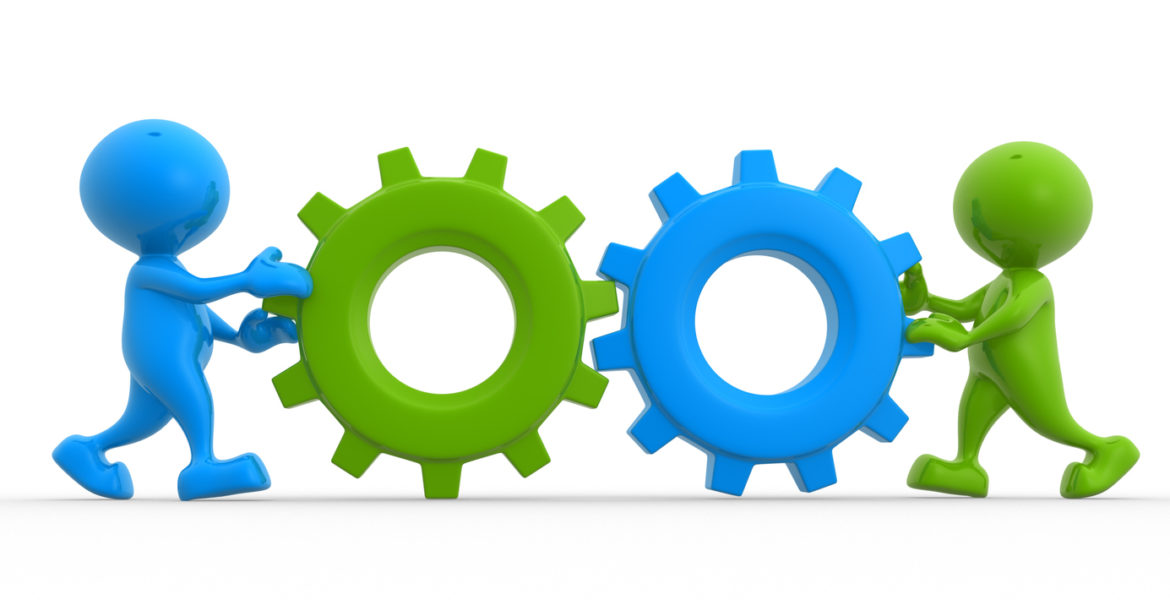
Implementatie

Gemaakt door: Kevin Mout

# OO-technieken.

Object georiënteerd programmerenObject georiënteerd programmeren (OOP) is een techniek binnen programmeertalen. Aan de hand van een object wordt een datastructuur gecreëerd met verschillende gegevens en procedures. is een manier van programmeren waarbij logica en gegevens worden georganiseerd rondom objecten. Aan de hand van een object wordt een datastructuur gecreëerd met verschillende gegevens en methodes. Dit programmeermodel geldt al vele jaren als de standaard binnen softwareontwikkeling. De manier waarop logica wordt opgedeeld rondom objecten zorgt ervoor dat code leesbaar en beheerbaar blijft (in tegenstelling tot sequentieel opgebouwde code).

## OO-principes.

S.O.L.I.D. is een afkorting van vijf principes voor het object georiënteerd programmeren die bedacht zijn door Robert C. Martin (ook bekend als Uncle Bob). Zie ook deze link (Links to an external site.) voor meer informatie. Wanneer deze vijf principes samen worden toegepast zal de code eenvoudig te onderhouden en uit te breiden zijn. Specifieke implementatie-leerdoelen zijn gebaseerd op deze principes.

### S: Single responsibility.

betekent dat de verantwoordelijkheid van het veranderen van een module bij 1 actor ligt. Een module kan zo simpel zijn als een klasse maar dit kan ook een groep van classes zijn binnen een package.

### O: Open/closed.

Software-entiteiten moeten open zijn voor uitbreiding, maar gesloten voor wijziging.

### L: Liskov substitution.

Objecten in een programma zouden vervangbaar moeten zijn door instanties van hun subtypes zonder dat de juiste werking van het programma beïnvloed wordt.

### I: Interface segregation.

Veel client-specifieke interfaces zijn beter dan één algemene interface.

### D: Dependency inversion.

Men zou afhankelijk moeten zijn van abstracties, niet van concrete implementaties.  
Dependency injection is een manier om dit principe toe te passen.

## Inherintance.

Inherintance is een relatie tussen objecten. Dit kan je meestal checken met “is a”, “is a kind of” of “is a type of”. In UML met klassendiagrammen, gaat de pijl van wat voor “is a” staat naar wat na “is a” staat. Die “child” class heeft alles van de “Parent” class.

## Inherintance in c#.

<https://www.youtube.com/watch?v=XK3ByhVcg3Q&ab_channel=JamesRoche>

In dit filmpje hebben ze een person class. Ze willen een nieuwe class maken met als naam student. We moeten meteen zien dat dit een Inheritance is, want “a student is a person”.

Public class Student : Person{ die : Person is de inheritance.

Public string course {get;set;}

Public Person(string name, int age, string course) : base(name,age)

{

Course = course;

}

}

Je kan nu alles van Person gebruiken in student class.

Foreach loop kan je (p as AnotherClass).function() gebruiken.

Als je een virtual object gebruikt in de parent class dan kan je het overriden in de child class.

## Abstracte klassen en interfaces.

Eerder in het vak is inheritance (generalisatie) behandeld, een manier om gedeelde functionaliteit over verschillende klassen te kunnen herbruiken. Dit leidt echter in sommige gevallen tot een probleem: soms is het niet gewenst of onlogisch dat je een instantie hebt van een bepaalde klasse. Een hond blaft, een paard hinnikt, maar wat voor een geluid komt er uit de base klasse dier?

Dit probleem is te ondervangen door gebruik te maken van abstracte klassen. Hiermee is aan te geven dat een klasse wel functionaliteit in zich heeft, maar de klasse zelf niet te instantiëren is. Het andere onderwerp van deze module, interfaces, trekken dit concept tot in het extreme door: een interface bevat helemaal geen implementatie, alleen definities.

Daarnaast maken we kennen met sorteren. Dit onderwerp spreekt redelijk voor zich: heel vaak heb je een aantal dingen in een lijst staan, en die dingen moeten gesorteerd worden. Voor getallen is dit conceptueel triviaal: je begint met de kleinste en eindigt met de grootste. Maar, hoe bepaal je nu welke waarde op welke positie in de lijst moet komen staan? Daarnaast kun je natuurlijk ook een lijst van objecten (instanties) hebben. Hoe bepaal je bijvoorbeeld hoe een lijst van studenten gesorteerd moet worden? In eerste instantie misschien op naam, maar wat nu als je studenten ook op woonplaats of semester wilt sorteren? Binnen gebruiken we hiervoor IComparable en IComparer: twee interfaces die verwerkt zitten in het .NET-framework.

### Abstracte classes.

Een abstracte class is gelijkwaardig aan een interface met een verschil. Namelijk dat een class die van deze class is afgeleid, niet van een andere class kan worden afgeleid.

Stel de Square class is afgeleid van de AbstractShape class dan kunnen we zeggen:

AbstractShape Shape = new Square();

AbstractShape shape = new AbstractShape(); kan niet.

Een abstracte class kan niet abstract zijn en afgesloten.

### Interfaces.

Methodes mag je niet instantiëren.

Properties mag je gebruiken.

Interface is een contract. Een class moet alle properties en methodes gebruiken dat in een interface staat.

Public, private en protected mag je niet gebruiken in een interface. Ze zijn allemaal public.

Reden van interfaces.

1. Om een klasse toe te staan meerdere gedragingen van meerdere interfaces te erven.

2. Om naamambiguïteit tussen de methoden van de verschillende klassen te vermijden, zoals bij het gebruik van meervoudige overerving in C ++.

3. Om twee of meer interfaces te combineren zodat een klasse alleen het gecombineerde resultaat hoeft te implementeren. De voorbeeldcode hiervoor wordt geleverd als een te downloaden bronbestand.

4. Om Naam verbergen toe te staan. Naam verbergen is de mogelijkheid om een overgenomen lidnaam te verbergen voor elke code buiten de afgeleide klasse.

<https://www.youtube.com/watch?v=P7Xa8zyNnpw&ab_channel=CheezyCode>