat er programmeercode gewijzigd hoeft te worden.

### Voordelen

* Zorgt ervoor dat functionaliteiten gemakkelijker toegevoegd kunnen worden in de toekomst

### Nadelen

* Kost meer tijd om te ontwikkelen en veilig te hebben

### Voorbeeld

<http://www.oodesign.com/open-close-principle.html>

## **L**iskov substitution

### Betekenis

De classes die erven van dezelfde interface moeten alles uit deze interface gebruiken. Wanneer een klasse geen gebruik maakt van een bepaalde methode dan hoort hij niet bij de interface waar hij van erft.

### Voordelen

* Dit zorgt ervoor dat alle klassen gerangschikt worden door middel van de interfaces.

### Nadelen

* Dit zorgt voor extra interfaces wat meer tijd kost om te maken.

### Voorbeeld

Een voorbeeld is de wekkerklasse. In de wekkerklasse staan de [functie](https://nl.wikipedia.org/wiki/Subprogramma) maakGeluid en het [veld](https://nl.wikipedia.org/w/index.php?title=Veld_(objectori%C3%ABntatie)&action=edit&redlink=1) geluid. Op het moment dat de wekker hoort te gaan wordt de functie maakGeluid aangeroepen en gaat het alarm af. Een analoge wekker is een wekker en is zo gemaakt dat het de wekker overerft. Ook een digitale wekker is een wekker en erft dus van de wekkerklasse. De dovenwekker is ook een wekker, maar nu is er een probleem. Deze wekker trilt in plaats van dat hij geluid maakt. Het is dus tegen het substitutieprincipe van Liskov om de dovenwekker te laten overerven van de wekkerklasse, de dovenwekker heeft namelijk geen nood aan het veld geluid en heeft een andere implementatie van maakGeluid.

Nog een voorbeeld:

<https://lassala.net/2010/11/04/a-good-example-of-liskov-substitution-principle/>

## **I**nterface segregation

### Betekenis

“veel client-specifieke interfaces zijn beter dan één algemene interface.”

### Voordelen

* Zorgt voor schone en duidelijke code.

### Nadelen

* Zorgt voor extra classes.

### Voorbeeld

Wanneer je een bedrijf hebt waar je een interface workers hebt. Stel deze workers werken en hebben een half uur lunchtijd. Dus een methode iWork en iLunch. Als er dan een deel van de workers wordt vervangen door robots dan loop je tegen het principe interface segregation aan. Als je je aan het principe interface segregation wil houden moet je nieuwe interfaces maken omdat de robots geen lunch pauze nodig hebben.

<http://www.oodesign.com/interface-segregation-principle.html>

## **D**ependency inversion

### Betekenis

"men zou afhankelijk moeten zijn van abstracties, niet van concrete implementaties"

In object-georiënteerd ontwerp, de afhankelijkheid inversie beginsel heeft betrekking op een specifieke vorm van ontkoppeling softwaremodules. Bij het volgen van dit principe, de gevestigde van hoog niveau conventionele afhankelijkheidsrelaties, politiek-instelling modules om low-level, afhankelijkheid modules zijn omgedraaid, waardoor high-level modules onafhankelijk van de low-level implementatie module informatie renderen. Het principe luidt als volgt:

A. High-level modules mag niet afhankelijk zijn van low-level modules. Beide moeten afhangen van abstracties.

B. Abstracties mag niet afhangen van details. Details moet afhangen van abstracties.

Dit ontwerp principe keert de manier waarop sommige mensen kunnen denken over het object-georiënteerd programmeren, dicteert dat zowel high- en low-level objecten moeten afhankelijk van dezelfde abstractie.

Het idee van het schrijven in twee punten van dit principe is dat bij het ontwerpen van de interactie tussen een hoog niveau module en een laag niveau is, de interactie is uitsluitend abstract interactie daartussen gedacht. Dit heeft niet alleen gevolgen voor het ontwerp van de high-level module, maar ook op een laag niveau: de low-level één moeten worden ontwikkeld om de interactie in het achterhoofd en kan het nodig zijn gebruik-interface veranderen.

Vaak denken de interactie op zichzelf als een abstract concept maakt de koppeling van de componenten te reduceren zonder dat er additionele coderende patronen, waarbij alleen een lichtere en minder uitvoering afhankelijke interactieschema.

Wanneer de ontdekte abstract interactie schema (s) tussen twee modules is / zijn generiek en generalisatie is logisch, dit ontwerp principe leidt ook tot de volgende afhankelijkheid inversie codering patroon.

### Voordelen

* Gemakkelijker onderhoudbaar in de lange termijn, vooral met grotere code

### Nadelen

* Mensen hebben soms de neiging om te veel onderscheid te maken tussen lower level en higher level.

### Voorbeeld

<http://www.oodesign.com/dependency-inversion-principle.html>