UML

```
/**
 * @file uml.md
 * @author Rik Teerling
 * @date 2024/01
*/
```

Inhoudsopgave

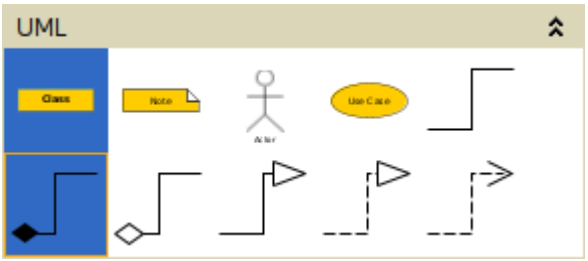
Class Diagrams

Je zult aan mede-programmeurs of eindgebruikers duidelijk moeten kunnen maken welke classes er in je applicatie bestaan, en hoe deze classes zich onderling verhouden.

Eén van de hulpmiddelen die je daarbij hebt is UML (Unified Modeling Language). Eén van de diagrammen die je kunt maken in UML is een 'klassediagram' (vanaf nu: 'class diagram').

yEd

Je kunt UML diagrammen maken in verschillende software paketten. Als je yEd gebruikt, kies de juiste tab UML in het Palette.



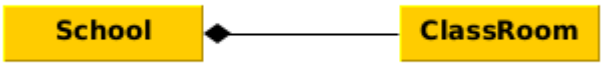
Relaties tussen classes

Composition

Composition beschrijft de relatie: "Een object *HEEFT* een een instance van een andere class".

Voorbeeld: School - ClassRoom.

In een classdiagram in UML wordt een 'composition' relatie aangegeven met een *dicht ruitje*. Het ruitje zit aan de class met het keyword 'new'.




```
class ClassRoom
{
    // fields, properties, constructor, methods
}

class School
{
    // fields
    private List<ClassRoom> classRooms;

    // constructor
    public School(int amount) // amount of classRooms
    {
        classRooms = new List<ClassRoom>();
        for (int i=0; i<amount; i++) {
            classRooms.Add(new ClassRoom()); // <-- NEW!
        }
    }
}
```

Op het moment dat het schoolgebouw wordt gebouwd, worden ook de ClassRooms gemaakt. Zie hiervoor in de constructor van School het keyword 'new' voor de ClassRooms. Een school *HEEFT* ClassRooms.

Is er geen School(gebouw), dan zijn er ook geen Lokalen. Als de school wordt afgebroken, verdwijnen ook de klaslokalen. ClassRooms kunnen niet bestaan, of hebben geen betekenis zonder School. Er staat nooit een ClassRoom ergens, zonder dat er een Schoolgebouw omheen zit.

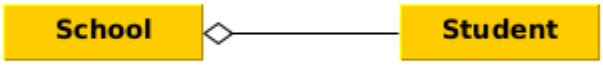
Terug naar boven

Aggregation

Aggregation beschrijft de relatie: "Een object *REFEREERT AAN* een ander object".

Voorbeeld: School - Student.

In een classdiagram in UML wordt een 'aggregation' relatie aangegeven met een *open ruitje*. Het ruitje zit aan de class die refereert aan de andere instance.



Studenten kunnen op zichzelf prima bestaan zonder School. Ze bestaan binnen School alleen administratief (een lijst met student-nummers en namen). Studenten worden buiten de School aangemaakt met het keyword 'new'. Bijvoorbeeld in een class 'World', waar ook de School wordt gemaakt met 'new'.

Als de School vind dat ze te weinig studenten heeft, zal de School goed onderwijs moeten bieden, en daar reclame voor moeten maken. Een School kan niet zelf met het keyword 'new' nieuwe Studenten maken.

Als de School wordt opgeheven (delete), zijn er nog steeds Studenten. Die melden zich dan aan op een andere School.

```
class Student
{
    // fields, properties, constructor, methods
}

class School
{
    // fields
    private List<Student> students;

    // methods
    public EnrollStudent(Student s)
    {
        students.Add(s); // <-- Wel op de lijst, GEEN new!
    }
}
```

Terug naar boven

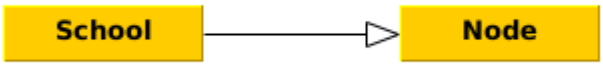
Inheritance

Inheritance beschrijft de relatie: "Een object *IS* een soort ander object".

Voorbeeld: School IS een Node (class School Extends Node). Ander voorbeeld: Dog IS an Animal (class Dog Extends Animal). Nog een voorbeeld: Student IS a Person.

Stel dat een School iets is dat een X,Y,Z positie heeft in een 3D wereld. Bijvoorbeeld omdat we een city-builder game maken. Denk aan SimCity, Tropico, of Cities: Skylines. We hebben een base-class in onze game-engine (GameObject, Actor, Node, Entity), en alle objecten in de wereld erven alle eigenschappen van deze base-class.

In een classdiagram in UML wordt een 'inheritance' relatie aangegeven met een *pijltje*. Het pijltje komt van de sub-class en wijst naar de base-class.



Inheritance wordt ook wel 'generalization' genoemd.

Als we onze class School laten overerven van Node, dan krijgen we alle functionaliteit uit de parent class Node gratis. Node heeft een Position, dus School heeft ook deze Position.

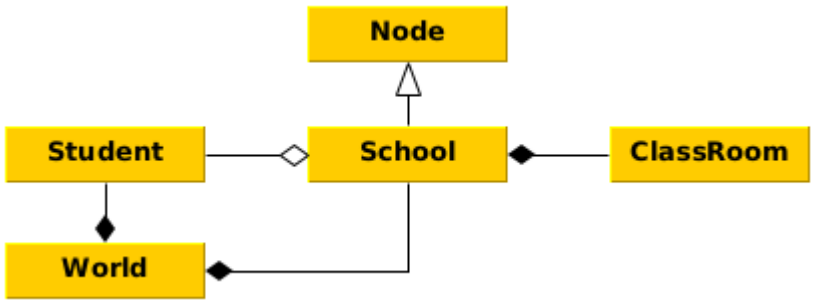
```
class Node
{
    // properties
    public Vector3 Position { get; set; }
    public Vector3 Rotation { get; set; }
    public Vector3 Scale    { get; set; }

    // constructor
    public Node()
    {
        Position = new Vector3(0, 0, 0);
        Rotation = new Vector3(0, 0, 0);
        Scale    = new Vector3(1, 1, 1);
    }
}

class School : Node
{
    // constructor : base() calls constructor of parent class
    public School() : base()
    {

    }
}
```

Terug naar boven

Samenvatting 1

We maken voor de volledigheid ook nog een class 'World', en vullen het class diagram aan met de classes en relaties die we tot nu toe hebben.

```
class World
{
    // constructor
    public World()
    {
        // A School with 3 ClassRooms
        School school = new School(3);

        // Some students
        Student joe = new Student();
        Student jane = new Student();

        // Enroll the Students
        school.Enroll(joe);
        school.Enroll(jane);
    }
}

// Instantiate everything
World earth = new World();
```



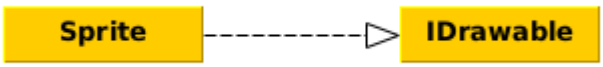
Terug naar boven

Realization

Realization wordt ook wel 'Implementation' genoemd. Voor deze relatie maken we gebruik van Interfaces, al ondersteunen sommige programmeertalen (c++ bijvoorbeeld) ook *multiple* inheritance.

De relatie: Een object *KAN* (meerdere) dingen (zijn).

In een classdiagram in UML wordt een 'realization' relatie aangegeven met een *stippel lijn met pijltje*. Het pijltje komt van de sub-class en wijst naar de base-class. Het open pijltje heeft wel een 'onderkantje'.



Een Interface is een contract met de buitenwereld, waarin je afspreekt dat je functionaliteit implementeert van de classes waarvan je wil over-erven. Op straffe van een compiler error.

Om duidelijk te maken dat de parent class een Interface is (en dus geen implementatie heeft) begint de class name met een hoofdletter 'I'.

Zo kan een Sprite zowel een Node, IMovable en IDrawable zijn.

```
class Node
{
    public Vector3 Position { get; set; }
}

interface IMovable
{
    Vector3 Velocity { get; set; }
    void Move();
}

interface IDrawable
{
    string ImagePath { get; set; }
    void Draw();
}

class Sprite : Node, IMovable, IDrawable
{
    // implementation of property Velocity (interface IMovable)
    public Vector3 Velocity { get; set; }
    // implementation of property ImagePath (interface IDrawable)
    public string ImagePath { get; set; }

    // implementation of Move() (interface IMovable)
    public void Move()
    {
        // Sprite extends Node, so we can move to 'Position'
        Position += Velocity;
```

```
    }

    // implementation of Draw() (interface IDrawable)
    public void Draw()
    {
        // Sprite extends Node, so we can draw an Image at 'Position'
        Renderer.RenderImage(Position, ImagePath);
    }
}
```

Voor de volledigheid, in java is het:

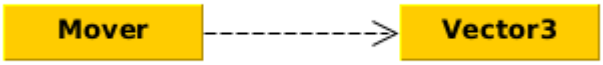
```
class Sprite extends Node implements IMovable, IDrawable
{

}
```

Terug naar boven

Dependancy

De relatie: Een object is *AFHANKELIJK* van een ander object.

In een classdiagram in UML wordt een 'dependancy' relatie aangegeven met een *stippel lijn met open pijltje*. Het pijltje komt van de dependant en wijst naar de dependancy. Het open pijltje heeft geen 'onderkantje'.



Een dependancy relatie leg je als een class afhankelijk is van een andere class.

De struct Vector3 is een 'ingebouwd' type in c# (System.Numerics.Vector3), maar zal er ongeveer zo uit zien:

```
struct Vector3
{
    // properties
    public float X;
    public float Y;
    public float Z;

    // constructor
    public Vector3(float x, float y, float z)
    {
        X = x;
        Y = y;
        Z = z;
    }
}
```

Deze hebben we best vaak nodig. Een dependancy heb je al vrij snel.

```
class Mover
{
    // Method Parameter Dependency (Vector3)
    Vector3 Seek(Vector3 target)
    {
        // Local Variable Dependency (Vector3)
        Vector3 desiredVelocity = normalize(Position - target) * MaxSpeed;
        Vector3 steer = desiredVelocity - Velocity;

        // Return Type Dependency (Vector3)
        return steer;
    }
}
```

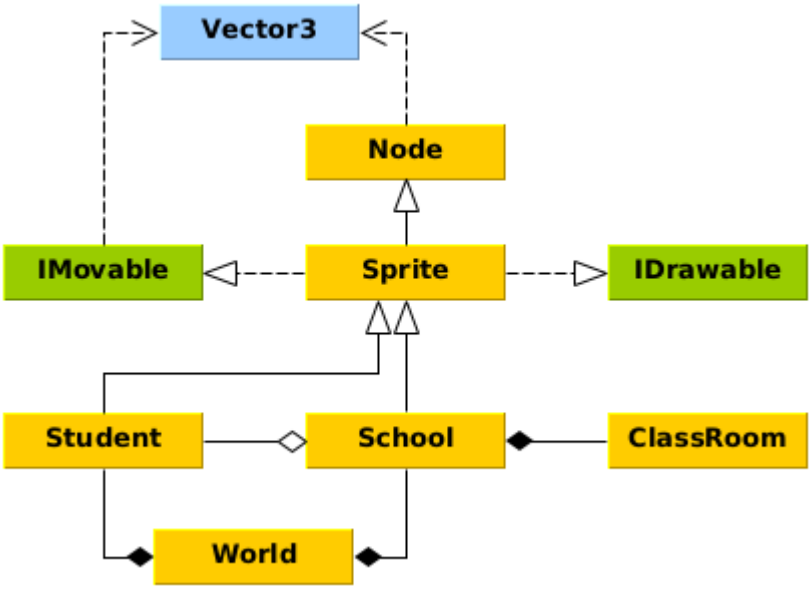
> Zorg dat je class diagram niet een kluwen van lijntjes wordt die allemaal naar Vector3 wijzen. Probeer niet ALLE relaties te willen leggen. Liever een duidelijk class diagram met alleen de belangrijkste relaties, dan een volledig, maar niet te begrijpen class diagram.

Samenvatting 2

Hierbij het volledige class diagram tot nu toe (inclusief Vector3). De interfaces zijn groen, om duidelijk aan te geven dat ze geen implementatie hebben. Vector3 is blauw om aan te geven dat het een ingebouwde struct in c# betreft. Hier zijn geen conventies voor. Als het maar duidelijk is.

Ondertussen hebben we zowel School als Student laten over-erven van Sprite, omdat we zowel de School als de Students willen tekenen op de Posities die ze hebben.

Dat betekent ook dat School een Move() method heeft. Da's een beetje gek, maar als de Velocity (0,0,0) is, blijft het schoolgebouw staan. We doen het er mee.



In de Class World kan nu een lijst bijgehouden worden van Nodes, waar zowel een School als een Student in kan. Het zijn beide instances van Sprite, en Sprites zijn Nodes (onderzoek 'Polymorphism').

```
class World
{
    // fields
    private List<Node> nodes;

    // constructor
    public World()
    {
        // Keep a list of all Nodes in the World
        nodes = new List<Node>();

        // A School with 3 ClassRooms, somewhere in the World
        School school = new School(3);
        school.Position = new Vector3(42, 138, 2); // Position in Node
        // Add school to node-list
```

```
        nodes.Add(school);

        // Some students
        Student joe = new Student();
        Student jane = new Student();

        // Add students to node-list
        nodes.Add(joe);
        nodes.Add(jane);

        // Enroll the Students
        school.Enroll(joe);
        school.Enroll(jane);
    }

    public Update(float deltaTime)
    {
        foreach (Node node in nodes)
        {
            // Update each node
        }
    }
}
```

Terug naar boven