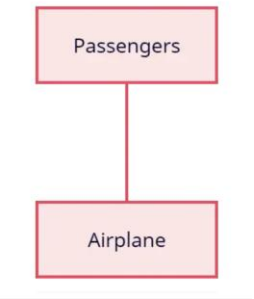
Association



- Een association geeft een algemene relatie tussen twee klassen aan.
- De ene klasse bevat een verwijzing naar een andere klasse.
- Er is sprake van samenwerking, maar geen eigendom of levenscyclus-koppeling.
- Beide klassen kunnen onafhankelijk bestaan.

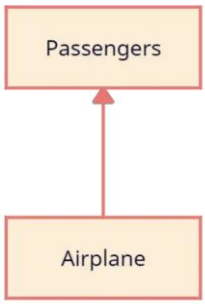
Voorbeeld in concept:
Een **Airline** heeft passagiers (**Passenger**). Ze zijn met elkaar verbonden, maar de passagier leeft los van de luchtvaartmaatschappij.

```
public class Passenger
{
    public string Name { get; set; }
}

public class Airline
{
    public List<Passenger> Passengers { get; set; } = new List<Passenger>();
}
```

Toegang: Airline heeft toegang tot Passenger-objecten via een lijst, maar bezit ze niet strikt.

Directed Association



- Een gerichte relatie: slechts één klasse wijst naar de andere.
- De pijl geeft de richting van afhankelijkheid aan.
- De klasse met de pijl "kent" of gebruikt de andere klasse.

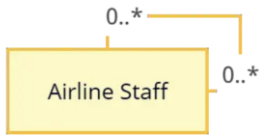
Voorbeeld in concept:
Een **Order** verwijst naar een **Customer**, maar een Customer weet niets van zijn orders.

```
public class Customer
{
    public string Name { get; set; }
}

public class Order
{
    public Customer Customer { get; set; }
}
```

Toegang: Order kent Customer, maar niet andersom. Directionele afhankelijkheid.

Reflexive Association



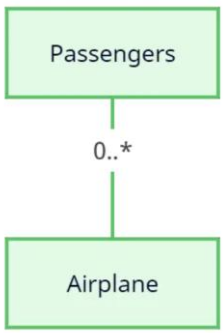
- Een klasse heeft een relatie met een ander object van dezelfde klasse.
- Wordt vaak gebruikt bij hiërarchieën of zelfbeheerstructuren.
- Beide objecten zijn van hetzelfde type maar hebben verschillende rollen.

Voorbeeld in concept:
Een **Employee** kan een manager hebben die ook een Employee is.

```
public class Employee
{
    public string Name { get; set; }
    public Employee Manager { get; set; }
}
```

Toegang: Een Employee-object verwijst naar een ander Employee-object — zichzelf als hiërarchie.

Multiplicity



- Geeft aan hoeveel objecten van de ene klasse gekoppeld zijn aan een ander object.
- Wordt aangeduid met notatie zoals 1, 0..1, 0..*, 1..*, enzovoort.
- Beschrijft kwantitatieve relaties in een associatie.

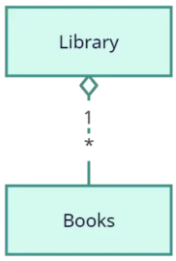
Voorbeeld in concept:
Een **Fleet** heeft meerdere **Airplanes**, maar elk Airplane hoort bij exact één Fleet.

```
public class Airplane
{
    public string Model { get; set; }
}

public class Fleet
{
    public List<Airplane> Airplanes { get; set; } = new List<Airplane>();
}
```

Toegang: Fleet heeft toegang tot een lijst van Airplanes, met een 1:N (één-op-meerdere) multipliciteit.

Aggregation



- "Heeft-een" relatie zonder eigendom.
- Kinderen (de componenten) kunnen blijven bestaan als de ouderklasse verdwijnt.
- Zwakke koppeling in vergelijking met composition.

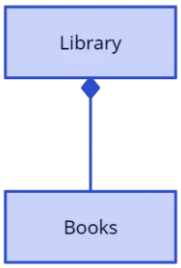
Voorbeeld in concept:
Een **Library** bevat meerdere **Books**. Als de bibliotheek sluit, blijven de boeken bestaan.

```
public class Book
{
    public string Title { get; set; }
}

public class Library
{
    public List<Book> Books { get; set; } = new List<Book>();
}
```

Toegang: Library bevat Books, maar beheert hun levenscyclus niet. Geen eigendomsrelatie.

Composition



- Sterke vorm van aggregatie met eigendom.
- Als de containerklasse verdwijnt, verdwijnen de componenten ook.
- De levenscyclus van de onderdelen is direct gekoppeld aan de hoofdklasse.

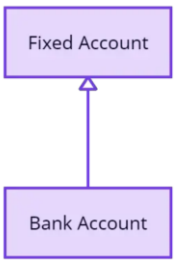
Voorbeeld in concept:
Een **ShoulderBag** heeft een **SidePocket**. Als de tas vernietigd wordt, is het zakje ook weg.

```
public class SidePocket
{
    public string Size { get; set; }
}

public class ShoulderBag
{
    private SidePocket pocket = new SidePocket();
}
```

Toegang: ShoulderBag bezit de SidePocket volledig. Levenscyclus is gekoppeld.

Inheritance / Generalization / Extends



- Een subklasse **erft** eigenschappen en methodes van een superklasse.
- Ondersteunt hergebruik en polymorfisme.
- Subklasse **is een** type van de superklasse.

Voorbeeld in concept:
Een **Dog** is een **Animal**. Hij kan de maakgeluid methode overnemen en hem anders uitvoeren.

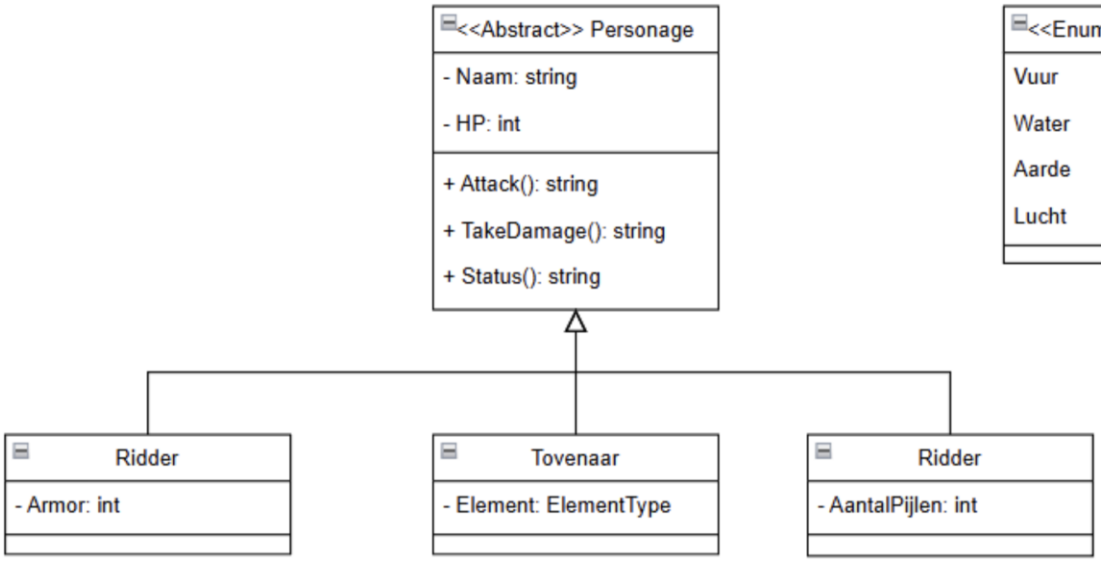
```
public class Dier
{
    public string Naam { get; set; }

    // Virtuele methode: kan overschreven worden
    public virtual void MaakGeluid()
    {
        Console.WriteLine("Dit dier maakt een geluid.");
    }
}

public class Hond : Dier
{
    // Override: eigen gedrag voor een hond
    public override void MaakGeluid()
    {
        Console.WriteLine("De hond blaft: Woef!");
    }
}
```

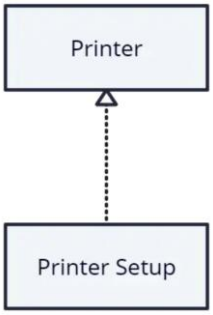
Toegang: Subklasse Dog krijgt alles van Animal.

Een ander voorbeeld:



De game applicatie, waar de Ridder, Tovenaar en Boogschutter alle **fields en methodes** overnemen. Tovenaar (als voorbeeld) bevat dus alle fields en methodes van Personage **+** Element. Deze methodes hoeven in de subclasses niet noodzakelijk aangepast te worden. Ridder **override de TakeDamage methode wel**, omdat die door zijn armor op een andere manier damage pakt. De Tovenaar en Boogschutter **implementeren die methode niet**, gezien die de standaard TakeDamage van Personage kunnen gebruiken. Het overriden van methodes uit de hoofdklasse is optioneel!

Realization



- Een klasse implementeert een interface, oftewel: een contract van gedrag.
- De interface beschrijft alleen wat moet gebeuren, de klasse beschrijft hoe.
- Interface biedt abstractie, implementatie voegt concrete logica toe.

Voorbeeld in concept:
Een **Printer** implementeert de interface **IPrinterSetup**, die een configuratiemethode vereist.

```
public class Animal
{
    public void Eat() => Console.WriteLine("Eating...");
}

public class Dog : Animal
{
    public void Bark() => Console.WriteLine("Woof!");
}
```

Toegang: Printer belooft dat hij alles uitvoert wat IPrinterSetup definieert.