Principles of Object Ooriented design (in Ditch)

Version: Draft (0.0.5)

Uitgelegd door:

Galina Slavova

Software Engineer en Consultant

LinkedIn: http://linkd.in/gslavova
Twitter: @galinas
Telefoon: 06-36149017
E-mail: galina@crafity.com
Website: www.crafity.com

Table of Contents

Principles of Object Ooriented design (in Ditch) **1**

1 Inleiding **3**

2 Class **4**

2.1 Wat is een class? **4**

2.2 Members van een class **4**

2.2.1 Methodes 4

2.2.1 Properties en variables 5

2.3 Encapsulation en scoping **7**

3 Relaties tussen objecten **9**

3.1 Association – “kent”, “is bekend bij” **9**

3.2 Aggregation – “bevat een”, “gebruikt” **10**

3.3 Composition – “bestaat uit”, “is een intrinsiek deel van” **11**

3.4 Inheritance - “is a” **12**

3.4.1 Single vs multiple inheritance 15

4 Data abstraction **16**

4.1 Abstracte classes **16**

4.2 Interfaces **18**

5 Polymorfism **22**

5.1 Overloading van methodes, operatoren **22**

6 Specialisatie en generalisatie **24**

7 SOLID Principles door uncle Bob (Robert C. Martin) **24**

8 Bronnen **24**

Crafity BUILT WITH PASSION

1 Inleiding

Het paradigma van Object Orientatie sluit aan bij de behoefte om uiteenlopende bedrijfsprocessen in brede schaal te automatiseren. OO bestaat sinds de jaren 70 en werd geimplementeerd in 3de generatie (3GL) talen, zoals C++, Java, C#.

De programmeertalen van tegenwoordig (jaar 2012) zijn vaak een gecombineerde implementatie van het functionele en het OO concept.

Het functioneel programmeren is geschikt in domeinen waar exacte antwoorden gevraagd worden op exacte vragen. Een input waarde komt een function in en wordt verder gebruikt in calculaties in dezelfde of vervolgfuncties… In het eind komt er een antwoord uit.

Het OO concept vervult de behoefte om “open” vragen te stellen. Daaruit kunnen uiteenlopende antwoorden komen. Met objectorientatie kan een simulatieprogramma gemaakt worden om het gedragspatroon te bepalen van autos, bussen, brommers, fietsers, voetgangers op een druk kruispunt bijvoorbeeld. In zo’n dynamisch uittreksel van een systeem wordt er rekening gehouden met het gedrag van elk onderdeel: wat is het gedrag van een teenager op de fiets ten opzichte van een bejaarde, of een moeder met haar kind op de fiets. De state van deze uiteenlopende objecten is dan verschillend. De antwoorden zijn verschillend.

Software applicaties met grafische user interface (views en controls, zoals buttons en boxes, sliders) is een ander voorbeeld waar het systeem d.m.v. objecten beschreven wordt. OO is vooral populair geworden met deze event-driven windows systemen.

De code snippets in deze tutorial zijn geschreven de taal C# en zijn bedoeld om een voorbeeld te geven over de besproken concepten. Ze zijn niet allemaal onderdelen van een geheel programma. Op sommige voorbeelden zou de ijverige lezer zich ongetwijfeld afvragen over enige refactoring in het huidige ontwerp. Graag vragen en opmerkingen die bewijzen dat het OO concept goed begrepen is.

De volgende termen zijn intrinsiek voor het OO paradigma en worden allemaal behandeld in de finale versie van deze tutorial:

- Encapsulation
- Data abstraction
- Inheritance
- Polymorphism
- Message passing
- Extensibility
- Persistence
- Delegation
- Genericity
- Multiple Inheritance

2 Class

2.1 *Wat is een class?*

Class is een model of sjabloon voor het genereren van objecten met gelijksoortige structuur en gedrag. Class is de beschrijving van een type entiteit, de DNA. Object is de instantiatie ervan. Class leeft in design-time. Object leeft in run-time van het programma.

2.2 *Members van een class*

Een class heeft een naam en de volgende members:

- constructor (en destructor)
- properties en variabelen
- methoden – functions en procedures, getters en setters
- delegates

2.2.1 Methodes

In OO gedesigned applicatie zijn methodes het middel voor communicatie tussen objecten. De “message passing”.

Crafity BUILT WITH PASSION

De **constuctor** is een speciale methode, waarmee nieuwe instanties (oftewel levende objecten) van deze class worden gemaakt.

De **destructor** is een speciale methode, waarmee een object uit het geheugen wordt opgeruimd (sommige talen, zoals C# en Java, hebben geen destructors nodig omdat de deallocatie van geheugen automatisch wordt verzorgd door een Garbage Collection mechanisme)

De class **methodes** vormen in het algemeen de interface van het geinstantieerde object met de buiten wereld.

Functions en Procedures

In de wereld van methodes worden er verschillende crassificaties gebruikt. Er wordt gesproken over twee soorten methodes: *functions* en *procedures*. **Functions** zijn de methodes die input ontvangen en uitput teruggeven (met de “return” keyword), terwijl **procedures** geen waarde teruggeven, maar kunnen de “state” van objecten veraanderen. Procedures worden ook *modifiers* genoemd.

Er is geen duidelijke grens nog restrictie in dit begrip. Het is de verantwoordelijkheid van de engineer om de consistentie van de method signatuur te behouden.

Getters en Setters

Voor het opvragen van de state van het object worden *getter* en *setter* methodes gebruikt. Met getters wordt de “state” van het object opgevraagd en nooit veranderd. Getters zijn “read-only”. Met de setters wordt de state van het object veranderd. Setters zijn dus modifiers, mutators.

Let op: Het is een goed OO gebruik om de implementatie van selectors en modifiers gescheiden te houden.

Nogmaals:

Methods	
Functions	Procedures
Selectors	Modifiers

2.2.1 Properties en variables

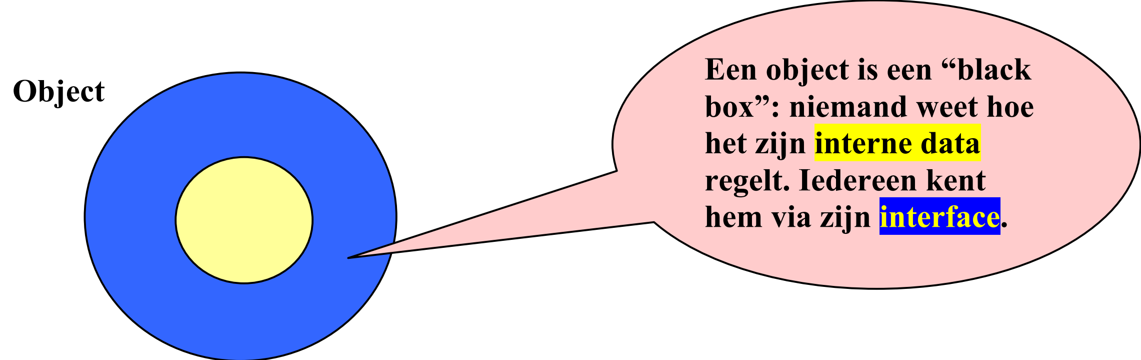
Class is een datastructuur waarin de state van elk dataonderdeel bijgehouden wordt. Variabelen en properties omhelzen deze dataonderdelen. In class Person, in het voorbeeld hieronder, zijn name, dateOfBirth en acquaintances:

```
public class Person
{
      private string _name;
      private DateTime _dateOfBirth;
      private List<Person> _acquaintances;

      public Person (string name, DateTime dateOfBirth)
      {
            _name = name;
            _dateOfBirth = dateOfBirth;
      }

      public string GetName()
      {
            return _name;
      }

      public DateTime DateOfBirth()
      {
            return _dateOfBirth;
      }

      public void MakeNewAcquaintance(Persoon p)
      {
            _acquaintances.Add(p);
      }
```

```
        // some other code...

        public string ToString()
        {
              return string.Format("Name: {0} \n\rDate of birth:
        {1} \n\r", name, dateOfBirth, PrintAcquaintances();
        }


        // entry point van het programma
        static void Main(string[] args)
        {
              Person annie = new Persoon("Annie", 12);
              Person roy = new Persoon("Roy", 21);

              Console.WriteLine(
                    annie.ToString() + roy.ToString()
              );
        }
}
// end of class Person
```

2.3 Encapsulation en scoping

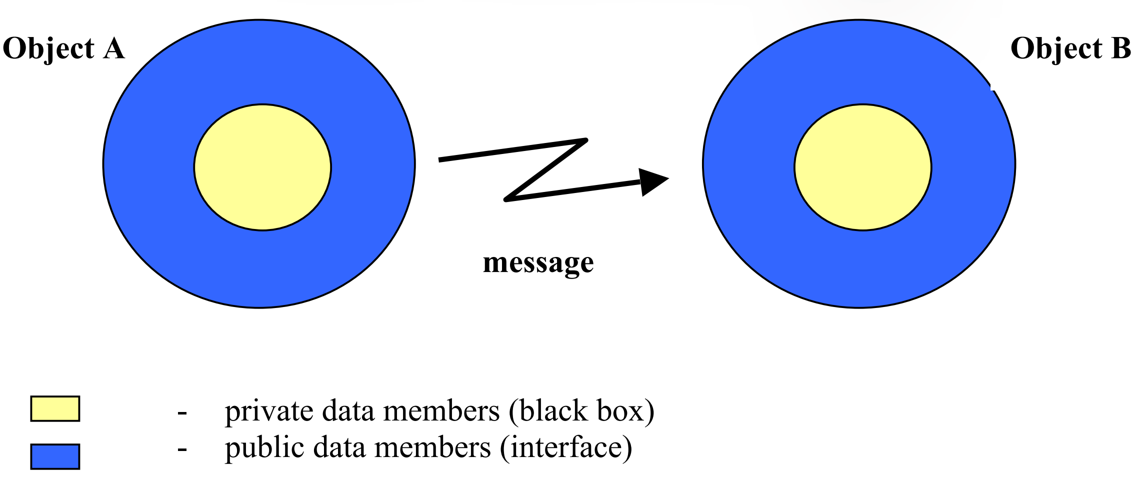
Wat in de scope van een class gebeurt is een prive zaak. De relaties met andere classes worden gerealiseerd d.m.v. hun public members en interfaces.



Gezien het feit dat een class zeer “sociaal” betrokken is in een systeem, is het logisch om restricties te leggen voor zijn communicatie met de buiten wereld. Class als een datastructuur en heeft specifieke data onderdelen die van de buitenwereld afgeschermd moeten blijven. Andere onderdelen van zijn data zijn juist publiektoegankelijk. In de OO wereld heet dit **Encapsulation** (encapselen van data).

Bijvoorbeeld een Persoon object zal niet willen dat iedereen zijn officiele naam of geboortedatum verandert:

BUILT WITH PASSION



- private data members (black box)
- public data members (interface)

Dus elke member van een class heeft een **“access modifier” – private, public, protected**.

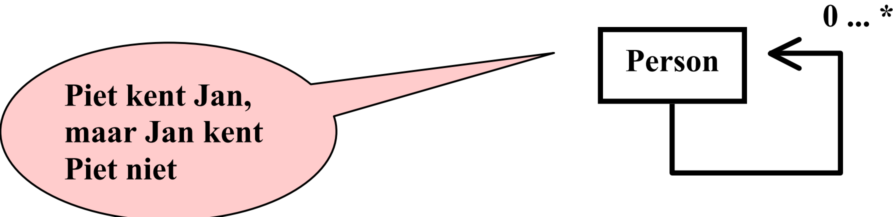
```
public class Person
{
	private string _name;
	private DateTime _datOfBirth;
	private List<Person> _acquaintances;

	public Persoon(string n, DateTime d)
	{
		_name = n;
		_dateOfBirth = d;
	}

	public string getName()
	{
		return _name;
	}

	// …
}
```

privé data members

publieke data members

3 Relaties tussen objecten

3.1 Association – “kent”, “is bekend bij”



Objecten zijn autonoom. Zij bestaan zonder elkaar te kennen. Associatie betekent een verbinding tussen twee autonome objecten. Assotiaties zijn de kanalen voor “message passing” tussen deze objecten. Een zo’n relatie kan enkelzijdig of wedezijdig zijn.

```
public class Person
{
	private string _name;
	private DateTime _datOfBirth;
	private IList<Person> _acquaintances;

	// …
}
```

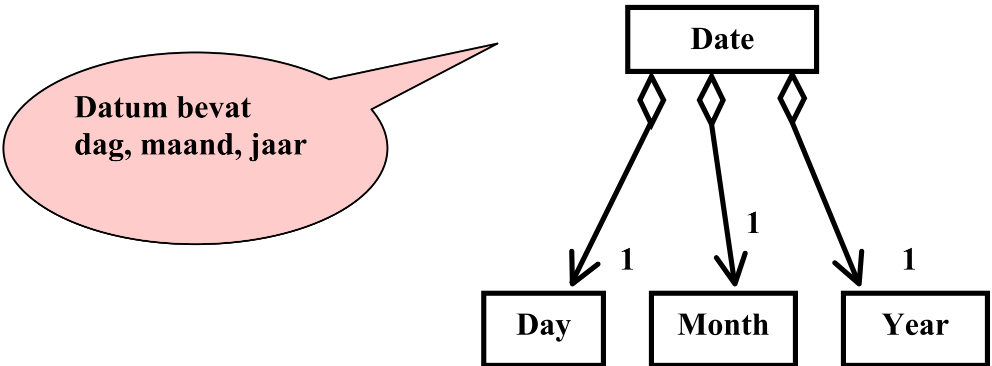
Crafity
BUILT WITH PASSION

Let op:
Een assotiatie werkt vaak als een dependencie.

In een domein-driven design neigt men om objecten snel met elkaar te verbinden. Hierdoor kan het overzicht verloren worden van wie in dit systeem de verantwoordelijkheid draagt, en dan - voor welke taken.

Een advies hier kan zijn om OO te gebruiken in een process-driven design: eerst nadenken over de hoofd “business” processen waarvoor dit systeem gaat dienen. Circulaire dependencies moeten in alle tijden vermeden worden.

3.2 Aggregation – “bevat een”, “gebruikt”



Aggregatie is een speciaal type assotiatie. Bij aggregatie spreken we over een relatie tussen geheel en deel, tussen een object en de samenstellende delen. Day, Month en Year zijn objecten die op zichzelf gebruikt kunnen worden in het programma

Voorbeelden van aggregaten:

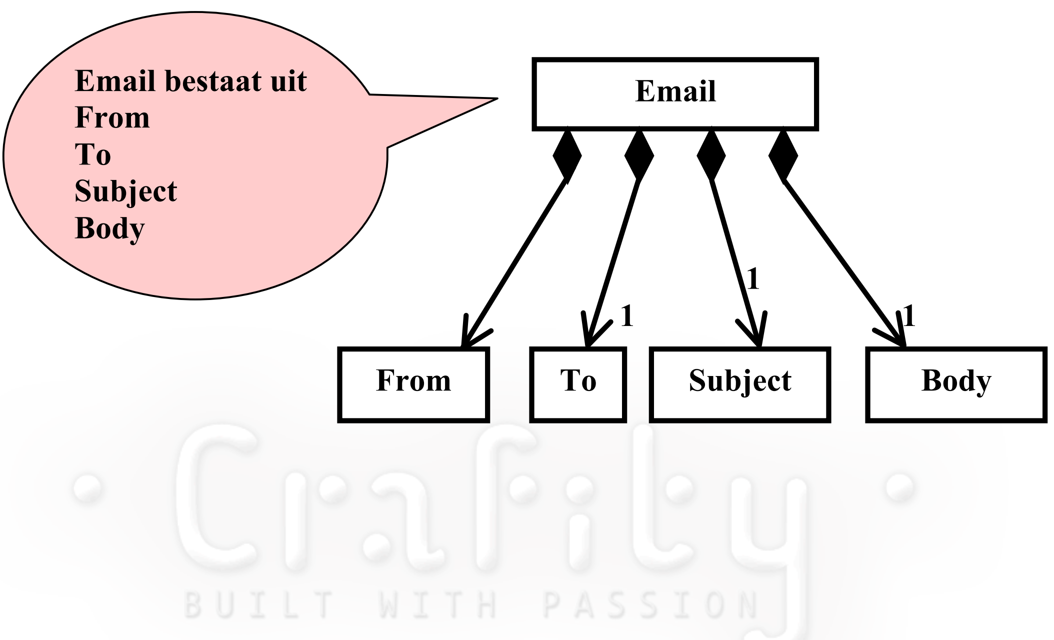
- koor bestaat uit zangers
- nummers van een tijdschrift (bestaan uit artikelen)
- een bedrijf (bestaat uit afdelingen)
- een binaire boom (bestaat uit andere bomen, of nodes)

BUILT WITH PASSION

3.3 Composition – “bestaat uit”, “is een intrinsiek deel van”

Dit is ook een speciaal type assotiatie. Compositie heeft, in tegenstelling tot aggregatie, sterkere kontrole tussen geheel en deel. Het geheel is nu ook verantwoordleijk voor het managen van life-cycle van het deel. Het geheel is de “eigenaar” van het deel. Als het geheel opgeruimd wordt (door garbage collector), dan wordt het deel ook opgeruimd.

In deze definitie kan het deel niet onafhankelijk van het geheel bestaan.

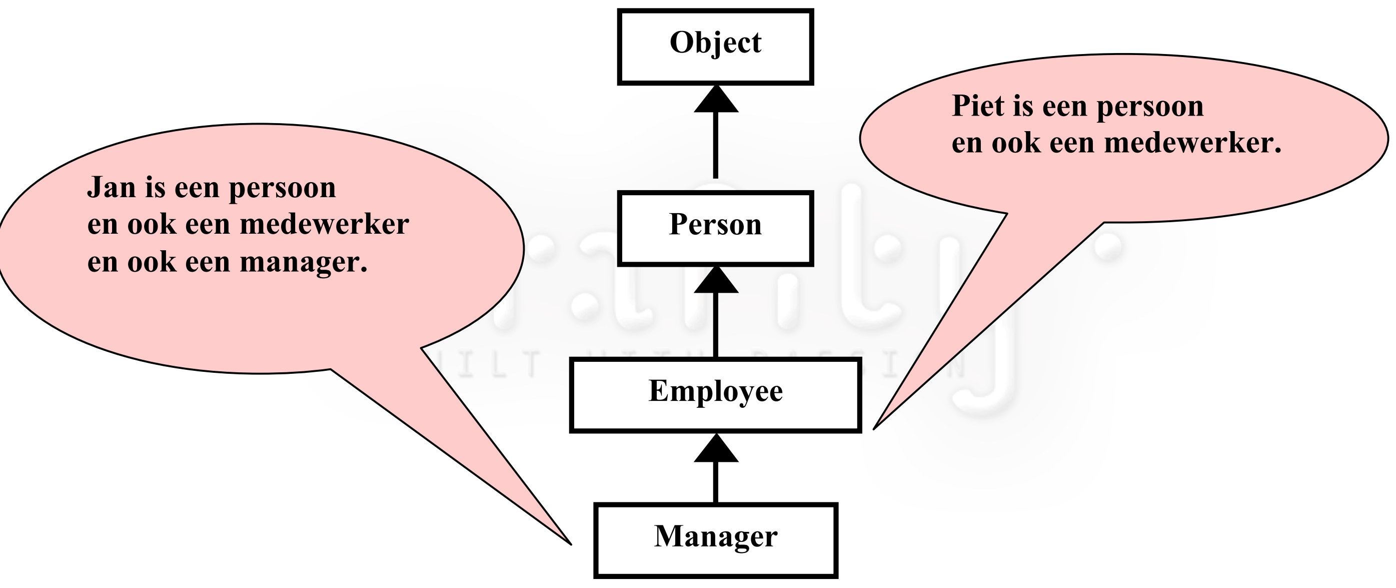


Craftly
BUILT WITH PASSION

3.4 Inheritance - "is a"

Inheritance (overerving) is een belangrijke concept in de OO wereld. Praktisch gezien is dit een hergebruik van al geschreven code. Sommige objecten erven alle eigenschappen van andere bestaande objecten en voegen extra eigenschappen aan zichzelf toe. Deze objecten noemen we ook "children" van de andere objecten.

Manager is een child van Employee en heeft alles wat Employee heeft plus nog meer.
Employee is een parent van Manager en een child van Person.
Person is een directe parent van Employee en indirecte voor Manager.
Person heeft ook een parent, bepaald in de concrete programmeertaal. In dit geva is dit class **Object**!



Let op: Inheritance is handig als deze met mate gebruikt wordt. Een datastructuur met inheritance kan beter niet verder gaan dan ongeveer twee niveau's diep. Diepe inheritance patronen kan bij snel groeiende applicaties extra complex worden. Advies is om eerder interfaces te gebruiken dan diepe inheritance structuren.

```
public class Employee : Person
{
	private int _sofinummer;
	private double _salary;

	public Employee(sring n, DateTime d, int sof, double sal)
: base(n, d)
	{
		_sofinummer = sof;
		_salary = sal;
	}

	// other code
}



public class Manager : Employee
{
	private double _bonus;

	public Manager(string n, DateTime d, int sof, double sal,
double b): base(n, d, sof, sal)
	{
		_bonus = b;
	}

	// other code
}
```

Crafity BUILT WITH PASSION

De class Employee heeft alles wat Person heeft (naam, geboortedatum en dezelfde methodes) plus extra attributen - sofi-nummer en salaris met bijbehorende methodes. Bij overerving praten we over **specialisatie** van het overervende object.

Het komt vaak voor dat de ervende class een deel van de functionaliteit van de base class wil aanpassen volgens zijn behoeftes. Dit betekent dat sommige methodes van de base class opnieuw gedefinieerd moeten worden bij de ervende class. Dit heer **method overriding**.

De **overriden** methoden van de kind-class zien er precies dezelfde uit als de methoden in de basis class. Ze hebben dezelfde **signatuur**. Alleen de implementatie in de method body's is anders. Zoals bijvoorbeeld de methode toString() van class Person. In de overriding-method kan de inhoud van de parent-class methode volledig overgeschreven worden of juist werken als een verrijking van de oude parent methode:

```
public class Employee : Person
{

      // attributes and some other code
      // …

      public string ToString()
      {
            string object = base.ToString() + "\nSalary: " +
            salary;

            return object;
      }
}
```

Er werd eerder in deze tutorial gezegd dat de scope van alle class members bepaald wordt me de access modifier: public, private en protected. In de context van inheritance worden de **protected** access modifiers gebruikt. Dit betekent dat elke protected member van deze class wel toegankelijk is voor al zijn child-classes en niemand anders. Een private member is niet toegankelijk. Ook niet voor de children.

```
public class Person
{
      protected string name;
      protected DateTime datOfBirth;
      protected IList<Person> acquaintances;

}
```

protected data members

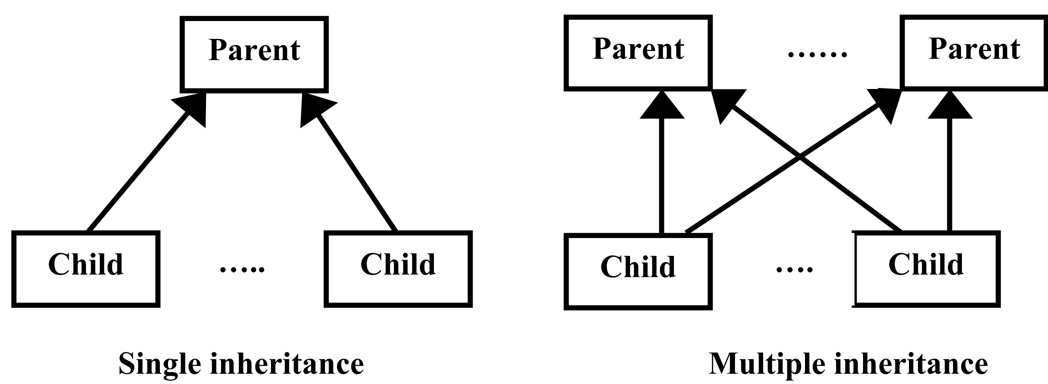
BUILT WITH PASSION

De basis class Person maakt nu zijn attributen toegankelijk ook voor de ervende classes Employee en Manager door de access modifier te vernaderen van **private** naar **protected**. Dit was bekend als encapsulatie. Dus in de class Employee kan nu in zijn toString methode direct gebruik maken van de attributen naam, geboortedatum en kennissen.

```
public class Employee : Person, IEmployee
{

      // attributes and some other code
      // …

      public string ToString()
      {
            return string.Format("Employee: {0}"\nSalary: {1}",
            name, salary;)
      }
}
```

Let op: Alle methodes in C# zijn per definitie non-virtual. Om een methode te kunnen aanbieden ter overschrijving moet deze virtual zijn in C#.

3.4.1 Single vs multiple inheritance



Met inheritance worden hiërarchieën van objecten gebouwd. Deze hierarchieen kunnen simpel of complex zijn. Een simpele hierarchie visueel gezien lijkt op een pyramide: er is een super class waarvan meerdere classes erven. Dit is bekend als **single inheritance**. Multiple inheritance is aanwezig als een ervende/child class van meerdere parents erft.

De meeste OO talen ondersteunen geen multiple inheritance om de complexieit te verminderen. Java en C# hebben geen multiple inheritance. C++ wel. De afwezigheid van multiple inheritance wordt gecompenseerd door interfaces.

4 Data abstraction

4.1 *Abstracte classes*

Alle voorbeelden van classes die we tot nu toe hebben bekeken zijn van concrete classes. Hun members zijn geïmplementeerd. Concrete classes kunnen geïnstantieerd worden tot objecten en ook afgeleid met inheritance.

Abstract classes zijn een blauwdruk en bevatten zowel geimplementeerde als abstracte members. De bedoeling is dat een abstract class geerfd wordt, omdat hij niet geïnstantieerd kan worden. Hij heeft echter een constructor die voor het instantieren van child-classes gebruikt wordt.

Een reden om een class als **abstract** te definieren is dat deze een abstracte beschrijving is van iets dat in de wereld geen instantie heeft. Bijvoorbeeld mensen, dieren, vogels en andere kunnen beschreven worden als een “levend wezen”.

```
public abstract class LivingBeing
{
	public string SortName { get; private set;}
	public Position Position { get; private set;}


	// Constructor
	public LivingBeing(string sortName)
	{
		SortName = sortName;
	}

	public abstract void MoveToPoint(int x, int y, int z);
}
```

De abstracte methode `MoveToPoint` heeft geen body, omdat het nog niet duidelijk is hoe dat per soort wezen moet gebeuren – lopen, kruipen, vliegen, zwemmen. De child-classes van `LivingBeing` zijn verplicht om de abstracte methode te implementeren.

```
public class Snake : LivingBeing
{
	// Constructor
	public Snake(string sortName) : base(sortName)
	{}

	// Implementation of the abstract method
	public void MoveToPoint(int x, int y, int z)
	{
		Crowl(x, y, z);
	}

	// This method has a private access modifier
	// because it's not necessary to expose it to ouside
	// world.
	private void Crowl(int x, int y, int z)
	{
		Position = new Position { x, y, z };
	}
}
```

Men kan zich hier afvragen of het per se nodig is dat een slang op een Z (hoogte) coördinaat kruipt. Denk groot, het kan zijn dat die in een groot terrarium woont op de 13de etage van een gebouw. Of iets totaal ondenkbaars. Ik refereer naar de Liskov substitusion principle bescreven door Robert C. Martin (zie **SOLID** principles).

```
public class Bird : LivingBeing
{
	public string SortName { get; private set; }

	// Constructor
	public Bird (string sortName) : base(sortName)
	{}

	// Implementation of the abstract method
	public void MoveToPoint(int x, int y, int z)
	{
		Fly(x, y, z);
	}

	private void Fly(int x, int y, int z)
	{
		Position = new Position { x, y, z };
	}
}
```

```
public class Position
{
	public int X { get; set; }
	public int Y { get; set; }
	public int Z { get; set; }
}
```

4.2 Interfaces

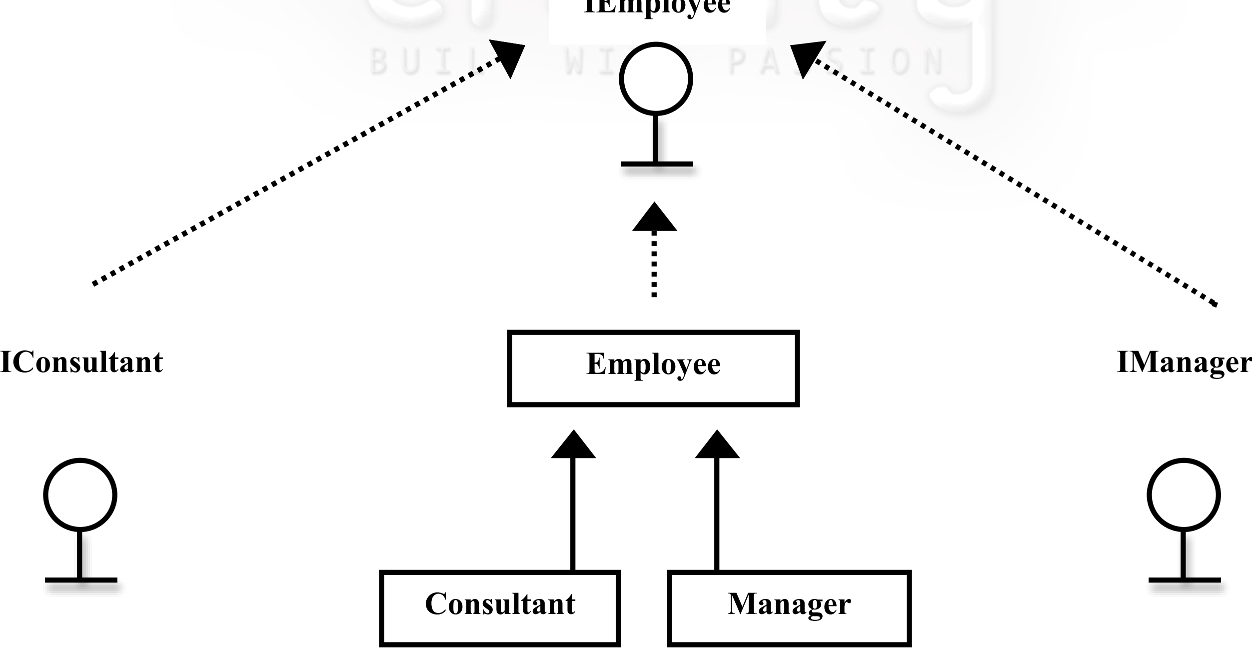
Interfaces zijn abstracte structuren, waarmee een contract gedefinieerd wordt. Interfaces zijn nog abstracter dan abstract classes, omdat abstract classes naast abstracte methodes, ook concrete methodes kunnen hebben, zoals eerder gezegd.

Interfaces worden gebruikt als contracten tussen objecten. In talen waar geen multiple inheritance geïmplementeerd is worden veel interfaces gebruikt.

Interfaces kunnen nooit een instantie zijn, vandaar dat ze geen constructor hebben. Hun members (methods en properties) zijn abstract.

Een iterface heeft geen betekenis als deze niet geimplementeerd wordt door een concrete class.

De methodes moeten altijd geimplementeerd woden door de class die deze interface implementeert.



Volgens de naamconventies in C# beginnen interfacenamen met een “I” (voor “interface”). Interfaces kunnen van elkaar “erven” om een contract op het generieke

geval te kunnen definieren. IConsultant en IManager zijn interfaces die van IEmployee overerven.

```
public interface IEmployee
{
      string UniqueNumber { get; set; }
      double AnnualSalary { get; set; }
      IManager Manager { get; set; }

      void DoYourJob();
      void Promote();
      void Resign();
}

public interface IConsultant : IEmployee
{
      string Expertise { get; set; }
}

public interface IManager : IEmployee
{
      IList<IEmployee> Employees { get; set; }
}
```

Zowel de consultant als de manager hebben een manager (beschreven in IEmployee). De consultant heeft als specialiteit bepaalde expertise. De manager werkt met mensen om het voorbeeld simpel te houden.

BUILT WITH PASSION

```
public class Employee : IEmployee
{
      public string UniqueNumber { get; set; }
      public double AnnualSalary { get; set; }
      public IManager Manager { get; set; }

      public Employee(string number, double salary, IManager
manager)
      {
            UniqueNumber = number;
            AnnualSalary = salary;
            Manager = manager;
      }

      public virtual void DoYourJob()
      {
            // do something generic, but let
            // children fill it in themselves

      }

      public void Promote(double raise)
      {
            AnnualSalary = raise;
      }
```

```
	public void Resign()
	{
		Manager = null;
		AnnualSalary = 0.0;
	}
}

public class Consultant : Employee, IConsultant
{
	public override void DoYourJob()
	{
		// do own implementation
	}
}

public class Manager : Employee, IManager
{
	IList<IEmployee> Employees { get; set; }

	public override void DoYourJob()
	{
		// do own implementation
	}
}
```

Manager erft de geimplementeerde members van Employee (zoals beschreven in IEmployee) en is van willekeurige client code aanspreekbaar via zijn interface IManager.

```
public class Program
{
	static void Main(string[] args)
	{

		// simple test
		var superManager = new Manager(“NLA20234”, 100000,
	null);
		var manager = new Manager(“NLA20234”, 100000,
	superManager);

		var employee1 = new Consultant(“NLA20120”, 100000,
	manager);
		var employee2 = new Consultant(“NLA20899”, 100000,
	manager);

		manager.Employees.Add(employee1);
		manager.Employees.Add(employee2);

	foreach (var employee in manager.Employees)
	{
		employee.DoYourJob();
	}

}
```

Alle `Consultant` an `Manager` types passen in de collectie `Employees` van het type `IEmployee`. Dit heet upcasting in de class hierarchie. Elke `Manager` en `Consultant` is `IEmployee`.

5 Polymorfism

Polymorfisme is een belangrijk concept en heeft te maken met de programmeertalen. Er zijn monomorfe talen (FORTRAN, C, COBOL) en polymorfe talen (C++, Java, C#). Bij de polymorfe talen hoeven de procedures, functies en operanden geen uniek type te hebben. Dat betekent dat een methode in een OO taal verschillende singaturen kan hebben (dus verschillend aantal argumenten).

Poly-morph betekent iets wat meerdere vormen kan nemen.

Polymorfisme maakt het mogelijk dat de levende objecten in run-time hun eigenaardigheden mogen uitvoeren op een uniform verzoek, bijvoorbeeld elke employee heeft eigen implementatie van `DoYourJob`:

```
public class Program
{
	static void Main(string[] args)
	{

		// simple test
		var superManager = new Manager("NLA20234", 100000,
	null);
		var manager = new Manager("NLA20234", 100000,
	superManager);

		var employee1 = new Consultant("NLA20120", 100000,
	manager);
		var employee2 = new Consultant("NLA20899", 100000,
	manager);

		manager.Employees.Add(employee1);
		manager.Employees.Add(employee2);

	foreach (var employee in manager.Employees)
	{
		employee.DoYourJob();
	}

}
```

Craftly BUILT WITH PASSION

5.1 Overloading van methodes, operatoren

Method-overloading betekent dat men al een bestaande methode herdefinieert, waarbij dezelfde naam voor methode gebruikt wordt maar met verschillende return types en aantal argumenten. Het volgende voorbeeld is van constructor overloading als een vaak gebruikte techniek.

```
public class Employee : IEmployee
{
      // members…

      // Default constructor - no arguments
      public Employee()
      {
      }

      // Overloaded constructor - with arguments
      public Employee(string number, double salary, IManager
manager)
      {
            UniqueNumber = number;
            AnnualSalary = salary;
            Manager = manager;
      }
}
```

6 Specialisatie en generalisatie

Het paragraaf over Inheritance geeft een voorbeeld over specialisatie en generalisatie. Om bepaalde object-eigenschappen in de Universe of Discourse te generalizeren worden superclasses (parent classes / base classes) geschreven. Subclasses (child classes) bevatten zijn de specializatie van eigenschappen en gedrag.

7 SOLID Principles door uncle Bob (Robert C. Martin)

S	The Single Responsibility Principle	*A class should have one, and only one, reason to change.*
O	The Open Closed Principle	*You should be able to extend a classes behavior, without modifying it.*
L	The Liskov Substitution Principle	*Derived classes must be substitutable for their base classes.*
I	The Interface Segregation Principle	*Make fine grained interfaces that are client specific.*
D	The Dependency Inversion Principle	*Depend on abstractions, not on concretions.*

BUILT WITH PASSION

8 Bronnen

OO paradigm in Wikipedia	http://en.wikipedia.org/wiki/Object-oriented_programming
OO polymorphism concept	http://www.codeproject.com/Articles/34125/Chapter-10-Object-Oriented-Programming-Polymorphis
Video: Object Oriented programming, part 1	http://www.youtube.com/watch?v=qbUJXsKAtU0
C#: abstract class versus interface	http://www.codeproject.com/Articles/11155/Abstract-Class-versus-Interface
The SOLID principles of Robert C. Martin (Uncle Bob)	http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
Principles of OOD	http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod