

League	2
Beschrijving project	2
Domein	2
Speler.....	2
Team.....	2
Transfer	2
Architectuur.....	3
Technologie	4
Projectstructuur	4
Applicatieontwikkeling – cyclus 1.....	5
Domein	5
Ontwerp.....	5
Implementatie.....	5
Speler.....	5
Team	8
Relatie Team – Speler.....	9
Transfer	11
Unit Test	14
UnitTestSpeler	15
Databank model	20
Anayse user stories.....	22
User story : voeg een speler toe aan het systeem	23
User story : voeg een team toe aan het systeem.....	29
User story : selecteer een team op basis van zijn stamnummer.....	33
User story : voeg een transfer toe aan het systeem.	38
User story : pas de eigenschappen van een speler aan.	41
User story : pas de eigenschappen van een team aan.....	43
Stand van zaken project.	45
Applicatieontwikkeling – cyclus 2.....	46
UI-ontwerp.	47
Data Transfer Objects (DTO)	49
Selecteer spelers.	51
Selecteer teams.	53
User story : pas de eigenschappen van een speler aan.	54
User story : pas de eigenschappen van een team aan.....	61
User story : voeg een transfer toe aan het systeem.	64

Applicatieontwikkeling – cyclus 3.....	65
Hoofdvenster.....	66
Registreer spelers.....	67
Update spelers	71

League

Beschrijving project

Het league-project heeft tot doel om een desktopapplicatie te bouwen die het beheer van spelers en hun transfers kan afhandelen.

Domein

De entiteiten waarmee we te maken hebben zijn spelers, teams en transfers. Daarnaast hebben we ook het beheer van de teams en een spelermanager.

Voor de verschillende entiteiten gelden de volgende regels.

Speler

- Elke speler heeft een unieke Id, deze Id is numeriek en moet steeds groter zijn dan 0.
- Een speler heeft ook een naam die niet leeg mag zijn.
- Een speler heeft ook een lengte (in cm) die minstens 150 moet zijn, we beschikken niet altijd over deze gegevens.
- Een speler heeft ook een gewicht (in kg), minstens 50 en ook hier hebben we niet altijd de gegevens.
- Aan een speler wordt een rugnummer toegekend en dit nummer moet groter zijn dan 0 en kleiner dan 100. Rugnummers zijn niet altijd toegekend.
- Een speler behoort meestal tot een team.

Team

- Elk team heeft een uniek stamnummer (numeriek en positief).
- Een team heeft ook een naam en bijnaam. Een naam is verplicht, de bijnaam niet. Wanneer een naam of bijnaam wordt toegekend mag deze echter niet leeg zijn.
- Een team beschikt over een aantal spelers.

Transfer

- Wanneer we een speler willen transfereren naar een ander team dan wensen we dat te registreren en noemen we dat een transfer.
- Elke transfer wordt uniek geïdentificeerd door middel van een Id (numeriek en groter dan 0).

- Bij een transfer hoort dus altijd een speler, maar ook een prijs. Deze prijs wordt uitgedrukt in euros en kan niet negatief zijn (wel 0).
- Een transfer bevat zowel het oude/vorige team als het nieuwe team.
- Als een speler stopt dan registreren we dat ook als een transfer, waarbij we het nieuwe team niet invullen.
- Het toekennen van spelers die nog niet tot een team behoren doen we eveneens via een transfer, het oude team wordt dan niet ingevuld.
- Transfers gebeuren altijd tussen verschillende teams

Voor het beheer van de spelers gelden er eveneens een aantal regels :

- Zorg ervoor dat een speler maar 1 keer kan voorkomen.
- Spelers moeten kunnen worden toegevoegd aan het systeem.
- Spelers moeten ook kunnen worden verwijderd uit het systeem.
- Het moet ook mogelijk zijn om de eigenschappen van een speler aan te passen.
- Je moet een overzicht kunnen geven van alle spelers (zowel spelers die tot een team behoren als spelers die niet tot een team behoren).
- Je moet ook een overzicht kunnen geven van alle transfers die hebben plaats gevonden.

Net zoals voor het beheer van de spelers zijn er ook vereisten voor het beheer van de teams :

- Zorg ervoor dat een team maar 1 keer kan voorkomen.
- Teams moeten kunnen worden toegevoegd aan het systeem.
- Teams moeten ook kunnen worden verwijderd uit het systeem.
- Je moet een overzicht kunnen geven van alle Teams.
- Je moet ook een team kunnen opzoeken op basis van zijn stamnummer of naam.

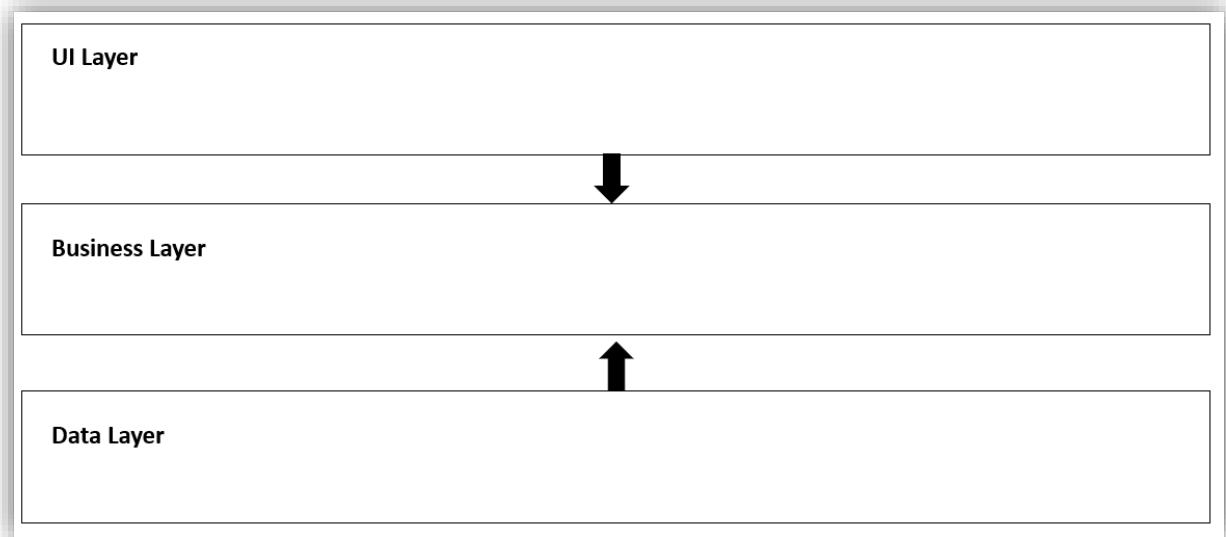
Architectuur

Het meest gekende/gebruikte architectuur-patroon is de gelaagde architectuur, ook wel n-tier architectuur genoemd.

In een gelaagde architectuur worden de componenten georganiseerd in een aantal horizontale lagen, waarbij elke laag een specifieke rol speelt. Eén van de voordelen van deze architectuur is de scheiding van de verantwoordelijkheden (separation of concerns) tussen de componenten. Voorbeeld : de business-laag is enkel verantwoordelijk voor de business-logica, de UI enkel voor de presentatie logica en de data-laag handelt het beheer van de gegevens af.

In deze architectuur zullen verzoeken zich enkel bewegen tussen opeenvolgende lagen en speelt het ook een rol in welke richting de lagen elkaar kennen. Klassiek gezien hadden we de UI-laag die de business-laag kent en de business-laag die de data-laag kent. Deze laatste link wordt echter vaak omgewisseld, omdat de business-laag meer en meer als de belangrijkste laag wordt gezien.

In het voorbeeld dat we hier gaan bespreken onderscheiden we de volgende lagen :



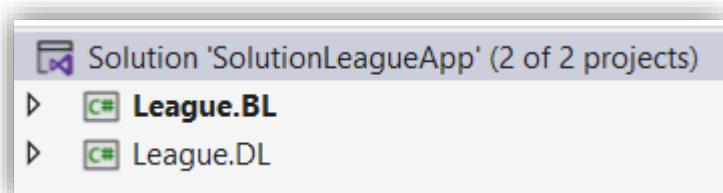
De business/domeinlaag zetten we centraal en daarin gaan we ook onze entiteiten definiëren. Verder laten we de UI-laag verwijzen naar de domeinlaag, met andere woorden de UI-laag kent de domeinlaag en kan dan ook objecten uit deze laag gaan gebruiken. De datalaag op zijn beurt kent eveneens de domeinlaag en is dus in staat om domein-entiteiten aan te leveren en op te slaan.

Technologie

De ontwikkeling zal gebeuren in C# (.NET 5.0) waarbij we gebruik maken van Visual Studio. Voor de Unit Tests gebruiken we xUnit, de databank is een SQL server en de communicatie met de databank gebeurt via ADO.NET. De UI zal gemaakt worden met WPF. De Ids voor de spelers en de transfers zullen door de databak worden aangemaakt (identity).

Projectstructuur

We kiezen voor een ‘blank solution’ en daarin zullen we de verschillende projecten toevoegen. Zowel voor de business laag als de datalaag gebruiken we een class library.

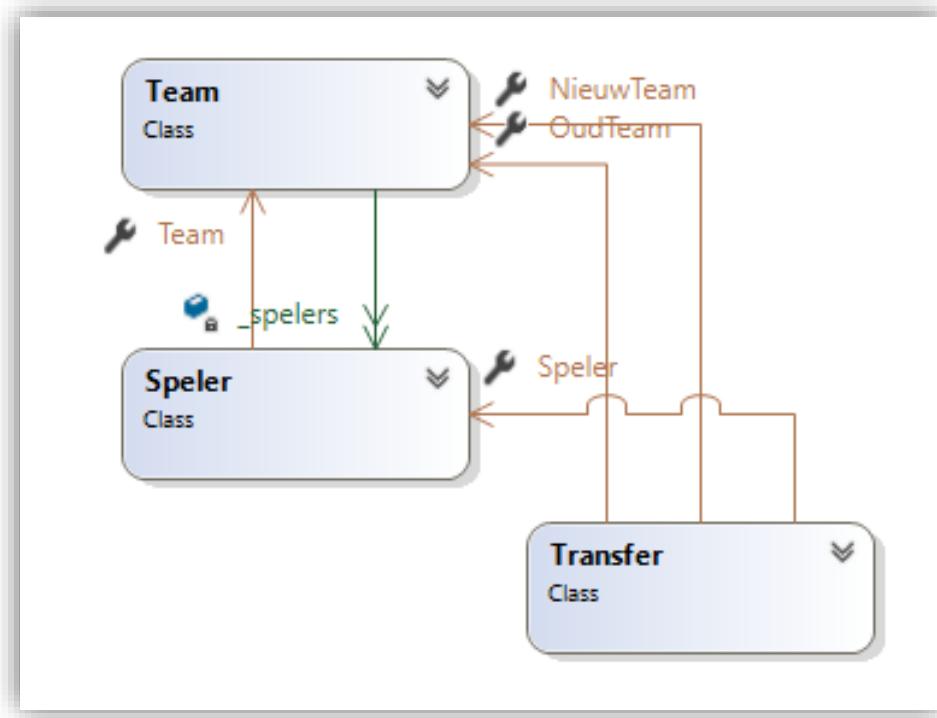


Applicatieontwikkeling – cyclus 1

Domein

Ontwerp

In het volgende diagram hebben we de verschillende entiteiten en hun relaties gemodelleerd. Een Team heeft een lijst van Spelers en een Speler kan tot een Team behoren. Verder hebben we een klasse Transfer die steeds verwijst naar een Speler, een mogelijk oud Team en een mogelijk nieuw Team (één van beiden moet minimaal worden opgegeven).



Implementatie

In dit hoofdstuk bekijken we nu elk van de klassen apart en zien we op welke manier de opgelegde requirements worden afgedwongen. De entiteiten die tot het domein behoren plaatsen we in de business-laag en we voorzien een aparte folder 'Domein' om deze klassen in onder te brengen.

Speler

De klasse Speler heeft als properties een Id, Naam, Team, Rugnummer, Lengte en Gewicht. De laatste drie zijn optioneel en dus voorzien we een nullable int (int?).

```
public int Id { get; private set; }
2 references
public string Naam { get; private set; }
10 references
public Team Team { get; private set; }
1 reference
public int? Rugnummer { get; private set; }
2 references
public int? Lengte { get; private set; }
2 references
public int? Gewicht { get; private set; }
```

Aan elk van deze properties zijn er voorwaarden (requirements) verbonden en om deze af te dwingen maken we de setters private en voorzien we telkens een methode waarin we die voorwaarden controleren.

De naam mag niet leeg zijn en dus zullen we bij het instellen een exception gooien als dit wel het geval zou zijn. We kiezen ervoor om een custom exception te gooien zodat we duidelijk kunnen aangeven waar de exception vandaan komt. We kiezen er ook voor om de (eventuele) spaties voor en achter de naam automatisch te verwijderen.

```
public void ZetNaam(string naam)
{
    if (string.IsNullOrWhiteSpace(naam)) throw new SpelerException("ZetNaam");
    Naam=naam.Trim();
}
```

De exception ziet er als volgt uit :

```
public class SpelerException : Exception
{
    7 references
    public SpelerException(string message) : base(message)
    {
    }
    0 references
    public SpelerException(string message, Exception innerException) : base(message, innerException)
    {
    }
}
```

En we plaatsen al de custom exceptions die we maken in onze business-laag eveneens in een aparte folder.

Ook voor de lengte, id, gewicht en rugnummer zijn er voorwaarden, de implementatie ziet er dan als volgt uit :

```

public void ZetLengte(int lengte)
{
    if (lengte<150) throw new SpelerException("ZetLengte");
    Lengte = lengte;
}
1 reference
public void ZetId(int id)
{
    if (id <= 0) throw new SpelerException("ZetId");
    Id = id;
}
1 reference
public void ZetGewicht(int gewicht)
{
    if (gewicht < 50) throw new SpelerException("Zetgewicht");
    Gewicht=gewicht;
}
0 references
public void ZetRugnummer(int rugnummer)
{
    if ((rugnummer <= 0) || (rugnummer > 99)) throw new SpelerException("ZetRugnummer");
    Rugnummer = rugnummer;
}

```

We hebben ook nog twee constructors nodig, een eerste constructor met een minimum aan instellingen, namelijk een naam, lengte en gewicht (met de laatste twee die null mogen zijn). En een tweede constructor die ook nog een id instelt. We voorzien beide constructors omdat het de bedoeling is dat de databank de IDs gaat genereren, met andere woorden we moeten een object reeds kunnen aanmaken voor dat we een id hebben ontvangen en we moeten een object ook kunnen instantiëren na het lezen uit de databank (met id).

In de constructor controleren we ook de twee nullable velden of ze effectief een waarde hebben, enkel als er een waarde is meegegeven roepen we onze instelmethoden (ZetXxx) op (anders zou dit een exception opleveren). Opmerking : kijken of een waarde is ingesteld kan zowel door het vergelijken met null als met de HasValue methode (beiden zijn hier weergegeven).

Wat valt er nog op ? Onze constructors hebben als access modifier ‘internal’ meegekregen en dat wil zeggen dat ze enkel binnen de assembly (class library) kunnen worden gebruikt. Waarom doen we dit ? We willen niet dat een applicatie die onze library gebruikt zomaar een Speler object kan aanmaken. Enkel spelers die in het systeem zijn opgenomen kunnen gebruikt worden, we zullen verder zien op welke manier we dit aanpakken.

```

internal Speler(int id, string naam, int? lengte, int? gewicht) : this(naam,lengte,gewicht)
{
    ZetId(id);
}
1 reference
internal Speler(string naam, int? lengte, int? gewicht)
{
    ZetNaam(naam);
    if (lengte != null) ZetLengte(lengte.Value);
    if (gewicht.HasValue) ZetGewicht(gewicht.Value);
}

```

Team

De klasse Team heeft als properties een stamnummer, een naam en bijnaam. Ook hier zijn er voorwaarden aan verbonden en kiezen we dus op de setters private te houden. Daarnaast heeft een team ook een lijst van spelers.

```
public class Team
{
    4 references
    public int Stamnummer { get; private set; }
    1 reference
    public string Naam { get; private set; }
    1 reference
    public string Bijnaam { get; private set; }
    private List<Speler> _spelers = new List<Speler>();
```

De voorwaarden dwingen we af door middel van de volgende methodes :

```
public void ZetStamnummer(int stamnummer)
{
    if (stamnummer <= 0) throw new TeamException("ZetStamnummer");
    Stamnummer = stamnummer;
}
1 reference
public void ZetNaam(string naam)
{
    if (string.IsNullOrWhiteSpace(naam)) throw new TeamException("ZetNaam");
    Naam = naam.Trim();
}
0 references
public void ZetBijnaam(string bijnaam)
{
    if (string.IsNullOrWhiteSpace(bijnaam)) throw new TeamException("ZetBijnaam");
    Bijnaam = bijnaam.Trim();
}
```

Opmerking : zowel voor naam als bijnaam kiezen we ervoor om de eventuele spaties voor en achter te verwijderen.

Er wordt ook een constructor voorzien die het stamnummer en de naam instelt (de bijnaam laten we er uit omdat deze optioneel is).

```
internal Team(int stamnummer, string naam)
{
    ZetStamnummer(stamnummer);
    ZetNaam(naam);
}
```

Voor het beheer van de spelers in een team zijn er ook voorwaarden gesteld, het mag bijvoorbeeld niet mogelijk zijn dat een speler tweemaal voorkomt. Over de relaties tussen speler en team en e manier waarop we dit implementeren gaan we dieper in het volgende stuk. Wat we hier wel nog

meegeven is een methode om de lijst van spelers op te vragen. Aangezien we methodes zullen voorzien om spelers toe te voegen of te verwijderen zorgen we ervoor dat we een `ReadOnlyList` van spelers terug geven. Dat zorgt er voor dat de lijst niet aanpasbaar is.

```
public IReadOnlyList<Speler> Spelers()
{
    return _spelers.AsReadOnly();
}
```

Relatie Team – Speler

We geven wat extra aandacht aan de relatie tussen de klassen Speler en Team. De klasse Speler verwijst via een property naar een Team en een Team beschikt over een lijst van spelers.

In de klasse Team voorzien we de volgende methodes :

```
internal void VerwijderSpeler(Speler speler)...
1 reference
internal void VoegSpelerToe(Speler speler)...
```

waarmee we de spelers kunnen beheren.

In de klasse Speler maken we het ook mogelijk om een team toe te wijzen aan een speler of een team te verwijderen.

```
internal void VerwijderTeam()...
1 reference
internal void ZetTeam(Team team)...
```

De beheerklassen die gebruik maken van ons domein kunnen dus op verschillende manieren de relatie tussen de klassen gaan beheren.

Wij moeten er nu wel voor zorgen dat wanneer één van deze acties wordt uitgevoerd de consistentie blijft behouden. Wat wil dit zeggen ? Als we in de klasse Team een speler toevoegen dan moet deze niet alleen worden opgenomen in de lijst van spelers (van team) maar dan moet ook de property Team in de klasse Speler correct worden ingesteld. We hebben hier een relatie die in twee richtingen loopt en die moet dus ook onderhouden worden. Omgekeerd als we de property Team in de klasse Speler aanpassen dan moeten we er voor zorgen dat de speler uit de lijst van zijn oud team wordt verwijderd en aan de lijst van het nieuwe team wordt toegevoegd.

Laten we starten in de klasse Team. De methode `VerwijderSpeler` moet dus de speler uit de lijst van spelers halen en moet de property `Team` van speler op null zetten. Alvorens we dit uitvoeren doen we nog een aantal controles : (1) is de meegegeven speler null dan gooien we een exception, (2) vragen we om een speler te verwijderen die niet in onze lijst zit dan gooien we eveneens een exception. Om te controleren of een speler in de lijst zit gebruiken we de methode `Contains`.

Opmerking : het gebruik van contains moet je er altijd doen aan denken dat je de Equals-methode (in spelers) zal moeten overschrijven !

Het verwijderen van een speler uit de lijst kan eenvoudigweg met de methode Remove. Daarna zetten we de property team van speler gelijk aan null en dit laatste doen we met de methode VerwijderTeam uit de klasse spelers.

We hebben nu onze logica opgebouwd startende vanuit de klasse Team, maar we zouden ook kunnen starten vanuit de klasse spelers en daar de methode VerwijderTeam oproepen. Omdat beide methoden naar elkaar verwijzen gaan we op die manier in een oneindigelus van verwijzingen terechtkomen en uiteindelijk een stack overflow error genereren. Om het heen-en-weer verwijzen te stoppen moeten we in beide methodes een stop-controle toevoegen. Voor VerwijderSpeler kan dit door te stellen dat we enkel de VerwijderTeam methode gaan oproepen als de team property van spelers nog staat ingesteld op het team-object waarin we ons nu bevinden (this).

In de klasse spelers passen we de VerwijderTeam methode aan door enkel maar de verwijderspeler methode op te roepen als de speler nog tot het team behoort.

Voor het toevoegen van een speler is de logica volledig analoog.

```
internal void VerwijderSpeler(Speler spelers)
{
    if (speler == null) throw new TeamException("Verwijderspeler");
    if (!_spelers.Contains(speler))
    {
        throw new TeamException("Verwijderspeler");
    }
    _spelers.Remove(speler);
    if (speler.Team==this)
        spelers.VerwijderTeam();
}
1 reference
internal void VoegSpelerToe(Speler spelers)
{
    if (speler == null) throw new TeamException("VoegspelerToe");
    if (_spelers.Contains(speler)) throw new TeamException("VoegspelerToe");
    _spelers.Add(speler);
    if (speler.Team!=this)
        spelers.ZetTeam(this);
}
```

We gaan nog even dieper in op de methode ZetTeam (in spelers). Hier hebben we nog een controle die het nieuwe team vergelijkt met het reeds bestaande team, als beiden gelijk zijn wordt er een exception gegooid omdat er inhoudelijk niets veranderd (en dat stemt niet overeen met de bedoeling van de methode, namelijk het instellen van een nieuw team). Na deze controle kijken we of er een huidig team is en als dat het geval is vragen we aan dat Team of deze (this) speler daar toe behoort. Is dit het geval dan moet de speler uit dat Team worden verwijderd want hij gaat aan een ander team worden toegekend. Daarna vragen we aan het nieuwe team of deze speler er reeds toe behoort, is dat niet het geval dan moeten we deze speler toevoegen aan het nieuwe team.

```

internal void VerwijderTeam()
{
    if (Team.HeeftSpeler(this))
        Team.VerwijderSpeler(this);
    Team = null;
}
1 reference
internal void ZetTeam(Team team)
{
    if (team == null) throw new SpelerException("ZetTeam");
    if (team == Team) throw new SpelerException("ZetTeam");
    if (Team != null)
    {
        if (Team.HeeftSpeler(this))
            Team.VerwijderSpeler(this);
    }
    if (!team.HeeftSpeler(this)) team.VoegSpelerToe(this);
    Team = team;
}

```

Opmerking : er worden ook Teams vergeleken met elkaar dus ook hier de Equal-methode overschrijven.

Voor de volledigheid geven we nog de methode HeeftSpeler in de klasse Team.

```

public bool HeeftSpeler(Speler speler)
{
    return _spelers.Contains(speler);
}

```

Transfer

De klasse Transfer bevat naast een id en prijs ook een referentie naar een speler en verder is er ook minstens één van de properties NieuwTeam of OudTeam ingevuld.

```

public int Id { get; private set; }
1 reference
public Speler Speler { get; private set; }
4 references
public Team NieuwTeam { get; private set; }
4 references
public Team OudTeam { get; private set; }
1 reference
public int Prijs { get; private set; }

```

Het controleren van de id en de prijs kan met de volgende methodes :

```

public void ZetId(int id)
{
    if (id <= 0) throw new TransferException("ZetId");
    Id = id;
}
3 references
public void ZetPrijs(int prijs)
{
    if (prijs < 0) throw new TransferException("ZetPrijs");
    Prijs = prijs;
}

```

We voorzien ook methodes om de property OudTeam aan te passen. Aangezien bij een transfer er altijd minstens één team moet worden ingevuld controleren we of NieuwTeam wel verschilt van null als we het OudTeam uit zetten. Bij het instellen van het OudTeam controleren we of het oude team niet hetzelfde is als het nieuwe team (want dat mag natuurlijk niet).

```

public void VerwijderOudTeam()
{
    if (NieuwTeam is null) throw new TransferException("VerwijderOudTeam"); //minstens 1 team
    OudTeam = null;
}
2 references
public void ZetOudTeam(Team team)
{
    if (team == null) throw new TransferException("ZetOudTeam");
    if (team == NieuwTeam) throw new TransferException("ZetOudTeam");
    OudTeam = team;
}

```

Het instellen van NieuwTeam is volledig analoog met het voorgaande.

```

public void VerwijderNieuwTeam()
{
    if (OudTeam is null) throw new TransferException("VerwijderNieuwTeam"); //minstens 1 team
    NieuwTeam = null;
}
2 references
public void ZetNieuwTeam(Team team)
{
    if (team == null) throw new TransferException("ZetNieuwTeam");
    if (team == OudTeam) throw new TransferException("ZetNieuwTeam");
    NieuwTeam = team;
}

```

Ook hier geven we voor de volledigheid nog de methode ZetSpeler mee.

```

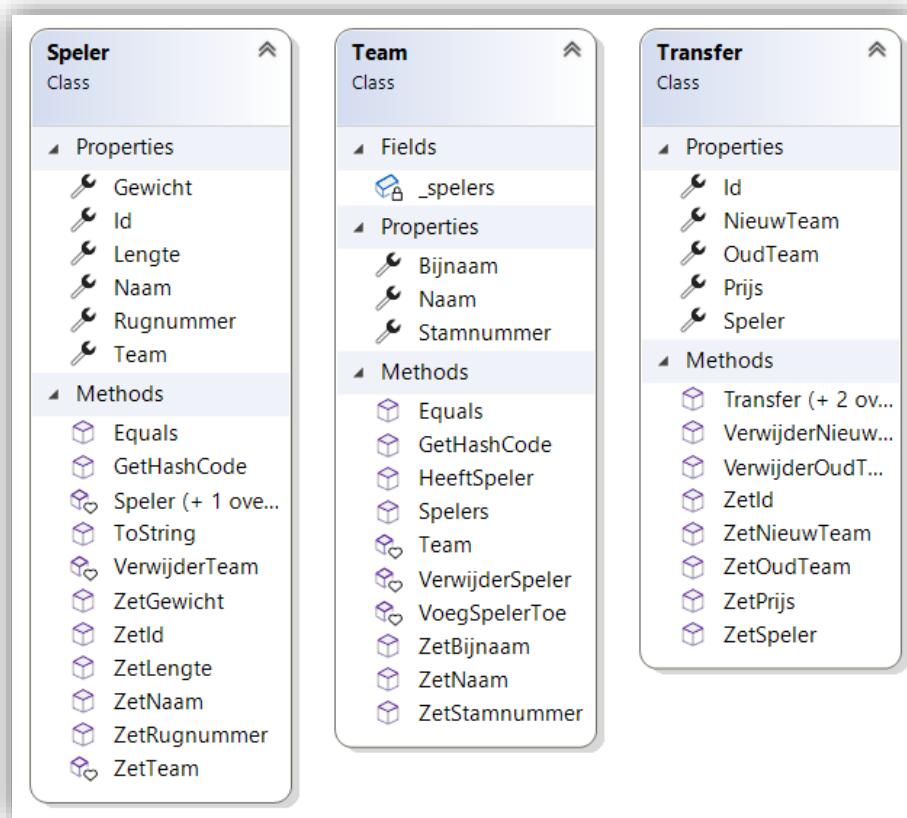
public void ZetSpeler(Speler spel)
{
    if (speler is null) throw new TransferException("ZetSpeler");
    Speler = spel;
}

```

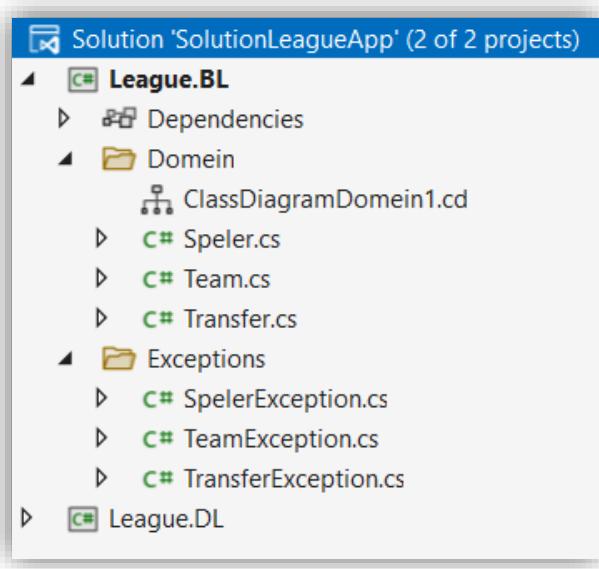
Dan zijn er ook nog de constructors voor transfer, daar onderscheiden we drie mogelijke scenario's, namelijk een transfer van een speler zonder team, een speler die stopt en de klassieke transfer tussen twee teams.

```
public Transfer(Speler speler, Team nieuwTeam, Team oudTeam, int prijs)
{
    ZetSpeler(speler);
    ZetNieuwTeam(nieuwTeam);
    ZetOudTeam(oudTeam);
    ZetPrijs(prijs);
}
// speler stopt
1 reference | 0 changes | 0 authors, 0 changes
public Transfer(Speler speler, Team oudTeam)
{
    ZetSpeler(speler);
    ZetOudTeam(oudTeam);
    ZetPrijs(0);
}
//speler is nieuw
1 reference | 0 changes | 0 authors, 0 changes
public Transfer(Speler speler, Team nieuwTeam, int prijs)
{
    ZetSpeler(speler);
    ZetNieuwTeam(nieuwTeam);
    ZetPrijs(prijs);
}
```

De uiteindelijk klassen zien er dan als volgt uit.



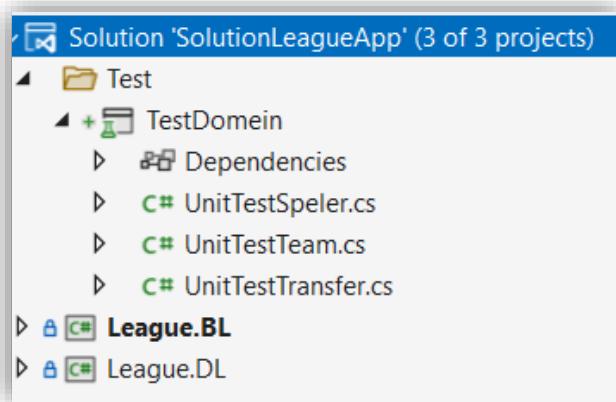
Onze solution ziet er voorlopig als volgt uit.



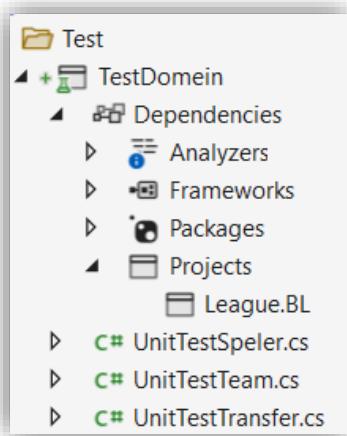
Unit Test

Om de code uit onze verschillende lagen duidelijk te onderscheiden van de testing, maken we een folder 'Test' aan in onze solution en daaraan voegen we een nieuw xUnit project aan toe.

Voor elke klasse uit ons domein maken we nu een unit test klasse aan, het resultaat ziet er dan als volgt uit.



Bij dependencies voegen we de link toe naar onze business-laag (League.BL) zodat onze domein klassen gekend zijn in ons test project.



We starten met de unit tests voor de klasse Speler.

UnitTestSpeler

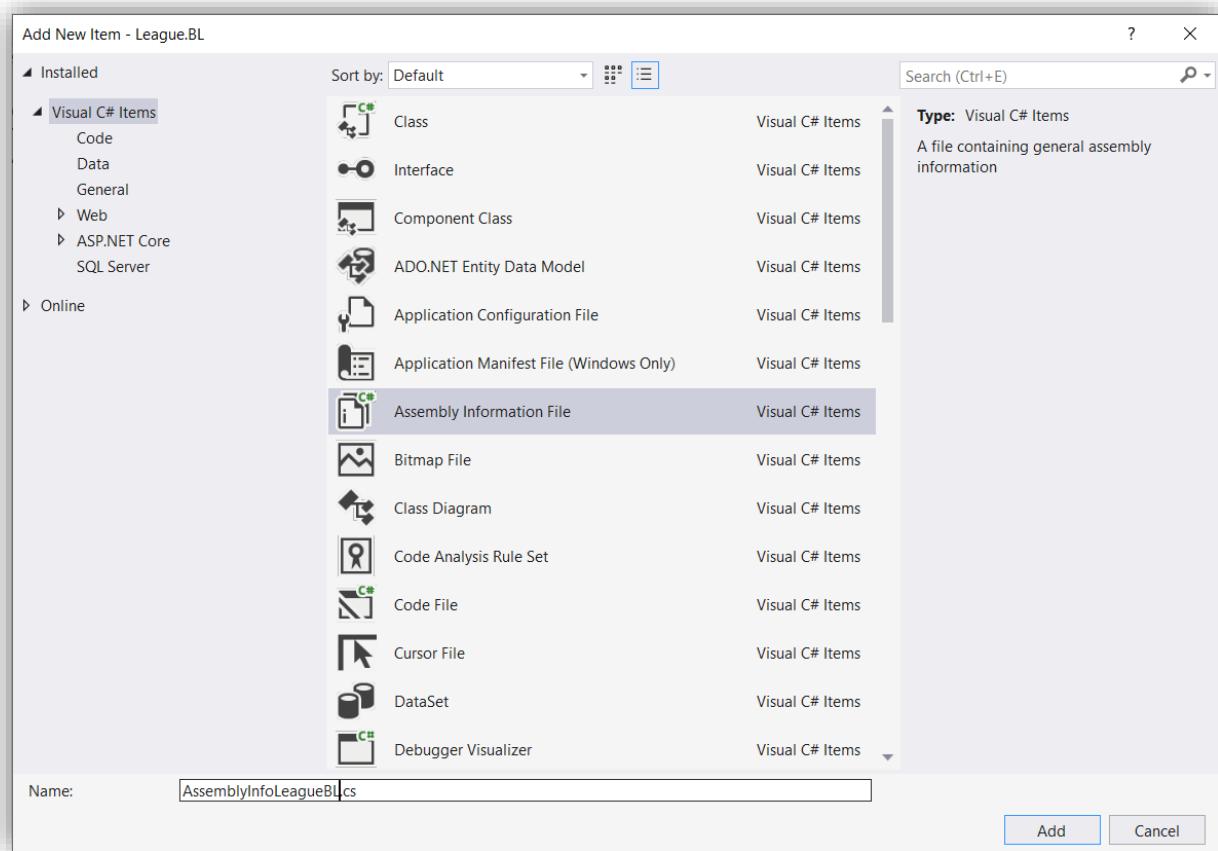
De eerste test die we wensen te schrijven is het ‘happy path’ voor de methode ZetId. We maken een speler aan met een bepaald id, controleren de waarde van de id, roepen dan de ZetId-methode op en verzekeren (assert) er ons van dat de nieuwe waarde is toegekend. Aangezien de toegestane range van waarden start vanaf 1 testen we dan ook deze grenswaarde.

```
public class UnitTestSpeler
{
    [Fact]
    public void ZetId_valid()
    {
        Speler s = new Speler(10, "Jos", 180, 80);
        Assert.Equal(10, s.Id);
        s.ZetId(1);
        Assert.Equal(1, s.Id);
    }
}
```

We stuiten echter op het volgende probleem :

```
class League.BL.Domein.Speler (+ 2 overloads)
CS0122: 'Speler.Speler(int, string, int?, int?)' is inaccessible due to its protection level
Show potential fixes (Alt+Enter or Ctrl+;)
```

De constructor van de klasse Speler heeft als access modifier internal en kan dus niet in een ander project worden opgeroepen. Daar is echter een mouw aan te passen als volgt.



In dit bestand voegen we dan de volgende lijn code aan toe :

[assembly: InternalsVisibleTo("TestDomein")]

Hier geven we aan dat het project "TestDomein" ook toegang krijgt tot de 'internal' methods van ons League.BL project (assembly).

Na deze aanpassing kunnen we onze test uitvoeren. Naast het valid-path voorzien we ook een invalid test die nagaat of de juiste exception wordt gegooid als we een niet toegestane waarde meegeven.

```
[Fact]
➊ | 0 references | 0 changes | 0 authors, 0 changes
public void ZetId_invalid()
{
    Speler s = new Speler(10, "Jos", 180, 80);
    Assert.Equal(10, s.Id);
    Assert.Throws<SpelerException>(() => s.ZetId(0));
    Assert.Equal(10, s.Id);
}
```

Voor rugnummer zijn er twee grenswaarden, namelijk 1 en 99. In dit geval gaan we gebruik maken van `InlineData` (in combinatie met `Theory`) om onze tests uit te voeren.

```

[Theory]
[InlineData(1)]
[InlineData(99)]
❶ | 0 references | 0 changes | 0 authors, 0 changes
public void ZetRugnummer_valid(int rugnr)
{
    Speler s = new Speler(10, "Jos", 180, 80);
    s.ZetRugnummer(rugnr);
    Assert.Equal(rugnr, s.Rugnummer);
}
[Theory]
[InlineData(0)]
[InlineData(-1)]
[InlineData(100)]
❶ | 0 references | 0 changes | 0 authors, 0 changes
public void ZetRugnummer_invalid(int rugnr)
{
    Speler s = new Speler(10, "Jos", 180, 80);
    Assert.Throws<SpelerException>(() => s.ZetRugnummer(rugnr));
}

```

We tonen nog de test voor ZetNaam, ook deze zijn zeer eenvoudig. Belangrijk hierbij is om de verschillende mogelijkheden toe te voegen aan onze inlinedata.

```

[Theory]
[InlineData("Jeff", "Jeff")]
[InlineData("Eden Hazard", "Eden Hazard")]
[InlineData("Eden Hazard ", "Eden Hazard")]
[InlineData(" Eden Hazard ", "Eden Hazard")]
❶ | 0 references | 0 changes | 0 authors, 0 changes
public void ZetNaam_valid(string naamin, string naamuit)
{
    Speler s = new Speler(10, "Jos", 180, 80);
    s.ZetNaam(naamin);
    Assert.Equal(naamuit, s.Naam);
}
[Theory]
[InlineData(" ")]
[InlineData("")]
[InlineData(null)]
[InlineData("\n")]
[InlineData("\r")]
❶ | 0 references | 0 changes | 0 authors, 0 changes
public void ZetNaam_invalid(string naam)
{
    Speler s = new Speler(10, "Jos", 180, 80);
    Assert.Throws<SpelerException>(() => s.ZetNaam(naam));
}

```

Voor ZetGewicht, ZetLengte is de opzet volledig analoog.

Laten we nu eens kijken naar de constructor. Daarin worden allemaal methodes opgeroepen die we reeds hebben getest, is het dan nog nodig om hier nog een test voor te schrijven ? Toch wel, als tester moet je niet naar de code kijken van de methode dat je wenst te testen, maar wel naar de gevraagde functionaliteit van deze methode. En dus moeten we hier alle mogelijkheden opnieuw afchecken.

```
[Theory]
[InlineData(10, "Jos", 150, 80, 10, "Jos", 150, 80)]
[InlineData(1, " Jos ", 150, 50, 1, "Jos", 150, 50)]
[InlineData(1, "Jos", null, 80, 1, "Jos", null, 80)]
[InlineData(1, "Jos", 170, null, 1, "Jos", 170, null)]
❶ | 0 references | 0 changes | 0 authors, 0 changes
public void ctor_withID_valid(int id, string naam, int? lengte, int? gewicht, int vid,
    string vnaam, int? vlengthe, int? vgewicht)
{
    Speler s = new Speler(id, naam, lengte, gewicht);

    Assert.Equal(vid, s.Id);
    Assert.Equal(vnaam, s.Naam);
    Assert.Equal(vlengthe, s.Lengte);
    Assert.Equal(vgewicht, s.Gewicht);
}
```

```
[Theory]
[InlineData(0, "Jos", 150, 80)]
[InlineData(1, " ", 150, 50)]
[InlineData(1, "", 150, 50)]
[InlineData(1, null, 150, 50)]
[InlineData(1, "\n", 150, 50)]
[InlineData(1, "\r", 150, 50)]
[InlineData(1, "Jos", 149, 80)]
[InlineData(1, "Jos", 170, 49)]
❶ | 0 references | 0 changes | 0 authors, 0 changes
public void ctor_withID_invalid(int id, string naam, int? lengte, int? gewicht)
{
    Assert.Throws<SpelerException>(() => new Speler(id, naam, lengte, gewicht));
}
```

De testen voor de andere constructor kunnen jullie zelf wel aanvullen.

Rest ons nog de test van de relaties met de Team klasse. Als we de methode VerwijderTeam wensen te testen dan moeten we natuurlijk eerst de situatie opbouwen dat er een Team wordt toegekend aan een speler. We verwijderen nu het team en controleren of de property Team wel degelijk null is. Maar we moeten ook controleren of de speler nu ook niet meer in de lijst van spelers zit van het verwijderde team en dat doen we met de DoesNotContains methode.

```

[Fact]
● | 0 references | 0 changes | 0 authors, 0 changes
public void verwijderTeam_valid()
{
    Speler s = new Speler(10, "Jos", 180, 80);
    Team t = new Team(1, "Antwerpen");
    s.ZetTeam(t);
    s.VerwijderTeam();
    Assert.Null(s.Team);
    Assert.DoesNotContain(s, t.Spelers());
}

```

Ook voor het checken van de ZetTeam methode moeten we een dubbele controle doen, namelijk is de property team correct toegekend en is de speler ook toegevoegd aan de lijst van spelers in dat team. In deze test maken we een tweede object team aan met dezelfde waarden als het eerste team en gebruiken dit als verwachtte waarde. Op deze manier checken we ook of de Equals-methode wel is geïmplementeerd.

```

[Fact]
● | 0 references | 0 changes | 0 authors, 0 changes
public void ZetTeam_valid()
{
    Speler s = new Speler(10, "Jos", 180, 80);
    Team t = new Team(1, "Antwerpen");
    Team t2 = new Team(1, "Gent"); //checkt ook equals
    s.ZetTeam(t);
    Assert.Equal(t2, s.Team);
    Assert.Contains(s, t.Spelers());
}

```

De laatste test die we hier bespreken is het invalid path van de ZetTeam-methode. We voeren de test uit met een null-object en met een identiek object. In beide gevallen moet er een SpelerException worden gegooid. We controleren ook of er geen zij-effecten zijn, het falen van de methode mag ook niet tot gevolg hebben dat de property team werd aangepast. Dus we controleren of deze effectief nog steeds dezelfde is. Daarnaast gaan we ook na of de speler nog in de lijst van spelers van het team zit.

```

[Fact]
● | 0 references | 0 changes | 0 authors, 0 changes
public void ZetTeam_invalid()
{
    Speler s = new Speler(10, "Jos", 180, 80);
    Team t = new Team(1, "Antwerpen");
    s.ZetTeam(t);
    Assert.Throws<SpelerException>(()=>s.ZetTeam(null));
    Assert.Throws<SpelerException>(() => s.ZetTeam(t));
    Assert.Equal(t, s.Team);
    Assert.Contains(s, t.Spelers());
}

```

In de test explorer kunnen we dan het resultaat bekijken van de tests die we reeds hebben beschreven.

```
TestDomein (31)          8 ms
  TestProjectDomein (31)  8 ms
    UnitTestSpeler (31)   8 ms
      ctor_withID_invalid (8) < 1 ms
      ctor_withID_valid (4)  5 ms
      verwijderTeam_valid   < 1 ms
      ZetId_invalid         < 1 ms
      ZetId_valid           < 1 ms
      ZetNaam_invalid (5)   < 1 ms
      ZetNaam_valid (4)     < 1 ms
      ZetRugnummer_invalid (3) < 1 ms
      ZetRugnummer_valid (2) < 1 ms
      ZetTeam_invalid       3 ms
      ZetTeam_valid          < 1 ms
```

Het schrijven van de unit testen voor de andere klassen is analoog en laat ik dus aan jullie over.

Databank model

Alvorens we user stories gaan implementeren kiezen we er voor om eerst het databank model te maken. Aangezien het een vrij eenvoudig domein is, kunnen we gerust het volledige model apart behandelen en direct implementeren. We moeten de teams kunnen opslaan en een team heeft enkel een stamnummer, naam en bijnaam. Het stamnummer is een uniek nummer dat aan een ploeg wordt toegekend en kan dus dienst doen als onze primary key voor de tabel. De naam mag niet leeg zijn dus dwingen we dat ook af in de databank.

```
CREATE TABLE [dbo].[Team]
(
    [Stamnummer] INT NOT NULL PRIMARY KEY,
    [Naam] NVARCHAR(150) NOT NULL,
    [Bijnaam] NVARCHAR(150) NULL
)
```

Ook voor de spelers voorzien we een tabel met voor elke property uit de klasse een overeenkomstige kolom. Tussen team en speler is er een één-op-veel relatie en dat modelleren we door een verwijzing naar team op te nemen in de speler tabel. Om te verhinderen dat een speler zou verwijzen naar een niet bestaand team voegen we nog een foreign key toe. Spelers hebben ook een id en in dit geval laten we die genereren door de databank vandaar dat we de IDENTITY eigenschap toevoegen aan het id veld.

```

CREATE TABLE [dbo].[Speler]
(
    [Id] INT NOT NULL PRIMARY KEY IDENTITY,
    [Naam] VARCHAR(150) NOT NULL,
    [Rugnummer] INT NULL,
    [Lengte] INT NULL,
    [Gewicht] INT NULL,
    [TeamId] INT NULL,
    CONSTRAINT [FK_Speler_Team] FOREIGN KEY ([TeamId]) REFERENCES [Team]([Stamnummer])
)

```

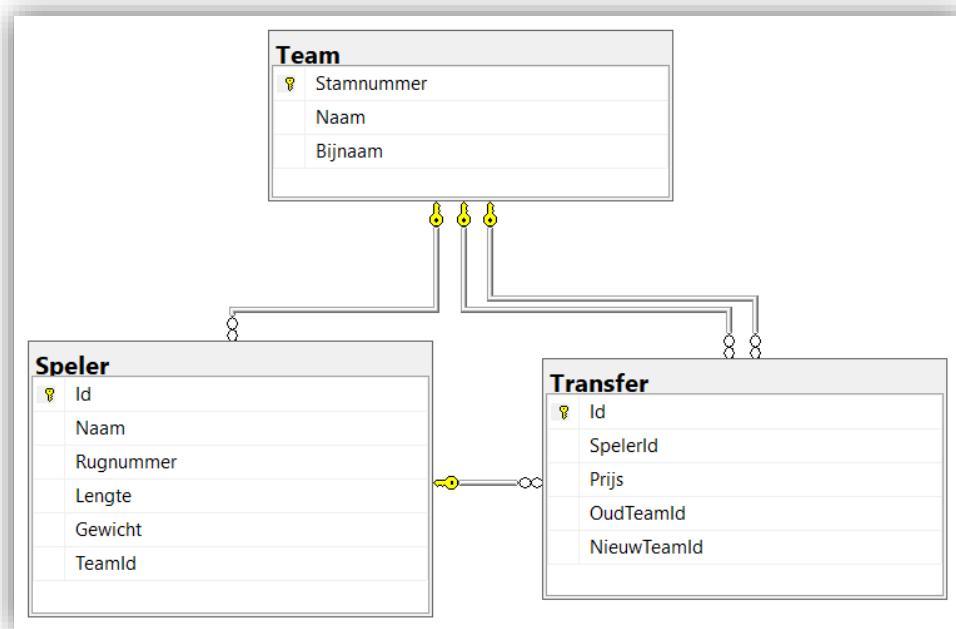
De laatste tabel dient om de transfers in op te slaan. Ook hier is er gekozen om de ids door de databank te laten aanmaken. In deze tabel zijn er een aantal relaties, zo is er een één-op-veel relatie tussen transfer en speler en voegen we dus een spelerid veld toe. Aangezien het verplicht is bij een transfer om te verwijzen naar een speler mag dit veld dus niet null zijn. Er zijn nog twee één-op-veel relaties te modelleren, namelijk tussen transfer en het oude tem en tussen transfer en het nieuwe team. Bij elk van de relaties voegen we nog een foreign key toe zodat er niet kan verwezen worden naar niet bestaande spelers of teams.

```

CREATE TABLE [dbo].[Transfer]
(
    [Id] INT NOT NULL PRIMARY KEY IDENTITY,
    [SpelerId] INT NOT NULL,
    [Prijs] INT NOT NULL,
    [OudTeamId] INT NULL,
    [NieuwTeamId] INT NULL,
    CONSTRAINT [FK_Transfer_Speler] FOREIGN KEY ([SpelerId]) REFERENCES [Speler]([Id]),
    CONSTRAINT [FK_Transfer_OudTeam] FOREIGN KEY ([OudTeamId]) REFERENCES [Team]([Stamnummer]),
    CONSTRAINT [FK_Transfer_NieuwTeam] FOREIGN KEY ([NieuwTeamId]) REFERENCES [Team]([Stamnummer])
)

```

Het complete databank model ziet er dan als volgt uit.



Anayse user stories

We hebben ondertussen het domein geïmplementeerd en het databankmodel aangemaakt. De volgende stap is nu het implementeren van de user stories. Maar vooraleer we hier aan beginnen moeten we ons eerst afvragen welke user stories er zijn en op welke manier we deze gaan uitvoeren. Over de user interface hebben we nog niet nagedacht, maar we weten wel reeds uit de beschrijving van ons systeem welke acties al zeker moeten kunnen worden uitgevoerd en dat zijn :

- Een speler toevoegen aan het systeem.
- De eigenschappen van een speler aanpassen.
- Een speler verwijderen uit het systeem
- Een team toevoegen aan het systeem.
- De eigenschappen van een team aanpassen.
- Een team verwijderen uit het systeem.

Daarnaast zijn er ook een aantal user stories die te maken hebben met het opvragen van gegevens, zo onderscheiden we :

- Een overzicht geven van alle teams.
- Een team opzoeken op basis van zijn stamnummer of naam.
- De spelers oplijsten die bij een team horen.
- De spelers oplijsten die bij geen enkel team horen.
- Transfers oplijsten.

Op het eerste zicht lijkt deze korte beschrijving voldoende om aan de implementatie te beginnen, maar toch zullen er afspraken moeten worden gemaakt wat we precies wel en niet zullen doen binnen een bepaalde user story. Laten we beginnen bij de eerste user story “voeg een speler toe aan het systeem”. Als we een speler toevoegen, kan deze speler dan al een team hebben ? Met andere woorden er zijn twee mogelijke scenario’s :

1. We voegen een speler toe zonder team en voegen in een aparte user story een transfer toe waarbij we de speler toekennen aan een team.
2. We voegen de speler direct toe aan een team, samen met de transfer die daarbij hoort.

We kiezen ervoor om steeds het volgende scenario (scenario 1) uit te voeren :

- We registreren een speler (toevoegen aan het systeem)
- We maken een transfer aan om de speler toe te kennen aan een team.

Voor het aanpassen van de eigenschappen van een speler zullen we ons beperken tot de rugnummer, lengte en gewicht. Het veranderen van team beschouwen we niet als een aanpassing van de speler en plaatsen we in de user story om een transfer uit te voeren.

Ook voor het toevoegen van een team moeten we keuzes maken en we kiezen om een team toe te voegen zonder spelers. Het aanpassen van de relaties tussen speler en team zullen we altijd regelen met een transfer.

Het verwijderen van spelers en teams uit het systeem stellen we voorlopig even uit, daar komen we later uitgebreid op terug.

De user stories die te maken hebben met bevragingen zullen we eveneens later bespreken. Dus wat de analyse betreft zijn er nog twee belangrijke taken die we moeten uitzoeken :

1. Verwijderen van spelers en teams.
2. Bevragingen van het systeem.

User story : voeg een speler toe aan het systeem

Het domein is geïmplementeerd en de databank aangemaakt, we vervolgen de ontwikkeling van onze applicatie nu aan de hand van user stories. De eerste user story die we wensen te implementeren is het toevoegen van een speler aan het systeem. Hoe gaan we nu te werk ? Het beheer van spelers is niet de verantwoordelijkheid van de reeds bestaande domeinklassen, dus moeten we een nieuwe klasse aanmaken die deze functie zal opnemen. We kiezen voor een aparte klasse die zal instaan voor het volledige beheer van de spelers. Opmerking : als we vooruit blikken dan merken we dat voor het beheer van teams een analoge klasse zal moeten worden aangemaakt vandaar dat we een extra folder 'managers' opnemen in ons project.

Het beheren van de spelers en de logica die hierbij moet worden gehanteerd is 'business' logica dus plaatsen we deze klasse dan ook in de business-laag. Om onze user story uit te voeren voegen we een methode `RegistreerSpeler` toe aan deze klasse. Deze methode heeft als parameters de noodzakelijk properties voor een speler en geeft als resultaat een speler-object terug. Als er eventueel iets zou mis gaan voorzien we ook een custom exception die bij deze klasse hoort, namelijk `SpelerManagerException` en plaatsen deze in de folder exceptions.

De methode `RegistreerSpeler` moet de volgende taken uitvoeren :

- Een speler aanmaken.
- Een speler opslaan in de databank.

Voor het aanmaken van een speler kunnen we eenvoudigweg de constructor van speler oproepen. Het effectief wegschrijven van de speler naar de databank daarentegen is feitelijk de verantwoordelijkheid van de datalaag. In deze klasse moeten we dan ook de link maken naar de datalaag en dat doen we door het definiëren van een interface die beschrijft welke acties de klassen uit de datalaag moeten uitvoeren. Voorlopig ziet onze interface er als volgt uit :

```
public interface ISpelerRepository
{
    0 references | 0 changes | 0 authors, 0 changes
    Speler SchrijfSpelerInDB(Speler s);
    0 references | 0 changes | 0 authors, 0 changes
    bool BestaatSpeler(Speler s);
}
```

We wensen dus een object dat ons kan vertellen of een bepaalde speler reeds bestaat (`BestaatSpeler`) en dat in staat is om een speler op te slaan in de databank (`SchrijfSpelerInDB`). Ook voor de interfaces voorzien we een aparte folder.

In de klasse `SpelerManager` voegen we een variabele toe van het type `ISpelerManager` die we via de constructor initialiseren.

```

public class SpelerManager
{
    private ISpelerRepository repo;

    1 reference | Tom Vande Wiele, 1 day ago | 1 author, 1 change
    public SpelerManager(ISpelerRepository repo)
    {
        this.repo = repo;
    }
}

```

In de RegistreerSpeler methode maken we een speler-object aan, daarna vragen we aan onze repository of deze speler reeds bestaat en indien dat niet het geval is dan schrijven we deze speler weg in de databank.

```

public Speler RegistreerSpeler(string naam, int? lengte, int? gewicht)
{
    try
    {
        Speler s = new Speler(naam, lengte, gewicht);
        if (!repo.BestaatSpeler(s))
        {
            s = repo.SchrijfSpelerInDB(s);
            return s;
        }
        else
        {
            throw new SpelerManagerException("RegistreerSpeler - speler bestaat al");
        }
    }
    catch (SpelerManagerException)
    {
        throw;
    }
    catch (Exception ex)
    {
        throw new SpelerManagerException("RegistreerSpeler", ex);
    }
}

```

De volgende stap is nu de implementatie van de interface ISpelerRepository. De klasse die we deze interface implementeert, SpelerRepositoryADO, behoort tot de datalaag. In het project League.DL maken we een folder repositories aan waarin we onze klasse onderbrengen. Opmerking : vergeet niet om bij de dependencies een project link te leggen naar de business-laag.

Om de connectie te kunnen maken met de databank hebben we nog een connectiestring nodig en die geven we mee met de constructor van onze klasse. De klasse ziet er dan als volgt uit.

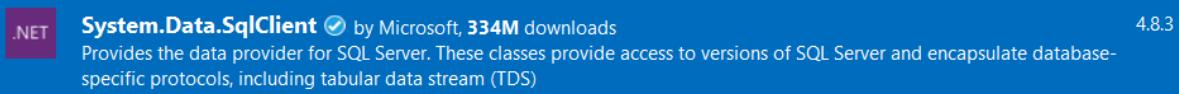
```
public class SpelerRepositoryADO : ISpelerRepository
{
    private string connectieString;

    0 references | 0 changes | 0 authors, 0 changes
    public SpelerRepositoryADO(string connectieString)
    {
        this.connectieString = connectieString;
    }

    2 references | 0 changes | 0 authors, 0 changes
    public bool BestaatSpeler(Speler s) ...

    2 references | 0 changes | 0 authors, 0 changes
    public Speler SchrijfSpelerInDB(Speler s) ...
}
```

Aangezien we gebruik gaan maken van ADO.NET moeten we nog het package System.Data.SqlClient installeren.



We kiezen er ook voor om een methode te schrijven die ons telkens van een connectieobject voorziet.

```
private SqlConnection getConnection()
{
    return new SqlConnection(connectieString);
}
```

De eerste methode die we nu implementeren is BestaatSpeler. Om te controleren of de speler reeds is aangemaakt gaan we gebruik maken van een query die telt hoeveel spelers er zijn met deze naam. Is er nog geen dan gaan we er van uit dat de speler nog niet bestaat, in het andere geval bestaat de speler wel al. Opmerking : in realiteit zal je meer dan de naam moeten checken.

```
public bool BestaatSpeler(Speler s)
{
    SqlConnection connection = getConnection();
    string query = "SELECT count(*) FROM dbo.Speler WHERE naam=@naam";

    using (SqlCommand command = connection.CreateCommand())
    {
        connection.Open();
        try
        {
            command.Parameters.Add(new SqlParameter("@naam", SqlDbType.NVarChar));
            command.CommandText = query;
            command.Parameters["@naam"].Value = s.Naam;
            int n = (int)command.ExecuteScalar();
            if (n > 0) return true;
            else return false;
        }
        catch (Exception ex)
        {
            throw new SpelerRepositoryException("BestaatSpeler", ex);
        }
        finally
        {
            connection.Close();
        }
    }
}
```

In onze methode maken we een connectieobject aan en een command object. We voegen de parameter toe en stellen deze in samen met de commandtext en daarna volgt de ExecuteScalar methode die ons het resultaat van de query geeft. We plaatsen alles in een try catch finally statement omdat we telkens zeker willen zijn dat de connectie wordt afgesloten.

De tweede methode ziet er als volgt uit :

```

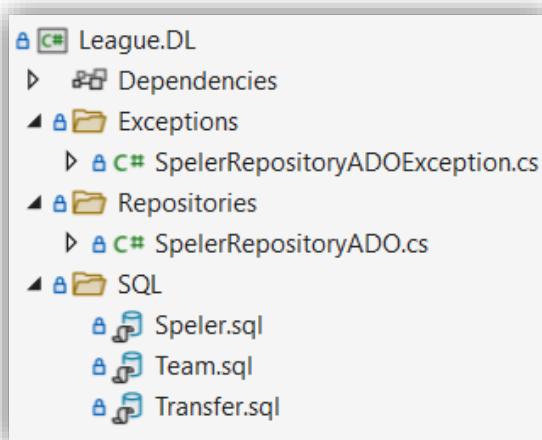
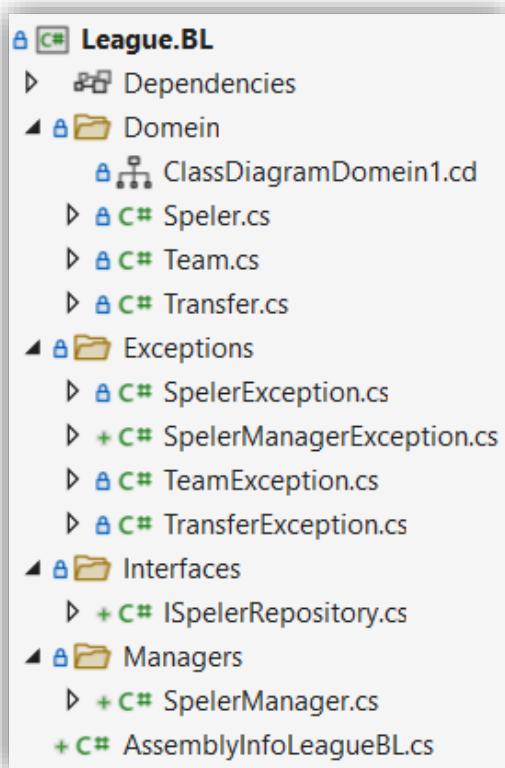
public Speler SchrijfSpelerInDB(Speler s)
{
    SqlConnection conn = getConnection();
    string query = "INSERT INTO dbo.Speler(naam,lengte,gewicht) "
        + "output INSERTED.ID VALUES(@naam,@lengte,@gewicht)";
    try
    {
        using (SqlCommand cmd = conn.CreateCommand())
        {
            conn.Open();
            cmd.Parameters.Add(new SqlParameter("@naam", System.Data.SqlDbType.NVarChar));
            cmd.Parameters.Add(new SqlParameter("@lengte", System.Data.SqlDbType.Int));
            cmd.Parameters.Add(new SqlParameter("@gewicht", System.Data.SqlDbType.Int));
            cmd.Parameters["@naam"].Value = s.Naam;
            if (s.Lengte == null) cmd.Parameters["@lengte"].Value = DBNull.Value;
            else cmd.Parameters["@lengte"].Value = s.Lengte;
            if (s.Gewicht == null) cmd.Parameters["@gewicht"].Value = DBNull.Value;
            else cmd.Parameters["@gewicht"].Value = s.Gewicht;
            cmd.CommandText = query;
            int newID = (int)cmd.ExecuteScalar();
            s.ZetId(newID);
            return s;
        }
    }
    catch (Exception ex)
    {
        throw new SpelerRepositoryException("SchrijfSpelerInDB", ex);
    }
    finally
    {
        conn.Close();
    }
}

```

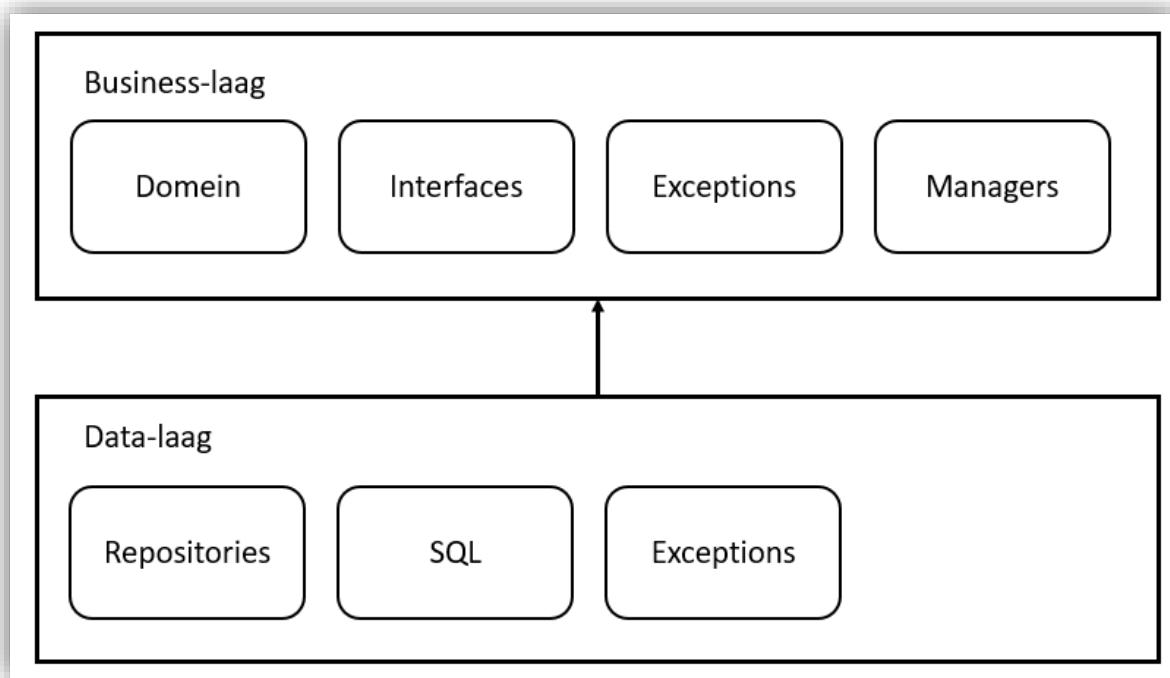
Enkele belangrijke punten in deze methode zijn de volgende :

- De methode geeft een speler-object terug en dat doen we omdat het speler-object waar we mee starten wordt aangepast. We hebben er namelijk voor gekozen om de spelerid door de databank te laten aanmaken. Deze id is pas beschikbaar nadat het insert-statement is uitgevoerd. We passen het speler-object aan met de methode ZetId en returnen het aangepaste object. Opmerking : het is niet strikt noodzakelijk om het op deze manier te doen, maar door een speler-object terug te geven maken we onze gebruiker er wel attent op dat er iets veranderd is.
- Om het insert-statement de nieuw aangemaakt id te laten teruggeven moeten we in het SQL-statement “output INSERTED.ID” opnemen en de query uitvoeren met ExecuteScalar.
- Wanneer we een null-waarde wensen weg te schrijven in de databank dan moeten we dat expliciet aangeven en dat kan door aan de parameter de waarde DBNull.Value toe te kennen.

Onze eerste user story is nu afgerond en in onze solution bevatten de business-laag en de data-laag nu de volgende elementen.



En dat kunnen we schematisch weergeven in onze lagen-architectuur als volgt:



User story : voeg een team toe aan het systeem

Het toevoegen van een team aan het systeem verloopt vrij analoog met het toevoegen van een speler. Wanneer we een nieuw team registreren dan is dat wel steeds een team zonder spelers, die moeten via transfers worden toegekend aan het team.

We starten met het aanmaken van een TeamManager klasse met daarin een methode registreer team. Aangezien we in de teammanager klasse instructies zullen geven aan de data laag voor het effectief opslaan van de gegevens zullen we ook hier een interface definiëren met daarin de methodes die in de data laag zullen moeten worden geïmplementeerd. De klasse die de interface implementeert geven we mee met de constructor.

```

public class TeamManager
{
    private ITeamRepository repo;

    0 references | 0 changes | 0 authors, 0 changes
    public TeamManager(ITeamRepository repo)
    {
        this.repo = repo;
    }
}

```

Voor de interface voorzien we voorlopig slechts twee methodes, één voor te zien of een team reeds bestaat en één voor het wegschrijven van de gegevens.

```
public interface ITeamRepository
{
    void SchrijfTeamInDB(Team t);
    bool BestaatTeam(Team t);
}
```

Voor het registreren van een team roepen we in eerste instantie de constructor van team op, daarna stellen we de bijnaam in als die ten minste is ingevuld. Als dat is gelukt vragen we aan de repository of dit team reeds bestaat en als dat niet het geval is dan voegen we het team toe in de databank. Ook voor de klasse TeamManager voorzien we een custom exception (net zoals bij de andre klassen).

```
public void RegistreerTeam(int stamnummer, string naam, string bijnaam)
{
    try
    {
        Team t = new Team(stamnummer, naam);
        if (!string.IsNullOrWhiteSpace(bijnaam)) t.ZetBijnaam(bijnaam);
        if (!repo.BestaatTeam(t))
        {
            repo.SchrijfTeamInDB(t);
        }
        else
        {
            throw new TeamManagerException("RegistreerTeam - team bestaat al");
        }
    }
    catch (TeamManagerException)
    {
        throw;
    }
    catch (Exception ex)
    {
        throw new TeamManagerException("RegistreerTeam", ex);
    }
}
```

Rest ons enkel nog de implementatie in de datalaag. De klasse die we daarvoor aanmaken is TeamRepositoryADO waarin we een connectiestring opnemen die we via de constructor mee krijgen.

```

public class TeamRepositoryADO : ITeamRepository
{
    private string connectieString;

    0 references | 0 changes | 0 authors, 0 changes
    public TeamRepositoryADO(string connectieString)
    {
        this.connectieString = connectieString;
    }

    2 references | 0 changes | 0 authors, 0 changes
    private SqlConnection getConnection()
    {
        return new SqlConnection(connectieString);
    }

    2 references | 0 changes | 0 authors, 0 changes
    public bool BestaatTeam(Team team) ...

    2 references | 0 changes | 0 authors, 0 changes
    public void SchrijfTeamInDB(Team team) ...
}

```

De implementatie van de beide methodes ziet er als volgt uit :

```

public void SchrijfTeamInDB(Team team)
{
    SqlConnection connection = getConnection();
    string query = "INSERT INTO dbo.Team(stamnummer,naam,bijnaam) VALUES(@stamnummer,@naam,@bijnaam)";

    using (SqlCommand command = connection.CreateCommand())
    {
        connection.Open();
        try
        {
            command.Parameters.Add(new SqlParameter("@naam", SqlDbType.NVarChar));
            command.Parameters.Add(new SqlParameter("@stamnummer", SqlDbType.Int));
            command.Parameters.Add(new SqlParameter("@bijnaam", SqlDbType.NVarChar));
            command.CommandText = query;
            command.Parameters["@naam"].Value = team.Naam;
            command.Parameters["@stamnummer"].Value = team.Stamnummer;
            if (team.Bijnaam == null)
                command.Parameters["@bijnaam"].Value = DBNull.Value;
            else
                command.Parameters["@bijnaam"].Value = team.Bijnaam;
            command.ExecuteNonQuery();
        }
        catch (Exception ex)
        {
            throw new TeamRepositoryADOException("VoegTeamToe", ex);
        }
        finally
        {
            connection.Close();
        }
    }
}

```

Aangezien voor de klasse team de unieke identificator (stamnummer) steeds een opgegeven waarde is en niet door de databank moet worden gegenereert is de methode iets eenvoudiger vergeleken met de methode SchrijfSpelerInDB. Ook hier moeten we rekening houden met het feit dat de bijnaam leeg kan zijn en dat we dan de DBNull.Value waarde moeten toekennen aan de parameter.

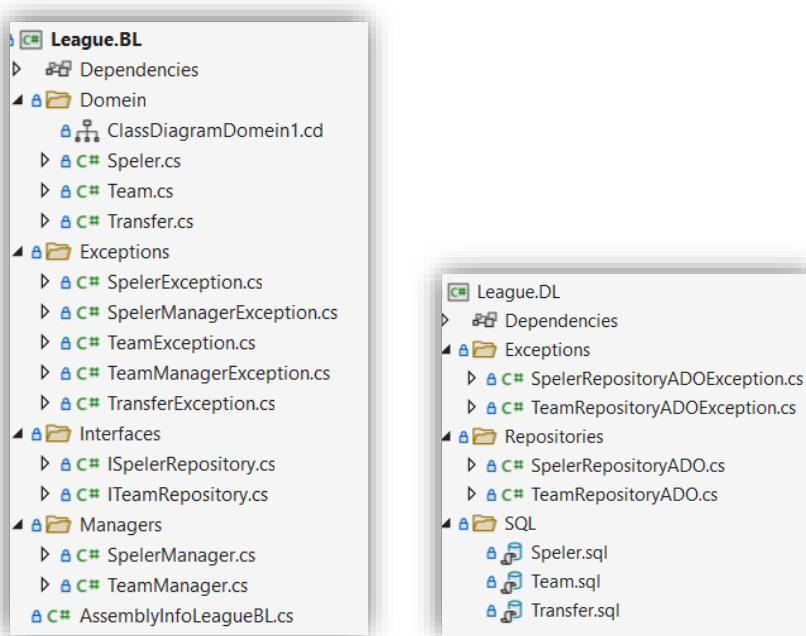
De BestaatTeam-methode daarentegen is volledig analoog aan de BestaatSpeler-methode, al kunnen we hier het stamnummer gebruiken om te zien of een team reeds bestaat.

```
public bool BestaatTeam(Team team)
{
    SqlConnection connection = getConnection();
    string query = "SELECT count(*) FROM dbo.Team WHERE stamnummer=@stamnummer";

    using (SqlCommand command = connection.CreateCommand())
    {
        connection.Open();
        try
        {
            command.Parameters.Add(new SqlParameter("@stamnummer", SqlDbType.Int));
            command.CommandText = query;
            command.Parameters["@stamnummer"].Value = team.Stamnummer;

            int n = (int)command.ExecuteScalar();
            if (n > 0) return true;
            else return false;
        }
        catch (Exception ex)
        {
            throw new TeamRepositoryADOException("BestaatTeam", ex);
        }
        finally
        {
            connection.Close();
        }
    }
}
```

Ons project ziet er ondertussen als volgt uit :



User story : selecteer een team op basis van zijn stamnummer.

We kiezen ervoor om deze user story nu te implementeren omdat we het resultaat nodig hebben om de andere user stories te kunnen uitvoeren. Zo kunnen we geen transfer aanmaken zonder eerst een team te selecteren of kunnen we ook geen team aanpassen zonder het eerst op te vragen. Om een team te kunnen opvragen voegen we eerst aan de IteamRepository een methode toe (SelecteerTeam) die aan de klasse TeamRepositoryADO zal opleggen om een team uit de databank op te halen.

```
public interface ITeamRepository
{
    void SchrijfTeamInDB(Team t);
    bool BestaatTeam(Team t);
    Team SelecteerTeam(int stamnummer);
}
```

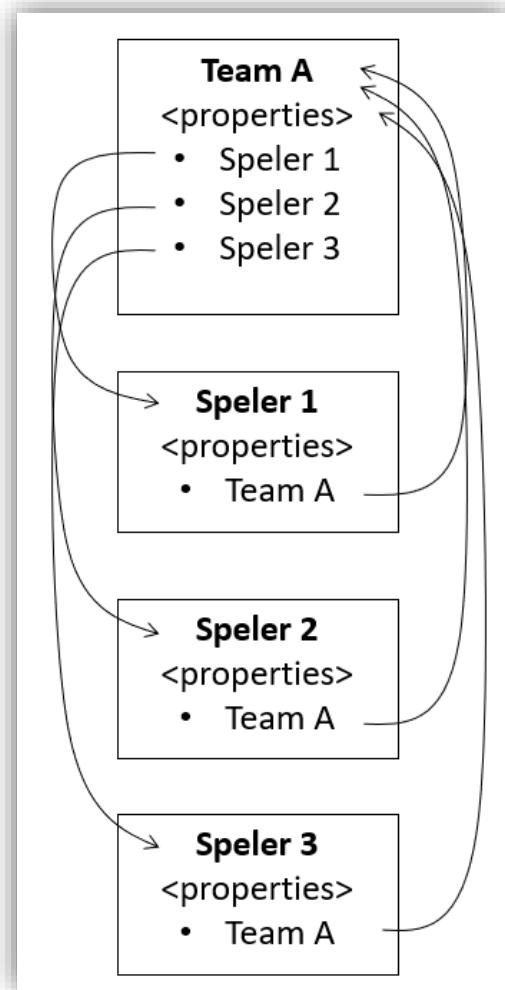
In de businesslaag voegen we dan in de teammanager klasse een methode toe om het team te selecteren. Deze methode zal gebruik maken van de nieuwe methode die we aan de ITeamRepository hebben toegevoegd. Is er geen team gevonden (en er is ook geen exception gegooid) dan gaan we er van uit dat het team niet bestaat en beschouwen we dat dus als een exception (er is een foute vraag gesteld).

```
public Team SelecteerTeam(int stamnummer)
{
    try
    {
        Team team= repo.SelecteerTeam(stamnummer);
        if (team == null) throw new TeamManagerException("SelecteerTeam - team bestaat niet");
        return team;
    }
    catch (TeamManagerException)
    {
        throw;
    }
    catch (Exception ex)
    {
        throw new TeamManagerException("SelecteerTeam", ex);
    }
}
```

De implementatie van de methode in de datalaag is minder evident dan we op het eerste zicht zouden denken. Als we een team opvragen dan moet dat een team zijn volgens de domeindefinities, met andere woorden een team MET spelers en waarbij de relaties tussen de verschillende objecten correct zijn ingevuld.

Aten we dat eens in detail bekijken. We hebben een object team (vb Team A) met enkele properties (naam, bijnaam) en een lijst van spelers. Het opvragen van een Team impliceert dus dat we ook de

verschillende spelers opvragen die tot dat team behoren. Elk speler object verwijst ook naar een team object dus die relaties zullen we ook moeten invullen als we het object samenstellen.



Het is ook belangrijk om op een efficiënte manier de gegevens op te vragen, dus gaan we het opvragen van een team met zijn spelers in één query uitvoeren. Die query ziet er dan als volgt uit :

```
SELECT t1.stamnummer, t1.naam as ploegnaam, t1.bijnaam, t2.*  
FROM [dbo].[Team] t1  
left join [dbo].[speler] t2 on t1.Stamnummer = t2.teamid  
WHERE stamnummer=1
```

In de tabel Team zoeken we de record die overeenstemt met het opgegeven stamnummer en om de spelers direct mee te krijgen voegen we een left join toe met de speler tabel (left join want het zou ook kunnen dat er teams zijn zonder spelers en dan willen we ook een resultaat hebben). Belangrijk is nog om een alias te gebruiken voor de naam kolom in de tabel ploeg, aangezien we twee kolommen hebben met dezelfde benaming ('naam' in team en 'naam' in speler). Het resultaat van deze query kan er dan als volgt uitzien :

stamnummer	ploegnaam	bijnaam	Id	Naam	Rugnummer	Lengte	Gewicht	TeamId
1	Antwerp	Great Old	2	jos	8	179	69	1
1	Antwerp	Great Old	4	Jordy	NULL	189	94	1
2	Westerlo	NULL	3	Mark	NULL	NULL	NULL	2
3	Deinze	NULL	NULL	NULL	NULL	NULL	NULL	NULL
4	Beveren	NULL	NULL	NULL	NULL	NULL	NULL	NULL

De implementatie in de SelecteerTeam methode is als volgt, we definiëren de query met het stamnummer als parameter.

```
public Team SelecteerTeam(int stamnummer)
{
    SqlConnection connection = getConnection();
    string query = "SELECT t1.stamnummer,t1.naam as ploegnaam,t1.bijnaam,t2.* "
        + "FROM [dbo].[Team] t1 left join [dbo].[speler] t2 on t1.Stamnummer = t2.teamid "
        + "WHERE stamnummer=@stamnummer";
    using (SqlCommand command = connection.CreateCommand())
    {
        command.Parameters.Add(new SqlParameter("@stamnummer", SqlDbType.Int));
        command.CommandText = query;
        command.Parameters["@stamnummer"].Value = stamnummer;
        connection.Open();
        try{...}
    }
}
```

De try catch bevat de uitvoering van de query en het toekennen van het resultaat aan een DataReader.

```
try
{
    Team team = null;
    IDataReader reader = command.ExecuteReader(); //of SqlDataReader
    while (reader.Read())...
    reader.Close();
    return team;
}
catch (Exception ex)
{
    throw new TeamRepositoryADOException("selecteerTeam", ex);
}
finally
{
    connection.Close();
}
```

In de while-lus doorlopen we alle records die de query teruggeeft. Bekijken we het resultaat van onze query dan zien we dat de info over het team voor elke record wordt herhaald. Het is zeker niet onze bedoeling om deze info telkens opnieuw te gaan lezen en gebruiken, één keer volstaat wel. We hebben een variabele team voorzien die we initieel op null instellen en bij de eerste record die we lezen gaan we de teaminfo ophalen en het team object initialiseren. Voor al de volgende records gaan we dat niet meer doen, vandaar dat we een if statement toevoegen die controleert of het team-object reeds is aangemaakt. De bijnaam van een team kan null (DBNull) zijn en dat vraagt een aparte behandeling. Om te vermijden dat onze code een exception gooit bij null-waarden, vragen we aan de datareader of de waarde DBNull is. Bij de IsDBNull-methode moet er een kolom-indexnummer worden meegegeven en het opvragen van de kolomindex kan door middel van de methode GetOrdinal die dan weer de naam van de kolom nodig heeft.

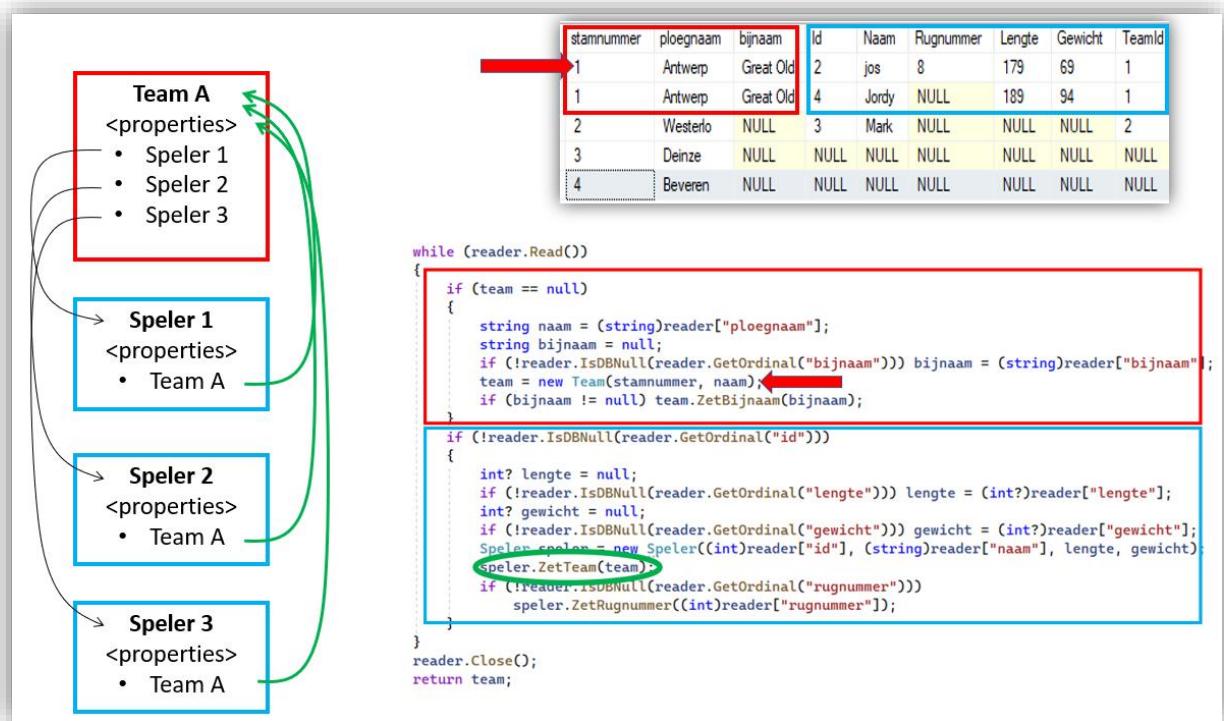
```

while (reader.Read())
{
    if (team == null)
    {
        string naam = (string)reader["ploegnaam"];
        string bijnaam = null;
        if (!reader.IsDBNull(reader.GetOrdinal("bijnaam")))
            bijnaam = (string)reader["bijnaam"];
        team = new Team(stamnummer, naam);
        if (bijnaam != null) team.ZetBijnaam(bijnaam);
    }
    if (!reader.IsDBNull(reader.GetOrdinal("id")))
    {
        int? lengte = null;
        if (!reader.IsDBNull(reader.GetOrdinal("lengte")))
            lengte = (int?)reader["lengte"];
        int? gewicht = null;
        if (!reader.IsDBNull(reader.GetOrdinal("gewicht")))
            gewicht = (int?)reader["gewicht"];
        Speler speler = new Speler((int)reader["id"], (string)reader["naam"], lengte, gewicht);
        speler.ZetTeam(team);
        if (!reader.IsDBNull(reader.GetOrdinal("rugnummer")))
            speler.ZetRugnummer((int)reader["rugnummer"]);
    }
}
reader.Close();
return team;

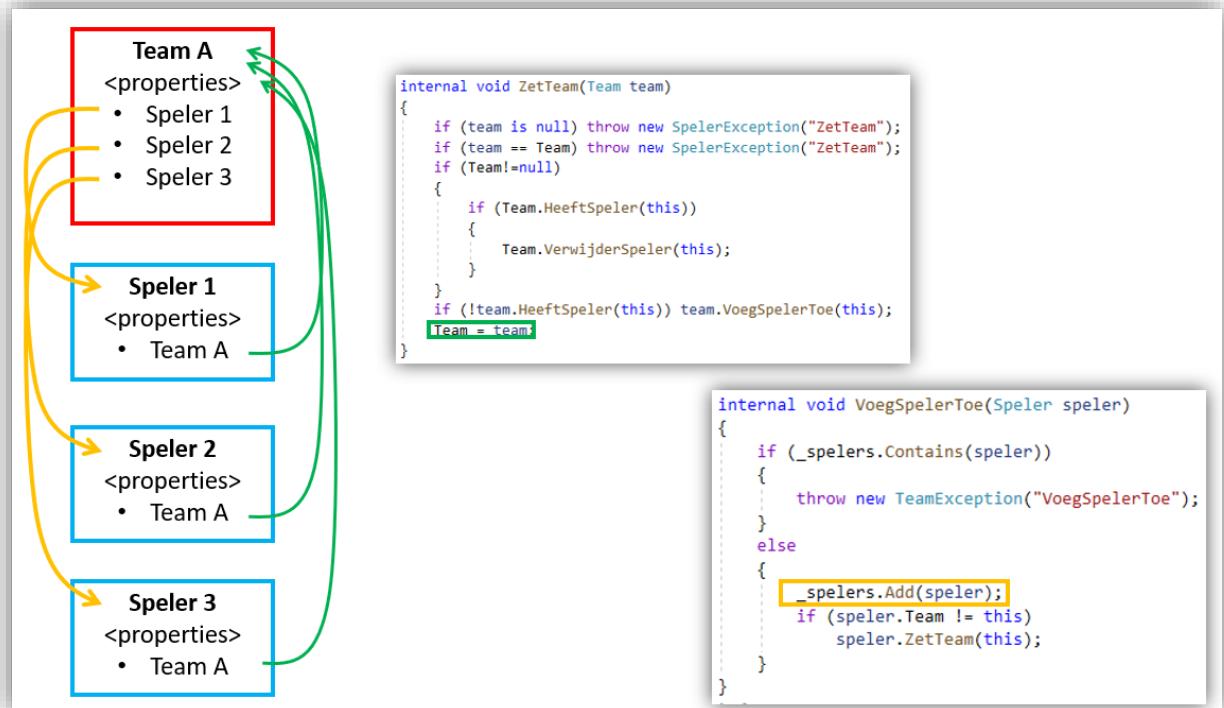
```

Na het ophalen van de teaminfo verwerken we de spelersinfo. Indien er geen spelers zijn dan zal de waarde van de kolom 'id' null zijn en dat moeten we eerst nagaan. Zijn er wel spelers dan lezen we de waarden voor lengte en gewicht. Ook hier moeten we nagaan of er geen null-waarden zijn ingevuld vandaar dat we deze apart behandelen. Daarna kunnen we een speler object aanmaken door de constructor van speler op te roepen (opmerking : ook hier moeten we in de assemblyinfo aangeven dat het datalaag-project de internals mag gebruiken). Eénmaal de speler is aangemaakt stellen we ook het team in en zo leggen we de relatie tussen team en speler. Tenslotte halen we nog het rugnummer op en als dat niet null is stellen we dat ook in.

In de volgende figuren geven we aan op welke manier we het team-object opbouwen. In het rood vinden we de info over het team, blauw bevat de info over de spelers, in het groen is aangeduid hoe de relatie tussen speler en team wordt ingesteld en in het geel tenslotte leggen we de relatie tussen team en speler. Op deze manier wordt het volledige team-object opgebouwd.



Het oproepen van de ZetTeam-methode triggert de VoegSpelerToe-methode waardoor de relatie tussen Team en Spelers wordt ingesteld.



User story : voeg een transfer toe aan het systeem.

We hebben reeds spelers en teams toegevoegd aan het systeem, nu gaan we via een transfer een speler toekennen aan een team. We onderscheiden daarbij drie verschillende scenario's :

1. Een speler heeft nog geen team en wordt nu toegekend aan een team.
2. Een speler heeft een team en verandert van team.
3. Een speler heeft een team en stopt (dus geen nieuw team).

We maken een klasse transfermanager aan in de businesslaag met daarin een methode RegistreerTransfer en een variabele repo waarvoor we een interface definiëren.

```
public class TransferManager
{
    private ITransferRepository repo;

    public TransferManager(ITransferRepository repo)
    {
        this.repo = repo;
    }

    public Transfer RegistreerTransfer(Speler speler, Team nieuwTeam, int prijs) ...
}
```

De interface bevat voorlopig slechts één methode namelijk SchrijfTransferInDB die we in de datalaag zullen implementeren.

```
public interface ITransferRepository
{
    Transfer SchrijfTransferInDB(Transfer transfer);
}
```

De methode RegistreerTransfer krijgt als parameters mee de speler die de transfer maakt, het nieuwe team en de prijs. Zoals reeds werd aangehaald zijn er drie mogelijke scenario's. Als de parameter nieuwTeam een null-waarde bevat dan wil dat zeggen dat de speler stopt we maken dan een Transfer-object aan met het speler-object en het oude team, verder verwijderen we het huidige team van de speler. Hebben we te maken met een nieuwe speler (een speler waar de property team een null-waarde bevat) dan roepen we de constructor van transfer op met de speler, het nieuwe team en de prijs. De property team van speler wordt ook hier aangepast. In het derde geval – de klassieke transfer tussen twee teams – maken we een transfer-object aan met daarin de speler, het nieuwe team en het oude team en de prijs. Het nieuwe team van de speler passen we eveneens aan. We eindigen met het wegschrijven van de transfer in de databank.

```

public Transfer RegistreerTransfer(Speler speler, Team nieuwTeam, int prijs)
{
    if (speler==null) throw new TransferManagerException("RegistreerTransfer - speler is null");
    Transfer ...transfer=null;
    try
    {
        //speler stopt
        if (nieuwTeam == null)
        {
            if (speler.Team == null) throw new TransferManagerException("RegistreerTransfer - team is null");
            transfer = new Transfer(speler, speler.Team);
            speler.VerwijderTeam();
        }
        //nieuwe speler
        else if (speler.Team == null)
        {
            speler.ZetTeam(nieuwTeam);
            transfer = new Transfer(speler, nieuwTeam, prijs);
        }
        //klassieke transfer
        else
        {
            transfer = new Transfer(speler, nieuwTeam, speler.Team, prijs);
            speler.ZetTeam(nieuwTeam);
        }
        return repo.SchrijfTransferInDB(transfer);
    }
    catch (TransferManagerException)
    {
        throw;
    }
    catch (Exception ex)
    {
        throw new TransferManagerException("RegistreerTransfer",ex);
    }
}

```

In de datalaag maken we de klasse TransferRepositoryADO die de interface ITransferRepository implementeert. Net zoals in onze andere repository klassen voegen we een connectiestring toe en een getConnection-methode.

```

public class TransferRepositoryADO : ITransferRepository
{
    private string ...connectieString;

    1 reference | 0 changes | 0 authors, 0 changes
    public TransferRepositoryADO(string connectieString)
    {
        this.connectieString = connectieString;
    }

    1 reference | 0 changes | 0 authors, 0 changes
    private SqlConnection getConnection()
    {
        return new SqlConnection(connectieString);
    }

    2 references | Tom Vande Wiele, 19 hours ago | 1 author, 1 change
    public Transfer SchrijfTransferInDB(Transfer transfer)...
}

```

De SchrijfTransferInDB-methode ziet er enigsinds anders uit dan de voorgaande methodes waarbij we gegevens wegschrijven naar de databank. Bij het wegschrijven van een transfer zijn er namelijk

twee acties nodig. Enerzijds moeten we een record aanmaken in de transfer-tabel, maar we moeten ook een update doen in de speler-tabel aangezien we de teamid-kolom moeten aanpassen. Beide acties moeten steeds samen gebeuren, het kan niet zijn dat slechts één van beide lukt want dan bevinden we ons in een inconsistente toestand. Om dat te vermijden maken we gebruik van een transactie. Nadat we de connectie openen, kunnen we een SqlTransaction-object aanmaken en dat toekennen aan de twee SqlCommands die we gaan uitvoeren. We maken een SqlCommand voor het wegschrijven van de transfer en één voor het updaten van de speler.

```
public Transfer SchrijfTransferInDB(Transfer transfer)
{
    SqlConnection connection = getConnection();

    string queryTransfer = "INSERT INTO dbo.Transfer(spelerid, prijs, oudteamid, nieuwteamid) "
        + "output INSERTED.ID VALUES(@spelerid, @prijs, @oudteamid, @nieuwteamid)";
    string querySpeler = "UPDATE speler SET teamid=@teamid WHERE id=@id";

    connection.Open();
    SqlTransaction transaction = connection.BeginTransaction();
    using (SqlCommand commandSpeler = connection.CreateCommand())
    using (SqlCommand commandTransfer = connection.CreateCommand())
    {
        commandTransfer.Transaction = transaction;
        commandSpeler.Transaction = transaction;
        try{...}
    }
}
```

In het try-gedeelte behandelen we eerst het transfer-command, waarbij we de verschillende parameters instellen. Ook hier moeten we rekening houden met eventuele null-waarden. Het uitvoeren van het insert-statement doen we via de ExecuteScalar-methode omdat we de aangemaakte transfer-id wensen terug te krijgen (vandaar ook het ‘output INSERTED.ID’ in het SQL-statement). De ontvangen id wordt dan ook in het transfer-object ingevuld.

Daarna stellen we het SpelerCommand-object in, we voegen de parameters toe, kennen het SQL-statement toe en voeren de update uit met ExecuteNonQuery. Om het geheel af te sluiten en de transactie uit te voeren roepen we de methode transaction.Commit op.

In het catch-gedeelte hebben we nu niet enkel het gooien van de custom-exception, maar dan we ook een RollBack. Dat wil zeggen dat als de transactie niet volledig is gelukt dat we terugkeren naar de begintoestand. Op die manier zorgen we ervoor dat de databank niet in een inconsistente toestand terecht kan komen.

```

try
{
    //transfer
    commandTransfer.Parameters.Add(new SqlParameter("@spelerid", SqlDbType.Int));
    commandTransfer.Parameters.Add(new SqlParameter("@prijs", SqlDbType.Int));
    commandTransfer.Parameters.Add(new SqlParameter("@oudteamid", SqlDbType.Int));
    commandTransfer.Parameters.Add(new SqlParameter("@nieuwteamid", SqlDbType.Int));
    commandTransfer.CommandText = queryTransfer;
    commandTransfer.Parameters["@spelerid"].Value = transfer.Speler.Id;
    commandTransfer.Parameters["@prijs"].Value = transfer.Prijs;
    if (transfer.OudTeam == null)
        commandTransfer.Parameters["@oudteamid"].Value = DBNull.Value;
    else
        commandTransfer.Parameters["@oudteamid"].Value = transfer.OudTeam.Stamnummer;
    if (transfer.NieuwTeam == null)
        commandTransfer.Parameters["@nieuwteamid"].Value = DBNull.Value;
    else
        commandTransfer.Parameters["@nieuwteamid"].Value = transfer.NieuwTeam.Stamnummer;
    int newID = (int)commandTransfer.ExecuteScalar();
    transfer.ZetId(newID);
    //speler
    commandSpeler.Parameters.Add(new SqlParameter("@id", SqlDbType.Int));
    commandSpeler.Parameters.Add(new SqlParameter("@teamid", SqlDbType.Int));
    commandSpeler.CommandText = querySpeler;
    commandSpeler.Parameters["@id"].Value = transfer.Speler.Id;
    if (transfer.Speler.Team == null)
        commandSpeler.Parameters["@teamid"].Value = DBNull.Value;
    else
        commandSpeler.Parameters["@teamid"].Value = transfer.Speler.Team.Stamnummer;
    commandSpeler.ExecuteNonQuery();
    transaction.Commit();
    return transfer;
}

```

```

catch (Exception ex)
{
    transaction.Rollback();
    throw new TransferRepositoryADOException("SchrijfTransfer", ex);
}
finally
{
    connection.Close();
}

```

User story : pas de eigenschappen van een speler aan.

In de analyse van de user stories hebben we beslist dat deze user story zich beperkt tot de properties rugnummer, lengte en gewicht.

In de klasse SpelerManager voegen we een UpdateSpeler-methode toe waar we eerst een aantal controles uitvoeren. Zo kan een speler maar geupdated worden als hij al is opgenomen in het systeem (id verschillend van 0). Indien de speler bestaat dan vragen we aan de repository om de

update uit te voeren in de datalaag. Daarbij is het dus noodzakelijk om de ISpelerRepository uit te breiden met een UpdateSpeler-methode.

```
public void UpdateSpeler(Speler speler)
{
    if (speler == null) throw new SpelerManagerException("updatespeler - speler is null");
    if (speler.Id == 0) throw new SpelerManagerException("updatespeler - id = 0");
    try
    {
        if (repo.BestaatSpeler(speler))
        {
            repo.UpdateSpeler(speler);
        }
        else
        {
            throw new SpelerManagerException("updatespeler - speler niet gevonden");
        }
    }
    catch(SpelerManagerException)
    {
        throw;
    }
    catch(Exception ex)
    {
        throw new SpelerManagerException("updatespeler", ex);
    }
}
```

```
public interface ISpelerRepository
{
    2 references | Tom Vande Wiele, 7 days ago | 1 author, 1 change
    Speler SchrijfSpelerInDB(Speler s);
    3 references | Tom Vande Wiele, 7 days ago | 1 author, 1 change
    bool BestaatSpeler(Speler s);
    1 reference | 0 changes | 0 authors, 0 changes
    void UpdateSpeler(Speler speler);
}
```

De implementatie zelf van de methode vindt plaats in de datalaag, namelijk in de klasse SpelerRepositoryADO. We schrijven het SQL-update-statement, maken een SqlCommand aan, stellen de CommandText in en voegen de parameters toe. Daarna kan de update uitgevoerd worden met de ExecuteNonQuery-methode.

```
public void UpdateSpeler(Speler speler)
{
    SqlConnection connection = getConnection();
    string query = "UPDATE speler SET naam=@naam, rugnummer=@rugnummer,lengte=@lengte,"
        + "gewicht=@gewicht WHERE id=@id";

    connection.Open();
    using (SqlCommand command = connection.CreateCommand())
    {
        try[...]
    }
}
```

```

try
{
    command.Parameters.Add(new SqlParameter("@id", SqlDbType.Int));
    command.Parameters.Add(new SqlParameter("@naam", SqlDbType.NVarChar));
    command.Parameters.Add(new SqlParameter("@lengte", SqlDbType.Int));
    command.Parameters.Add(new SqlParameter("@gewicht", SqlDbType.Int));
    command.Parameters.Add(new SqlParameter("@rugnummer", SqlDbType.Int));
    command.CommandText = query;
    command.Parameters["@id"].Value = spelers.Id;
    command.Parameters["@naam"].Value = spelers.Naam;
    if (speler.Lengte == null)
        command.Parameters["@lengte"].Value = DBNull.Value;
    else
        command.Parameters["@lengte"].Value = spelers.Lengte;
    if (speler.Gewicht == null)
        command.Parameters["@gewicht"].Value = DBNull.Value;
    else
        command.Parameters["@gewicht"].Value = spelers.Gewicht;
    if (speler.Rugnummer == null)
        command.Parameters["@rugnummer"].Value = DBNull.Value;
    else
        command.Parameters["@rugnummer"].Value = spelers.Rugnummer;
    command.ExecuteNonQuery();
}
catch (Exception ex)
{
    throw new SpelerRepositoryADOException("UpdateSpeler", ex);
}
finally
{
    connection.Close();
}

```

User story : pas de eigenschappen van een team aan.

Een volledig analoog verhaal als bij de aanpassingen voor een speler. Ook hier geldt dat enkel de eigenschappen van een team kunnen worden aangepast en niet de relaties (daarvoor hebben we transfers). In de TeamManager-klasse voegen we een UpdateSpeler-methode toe, hierin controleren we eerst of het team wel bestaat en als dat zo is dan voeren we de update uit. Het effectief uitvoeren van de update laten we over aan de datalaag, maar we moeten wel een extra UpdateSpeler-methode toevoegen aan de ITeamRepository.

```

public void UpdateTeam(Team team)
{
    if (team == null) throw new TeamManagerException("updateteam - team is null");
    try
    {
        if (repo.BestaatTeam(team))
        {
            repo.UpdateTeam(team);
        }
        else
        {
            throw new TeamManagerException("updateteam - team niet gevonden");
        }
    }
    catch (TeamManagerException)
    {
        throw;
    }
    catch (Exception ex)
    {
        throw new TeamManagerException("updateteam", ex);
    }
}

```

```

public interface ITeamRepository
{
    2 references | Tom Vande Wiele, 7 days ago | 1 author, 1 change
    void SchrijfTeamInDB(Team t);
    3 references | Tom Vande Wiele, 7 days ago | 1 author, 1 change
    bool BestaatTeam(Team t);
    3 references | Tom Vande Wiele, 23 hours ago | 1 author, 1 change
    Team SelecteerTeam(int stamnummer);
    2 references | 0 changes | 0 authors, 0 changes
    void UpdateTeam(Team team);
}

```

De implementatie in de datalaag ziet er als volgt uit.

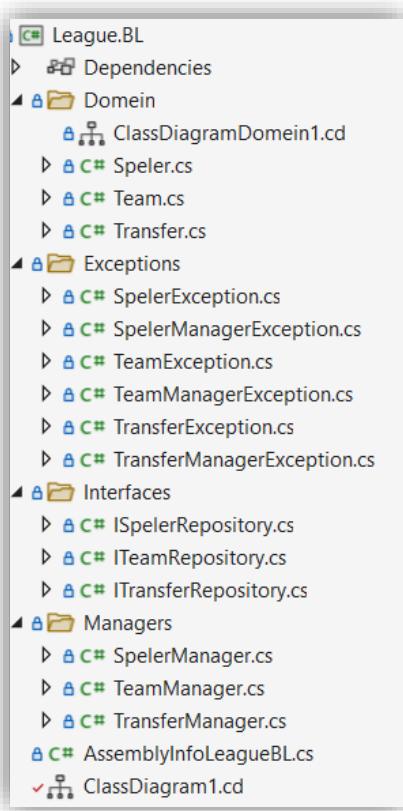
```

public void UpdateTeam(Team team)
{
    SqlConnection connection = getConnection();
    string query = "UPDATE team SET naam=@naam, bijnaam=@bijnaam WHERE stamnummer=@stamnummer";
    using (SqlCommand command = connection.CreateCommand())
    {
        connection.Open();
        try
        {
            command.Parameters.Add(new SqlParameter("@stamnummer", SqlDbType.Int));
            command.Parameters.Add(new SqlParameter("@naam", SqlDbType.NVarChar));
            command.Parameters.Add(new SqlParameter("@bijnaam", SqlDbType.NVarChar));
            command.CommandText = query;
            command.Parameters["@stamnummer"].Value = team.Stamnummer;
            command.Parameters["@naam"].Value = team.Naam;
            if (team.Bijnaam == null)
                command.Parameters["@bijnaam"].Value = DBNull.Value;
            else
                command.Parameters["@bijnaam"].Value = team.Bijnaam;
            command.ExecuteNonQuery();
        }
        catch (Exception ex)
        {
            throw new TeamRepositoryADOException("UpdateTeam", ex);
        }
        finally
        {
            connection.Close();
        }
    }
}

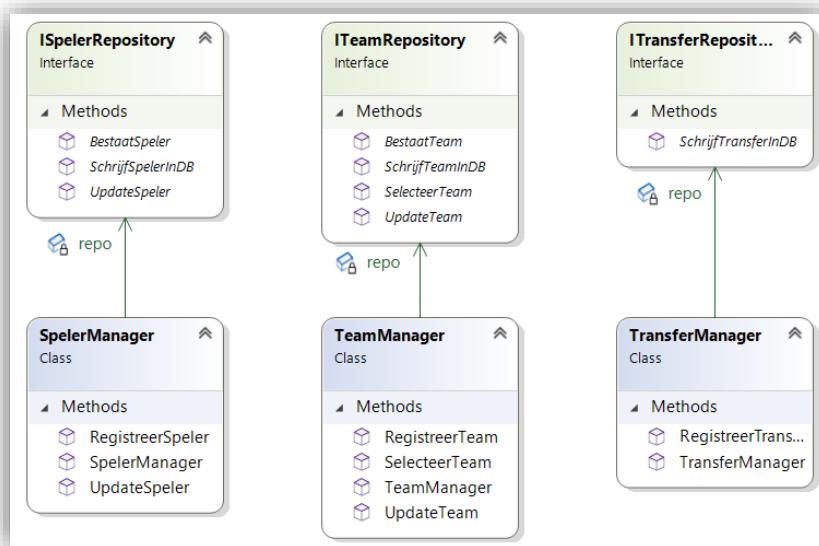
```

Stand van zaken project.

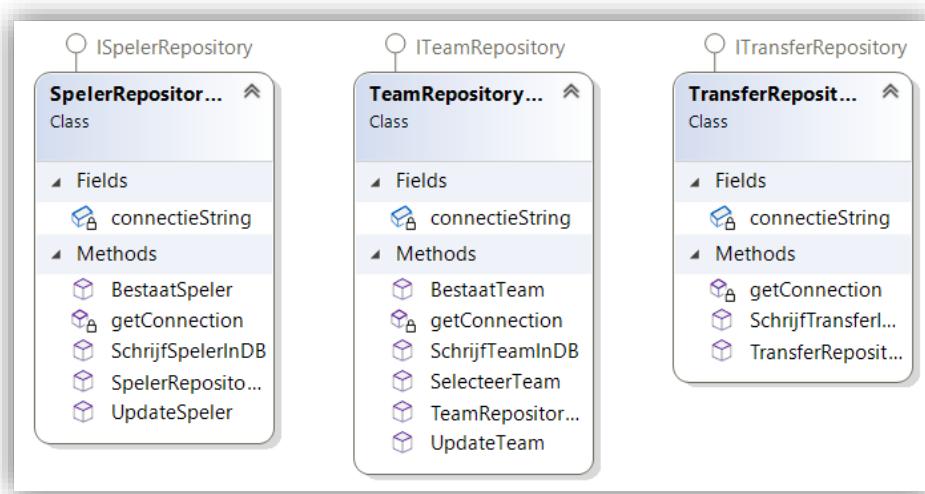
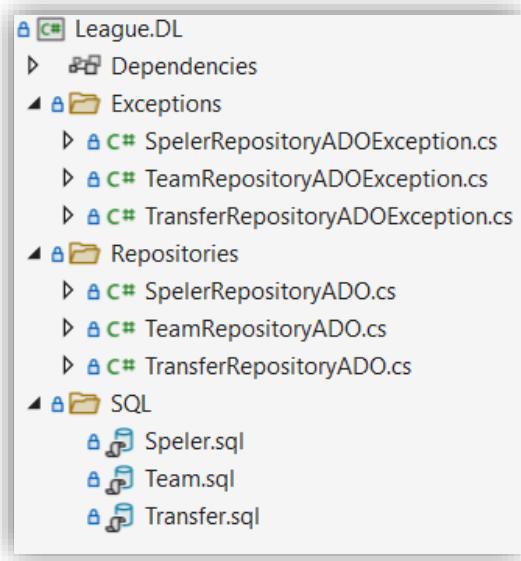
We maken even tijd om een overzicht te geven van ons project. In de businesslaag (project League.BL) hebben we naast het domein ook een folder met de exceptions, interfaces en managerklassen.



De managerklassen zelf bevatten methodes om de verschillende entiteiten toe te voegen en up te daten en verwijzen naar de interfaces die in de data laag worden geïmplementeerd.



In de dataaag (project League.DL) hebben we de repositories, de SQL-statements voor het aanmaken van het datamodel en een folder met de custom exceptions.



Applicatieontwikkeling – cyclus 2

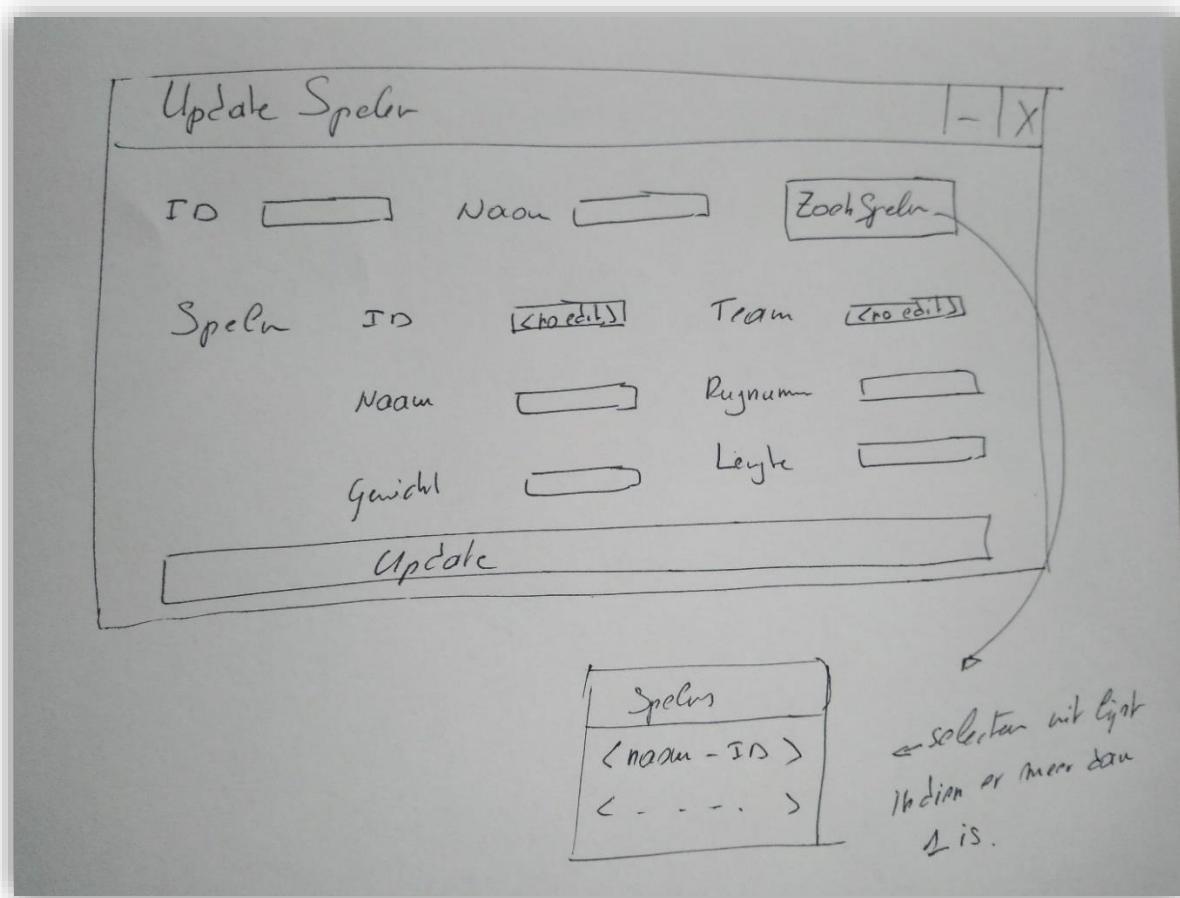
Het ontwikkelen van software gebeurt in verschillende cyclussen waarbij de grootte van de cyclus afhangt van verschillende factoren (aantal programmeurs, samenstelling team, ...). Een typische cyclus bestaat bijvoorbeeld uit een drietal weken waarin dan één of meerdere user stories worden aangepakt. Een cyclus bevat ook altijd de volgende onderdelen : analyse, ontwerp, implementatie en testen.

Waarom wordt er nu in verschillende cyclussen gewerkt, wel dat is om snel te kunnen reageren op veranderende requirements of om snel feedback te kunnen verwerken van de klant. Hoe dit concreet in zijn werk gaat is niet belangrijk voor deze cursus, wat wel van belang is, is het feit dat we moeten rekening houden met mogelijke aanpassingen en uitbreidingen.

In deze tweede cyclus gaan we nu in eerste instantie nadenken over het UI-ontwerp en de mogelijke invloeden dit kan hebben op de reeds geïmplementeerde code.

UI-ontwerp.

Laten we starten met enkele schetsen op papier van een aantal invoerschermen. Een eerste mogelijk scherm – voor het updaten van een speler - zou er als volgt kunnen uitzien. Om een speler te kunnen updaten is het in de eerste plaats noodzakelijk dat we de speler kunnen selecteren. Met andere woorden we moeten een speler kunnen opzoeken, dat kan bijvoorbeeld op basis van zijn id of naam. Wellicht is de naam meer gekend dan de id (aangezien het hier gaat om een oefening kunnen we niet echt bij de klant gaan checken wat de voorkeur is). We opteren om beide opties toe te laten. Als we zoeken op naam dan bestaat de mogelijkheid dat er meerdere spelers zijn met dezelfde naam, we zouden deze dan kunnen oplijsten in een apart venster en daarin een keuze maken. De info die we tonen van de speler zijn : id, naam, gewicht, lengte, rugnummer en de naam van het team. Het wijzigen van de id en het team laten we niet toe dus die velden maken we niet editeerbaar.

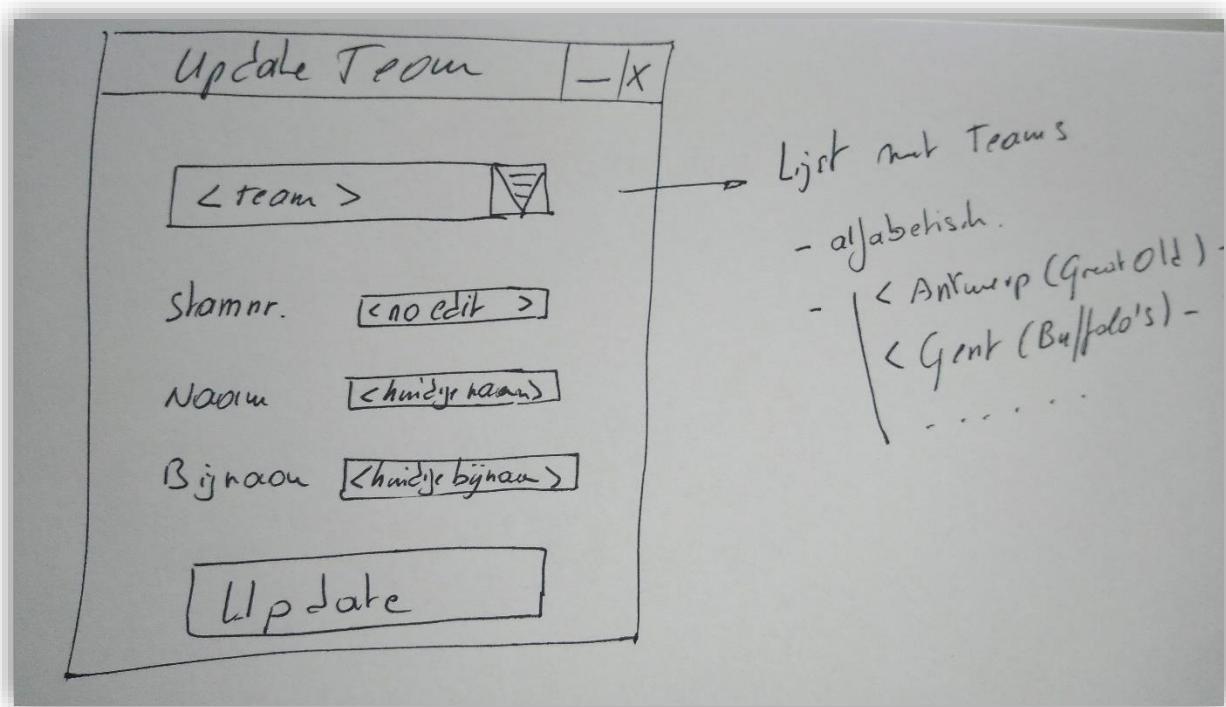


Wat hebben we nu nodig in de business-laag om dit venster te kunnen implementeren ?

- Methode om een speler op te zoeken op basis van zijn naam (kan meerdere spelers teruggeven).
- Methode om een speler op te zoeken op basis van zijn id.
- Methode om de update uit te voeren.

Opmerking : van een speler hebben we de properties (id, naam, gewicht, lengte en rugnummer) nodig en de naam van het team waartoe hij behoort (als er een team is). In tegenstelling tot het domein-model is er geen behoefte om een object team te hebben (met alle spelers) enkel de naam (of combinatie naam – bijnaam – stamnummer) volstaat.

Laten we een tweede scherm ontwerpen, deze keer voor het updaten van de team-eigenschappen. Een team heeft slechts twee eigenschappen die kunnen worden aangepast, namelijk de naam en de bijnaam. Net zoals bij het updaten van een speler moeten we eerst een team selecteren dan we wensen te updaten. Aangezien het aantal teams veel beperkter is dan het aantal spelers zouden we kunnen opteren om een dropdown-lijst te maken van de teams in plaats van een zoek-methode. Ook hier kan je best overleggen met de klant en bekijken wat de reële situatie is (zijn er honderden/duizenden teams dan zal er een extra filter nodig zijn – vb op competitie). Als er een team geselecteerd is worden de eigenschappen ingevuld en maken we de knop update beschikbaar.

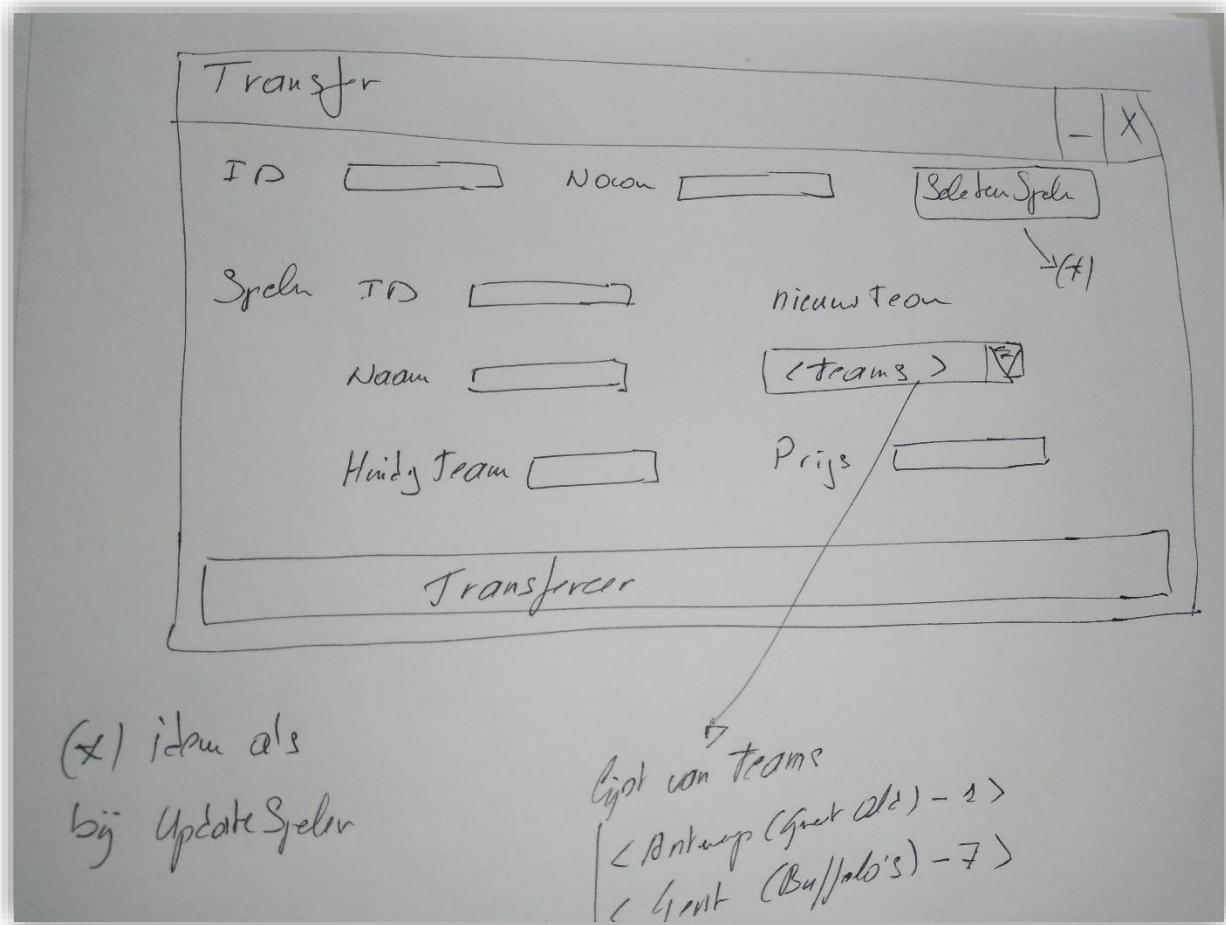


Ook hier stellen we de vraag wat hebben we nodig van onze business-laag.

- Een lijst met de verschillende teams (gesorteerd op naam). Een volledig object team uit het domein is hier niet nodig, enkel een combinatie naam-bijnaam-stamnummer. Opmerking : stel dat we alle teams hier oplijsten (volgens de domein-definitie, dan moeten we ook telkens alle spelers ophalen en dat is op zich geen efficiënte aanpak van ons probleem).
- Een update-methode voor het aanpassen van de team-eigenschappen.

Een derde venster dat we wensen te behandelen is voor het doorvoeren van een transfer. Wat hebben we allemaal nodig voor het definiëren van een transfer ? Dat is in de eerste plaats de speler die we wensen te transfereren. We kunnen dat op dezelfde manier doen dan bij het updaten van een speler, namelijk zoeken op id of naam. In het venster kunnen we dan de eigenschappen van de speler tonen (samen met zijn huidige team). Verder moet er een nieuw team gekozen worden (op geen team als de speler stopt) en moet er een prijs worden ingevuld. Het selecteren van het team

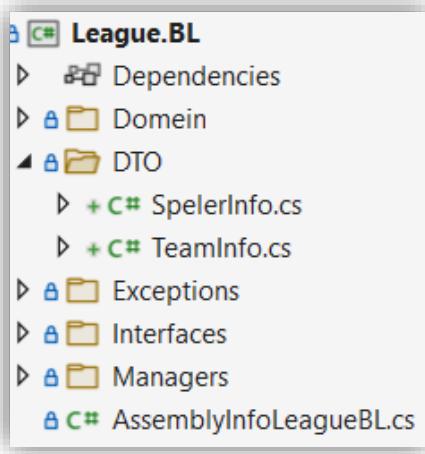
kan op analoge wijze als in het venster update team. Met de knop transfereer kunnen we dan de uiteindelijk transfer uitvoeren.



Net zoals bij de voorgaande venster hebben we een methode nodig die de namen van de teams oplijst, een methode voor een speler op te zoeken en een methode om de transfer uit te voeren.

Data Transfer Objects (DTO)

Na de UI-analyse zijn we tot de conclusie gekomen dat de UI nood heeft aan data, maar dat hoeven niet per se de domein-objecten te zijn. Objecten die enkel dienen om data door te geven tussen verschillende lagen noemen we Data Transfer Objecten of DTOs. In ons geval hebben we er twee nodig, namelijk één met info over de speler en één met info over het team. We voorzien in ons business-laag project een extra folder (DTO) met daarin deze twee klassen.



De klassen zelf zijn zeer eenvoudig en bevatten enkel een aantal properties en een constructor. De klasse TeamInfo bevat enkel het stamnummer, naam en bijnaam. Hier zijn geen verwijzingen naar spelers en er zijn ook geen kwaliteitsvoorwaarden die we afdwingen, dat hoort namelijk tot ons domein. Deze klassen dienen enkel om gegevens door te geven.

```
public class TeamInfo
{
    public TeamInfo(int stamnummer, string naam, string bijnaam)...
    public int Stamnummer { get; set; }
    public string Naam { get; set; }
    public string Bijnaam { get; set; }
}
```

In de klasse SpelerInfo is er wel nog een verwijzing naar Team, maar niet naar het object team, we hebben enkel maar de naam (of combinatie naam, bijnaam en stamnummer) van het team nodig om te tonen in de UI.

```
public class SpelerInfo
{
    public SpelerInfo(int id, string naam, string team, int? rugnummer, int? lengte, int? gewicht)...
    public int Id { get; set; }
    public string Naam { get; set; }
    public string Team { get; set; }
    public int? Rugnummer { get; set; }
    public int? Lengte { get; set; }
    public int? Gewicht { get; set; }
}
```

Selecteer spelers.

Zoals uit de analyse is gebleken, hebben we dus in de UI een methode SelecteerSpelers nodig die een lijst kan geven met de spelersinfo. Dat maakt dat we in de klasse spelermanager hiervoor een methode moeten voorzien. De return-waarde is een IReadOnlyList van SpelerInfo en de selectiecriteria zijn ofwel een spelerid ofwel een naam. In het eerste geval zal er altijd maar één speler worden teruggegeven in het tweede geval kunnen dat meerdere spelers zijn. Het opvragen van de spelers uit de databank laten we over aan de datalaag, we moeten natuurlijk wel een methode definiëren in de ISpelerRepository die dan moet worden geïmplementeerd.

```
public IReadOnlyList<SpelerInfo> SelecteerSpelers(int? id, string naam)
{
    if ((id == null) && (string.IsNullOrWhiteSpace(naam)))
        throw new SpelerManagerException("SelecteerSpelers - no input");
    try
    {
        return repo.SelecteerSpelers(id, naam);
    }
    catch (SpelerManagerException)
    {
        throw;
    }
    catch (Exception ex)
    {
        throw new SpelerManagerException("SelecteerSpelers", ex);
    }
}
```

```
public interface ISpelerRepository
{
    2 references | Tom Vande Wiele, 11 days ago | 1 author, 1 change
    Speler SchrijfSpelerInDB(Speler s);
    3 references | Tom Vande Wiele, 11 days ago | 1 author, 1 change
    bool BestaatSpeler(Speler s);
    2 references | Tom Vande Wiele, 3 days ago | 1 author, 1 change
    void UpdateSpeler(Speler speler);
    1 reference | 0 changes | 0 authors, 0 changes
    IReadOnlyList<SpelerInfo> SelecteerSpelers(int? id, string naam);
}
```

In de SpelerRepositoryADO klasse implementeren we dan deze methode. In de eerste plaats hebben we een SQL-statement nodig die ons de nodige info kan bezorgen over de spelers. Naast de properties van de speler hebben we ook de naam van het team nodig en zoals in het UI-design werd weergegeven, gaan we niet enkel de naam geven, maar een combinatie van naam, bijnaam en stamnummer. Indien een speler niet tot een team behoort geven we een null-waarde. In het SQL-statement maken we gebruik van een case constructie en gebruiken we concat om de namen aan elkaar te plakken. Afhankelijk of er wel of niet een id wordt meegegeven passen we de WHERE-filter aan.

```

public IReadOnlyList<SpelerInfo> SelecteerSpelers(int? id, string naam)
{
    if ((!id.HasValue) && (string.IsNullOrEmpty(naam) == true))
        throw new SpelerRepositoryADOException("SelecteerSpelers - no valid input");
    string query = "SELECT Id,t1.Naam,Rugnummer,Lengte,Gewicht, "
                  + " case when t2.Stamnummer is null then null "
                  + "       else concat(t2.Naam, ' (' , t2.Bijnaam, ') - ', t2.Stamnummer) "
                  + " end teamnaam "
                  + " FROM Speler t1 "
                  + " left join team t2 on t1.teamid = t2.Stamnummer";
    if (id.HasValue) query += " WHERE t1.id=@id";
    else query += " WHERE t1.naam = @naam";
    List<SpelerInfo> spelers=new List<SpelerInfo>();
    SqlConnection connection = getConnection();
    using (SqlCommand command = connection.CreateCommand())
    ...
}

```

Ook voor het invullen van de parameters kijken we of we op id of naam moeten filteren.

```

using (SqlCommand command = connection.CreateCommand())
{
    if (id.HasValue)
    {
        command.Parameters.Add(new SqlParameter("@id", SqlDbType.Int));
        command.Parameters["@id"].Value = id;
    }
    else
    {
        command.Parameters.Add(new SqlParameter("@naam", SqlDbType.VarChar));
        command.Parameters["@naam"].Value = naam;
    }
    command.CommandText = query;
    connection.Open();
    try
    ...
}

```

Aangezien er voor verschillende properties null-waarden mogelijk zijn moeten we daar dan ook rekening mee houden bij het opvragen van de records. Checken of er een null-waarde is kan met de methode IsDBNull van de DataReader. Daarbij moeten we het kolomnummer meegeven en dat kunnen we opvragen met de methode GetOrdinal waarbij we de naam van de kolom meegeven. Als alle nodige gegevens uit de reader zijn gehaald, kunnen we een SpelerInfo-object aanmaken en toevoegen aan onze lijst. We gebruiken een lijst omdat er meerdere spelers kunnen zijn als we zoeken op naam.

```

try
{
    IDataReader reader = command.ExecuteReader(); //of SqlDataReader
    while (reader.Read())
    {
        string teamnaam = null;
        if (!reader.IsDBNull(reader.GetOrdinal("teamnaam"))) teamnaam = (string)reader["teamnaam"];
        int? lengte = null;
        if (!reader.IsDBNull(reader.GetOrdinal("lengte"))) lengte = (int?)reader["lengte"];
        int? gewicht = null;
        if (!reader.IsDBNull(reader.GetOrdinal("gewicht"))) gewicht = (int?)reader["gewicht"];
        int? rugnummer = null;
        if (!reader.IsDBNull(reader.GetOrdinal("rugnummer"))) rugnummer = (int?)reader["rugnummer"];
        SpelerInfo spelers = new SpelerInfo((int)reader["id"], (string)reader["naam"], teamnaam, rugnummer, lengte, gewicht);
        spelers.Add(speler);
    }
    reader.Close();
    return spelers;
}
catch (Exception ex)
{
    throw new SpelerRepositoryADOException("selecteerSpelers", ex);
}
finally
{
    connection.Close();
}

```

Selecteer teams.

Voor het selecteren van de teams voorzien we in de TeamManager-klasse een methode SelecteerTeams dat een IReadOnlyList teruggeeft van TeamInfo. De methode zelf is zeer beperkt en geeft de opdracht enkel door aan de repository.

```

public IReadOnlyList<TeamInfo> SelecteerTeams()
{
    try
    {
        return repo.SelecteerTeams();
    }
    catch (Exception ex)
    {
        throw new TeamManagerException("SelecteerTeams");
    }
}

```

Dit betekent dat we de interface ITeamRepository moeten uitbreiden als volgt.

```

public interface ITeamRepository
{
    void SchrijfTeamInDB(Team t);
    Team SelecteerTeam(int stamnummer);
    void UpdateTeam(Team team);
    IReadOnlyList<TeamInfo> SelecteerTeams();
}

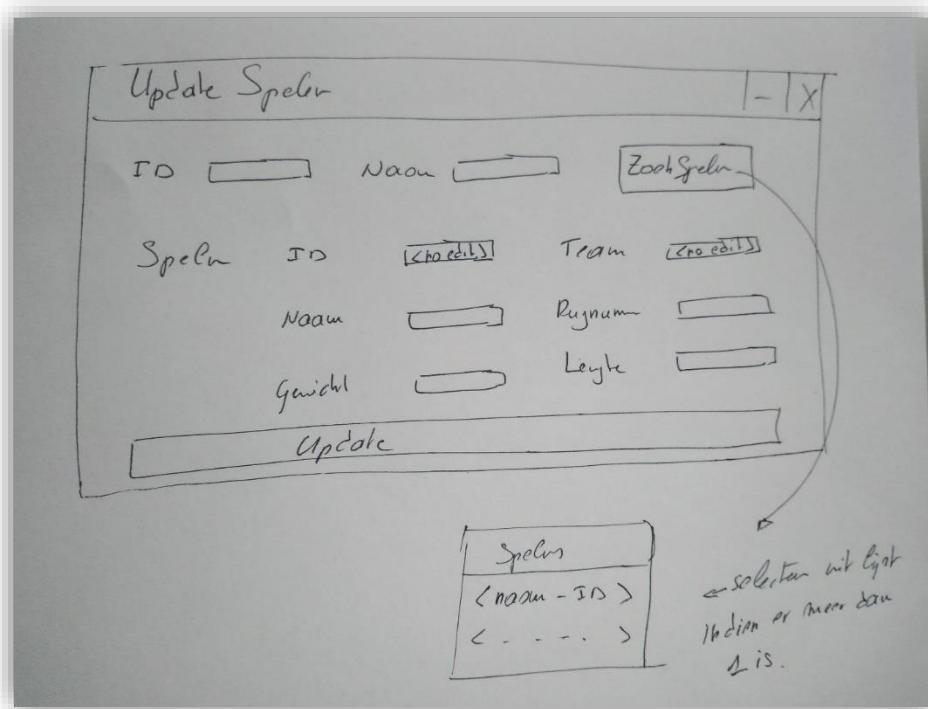
```

De implementatie van de methode in de datalaag (klasse TeamRepositoryADO) bevat een eenvoudige query die alle teams ophaalt. Er moet enkel rekening gehouden worden met de bijnaam die eventueel null kan zijn.

```
public IReadOnlyList<TeamInfo> SelecteerTeams()
{
    SqlConnection connection = getConnection();
    string query = "SELECT stamnummer,naam,bijnaam FROM [dbo].[Team]";
    List<TeamInfo> teams=new List<TeamInfo>();
    using (SqlCommand command = connection.CreateCommand())
    {
        command.CommandText=query;
        connection.Open();
        try
        {
            IDataReader reader = command.ExecuteReader(); //of SqlDataReader
            while (reader.Read())
            {
                string bijnaam = null;
                if (!reader.IsDBNull(reader.GetOrdinal("bijnaam")))
                    bijnaam = (string)reader["bijnaam"];
                teams.Add(new TeamInfo((int)reader["stamnummer"], (string)reader["naam"], bijnaam));
            }
            reader.Close();
            return teams;
        }
        catch (Exception ex)
        {
            throw new TeamRepositoryADOException("selecteerTeams", ex);
        }
        finally
        {
            connection.Close();
        }
    }
}
```

User story : pas de eigenschappen van een speler aan.

We bekijken nog eens het UI-ontwerp voor het updaten van een speler. We gaan een speler selecteren door te kiezen uit een lijst met daarin SpelerInfo-objecten. Na het invullen van de spelereigenschappen selecteren we de update-knop en wensen we dat de update wordt uitgevoerd.



Het effectief uitvoeren van de update vragen we aan de SpelerManager-klasse in de businesslaag. We hadden daarvoor een methode voorzien die er als volgt uit ziet.

```
public void UpdateSpeler(Speler speler)
```

Maar aangezien we nu niet met Speler-objecten werken, maar met SpelerInfo-objecten gaan we de signatuur van de methode aanpassen en Speler vervangen door het DTO-object SpelerInfo. Ook in de update-methode zelf doen we een aantal aanpassingen.

Aangezien we nog geen Speler-object hebben kunnen we ook de huidige BestaatSpeler-methode in de repository niet gebruiken. We hebben dus nood aan een BestaatSpeler-methode die enkel een id gebruikt om de controle uit te voeren. Het is nu even tijd om stil te staan bij de BestaatSpeler-logica. We hebben feitelijk twee verschillende situaties.

We wensen te vermijden dat een speler tweemaal wordt geregistreerd, maar dat kunnen we niet doen op basis van een id omdat bij het registreren er nog geen id is toegekend. Vandaar dat we hadden gekozen om de naam van de speler te gebruiken, maar dat is feitelijk niet helemaal correct ! Om helemaal correct te zijn zouden er meerdere properties moeten zijn waarmee we onderscheid kunnen maken tussen verschillende spelers (vb geboortedatum, geboorteplaats, ...).

Wanneer we BestaatSpeler uitvoeren in de update-methode dan dient deze methode om te controleren of de speler wel degelijk in ons systeem zit en dan is het voldoende om de id te controleren. Vandaar dat we twee verschillende BestaatSpeler-methoden zullen opnemen in ISpelerRepository interface.

```

public void UpdateSpeler(SpelerInfo spelerInfo)
{
    if (spelerInfo == null) throw new SpelerManagerException("updatespeler - spelerInfo is null");
    if (spelerInfo.Id == 0) throw new SpelerManagerException("updatespeler - id = 0");
    try
    {
        if (repo.BestaatSpeler(spelerInfo.Id))...
        else
        {
            throw new SpelerManagerException("updatespeler - speler niet gevonden");
        }
    }
    catch (SpelerManagerException)
    {
        throw;
    }
    catch (Exception ex)
    {
        throw new SpelerManagerException("updatespeler", ex);
    }
}

```

```

if (repo.BestaatSpeler(spelerInfo.Id))
{
    Speler speler = repo.SelecteerSpeler(spelerInfo.Id);
    bool changed = false;
    if (speler.Naam != spelerInfo.Naam)
    {
        speler.ZetNaam(spelerInfo.Naam);
        changed = true;
    }
    if (((speler.Lengte.HasValue) && (speler.Lengte != spelerInfo.Lengte))
        || ((spelerInfo.Lengte.HasValue) && (!speler.Lengte.HasValue)))
    {
        speler.ZetLengte((int)spelerInfo.Lengte);
        changed = true;
    }
    if (((speler.Gewicht.HasValue) && (speler.Gewicht != spelerInfo.Gewicht))
        || ((spelerInfo.Gewicht.HasValue) && (!speler.Gewicht.HasValue)))
    {
        speler.ZetGewicht((int)spelerInfo.Gewicht);
        changed = true;
    }
    if (((speler.Rugnummer.HasValue) && (speler.Rugnummer != spelerInfo.Rugnummer))
        || ((spelerInfo.Rugnummer.HasValue) && (!speler.Rugnummer.HasValue)))
    {
        speler.ZetRugnummer((int)spelerInfo.Rugnummer);
        changed = true;
    }
}
if (!changed) throw new SpelerManagerException("UpdateSpeler - no changes");
repo.UpdateSpeler(speler);
}

```

De logica in de UpdateSpeler-methode van de businesslaag checkt eerst of er een correct SpelerInfo-object is meegegeven (niet null en id ingevuld). Daarna wordt er gecontroleerd of de speler wel in het systeem zit. Eénmaal we weten dat de speler bestaat kunnen we deze ook opvragen en de nieuwe waarden instellen. Om af te sluiten geven we aan de repository de opdracht om de update effectief uit te voeren in de databank.

De ISpelerRepository ziet er ondertussen als volgt uit.

```

public interface ISpelerRepository
{
    2 references | Tom Vande Wiele, 20 days ago | 1 author, 1 change
    Speler SchrijfSpelerInDB(Speler s);
    2 references | Tom Vande Wiele, 20 days ago | 1 author, 1 change
    bool BestaatSpeler(Speler s);
    1 reference | 0 changes | 0 authors, 0 changes
    bool BestaatSpeler(int spelervId);
    1 reference | 0 changes | 0 authors, 0 changes
    Speler SelecteerSpeler(int id);
    2 references | Tom Vande Wiele, 12 days ago | 1 author, 1 change
    void UpdateSpeler(Speler spelerv);
    2 references | Tom Vande Wiele, 9 days ago | 1 author, 1 change
    IReadOnlyList<SpelerInfo> SelecteerSpelers(int? id, string naam);
}

```

De nieuwe BestaatSpeler-methode is helemaal analoog aan de reeds bestaande, alleen werken we nu met een id in plaats van een naam. De code ziet er als volgt uit :

```

public bool BestaatSpeler(int spelervId)
{
    SqlConnection connection = getConnection();
    string query = "SELECT count(*) FROM dbo.Speler WHERE id=@id";

    using (SqlCommand command = connection.CreateCommand())
    {
        connection.Open();
        try
        {
            command.Parameters.Add(new SqlParameter("@id", SqlDbType.Int));
            command.CommandText = query;
            command.Parameters["@id"].Value = spelervId;
            int n = (int)command.ExecuteScalar();
            if (n > 0) return true;
            else return false;
        }
        catch (Exception ex)
        {
            throw new SpelerRepositoryADOException("BestaatSpeler", ex);
        }
        finally
        {
            connection.Close();
        }
    }
}

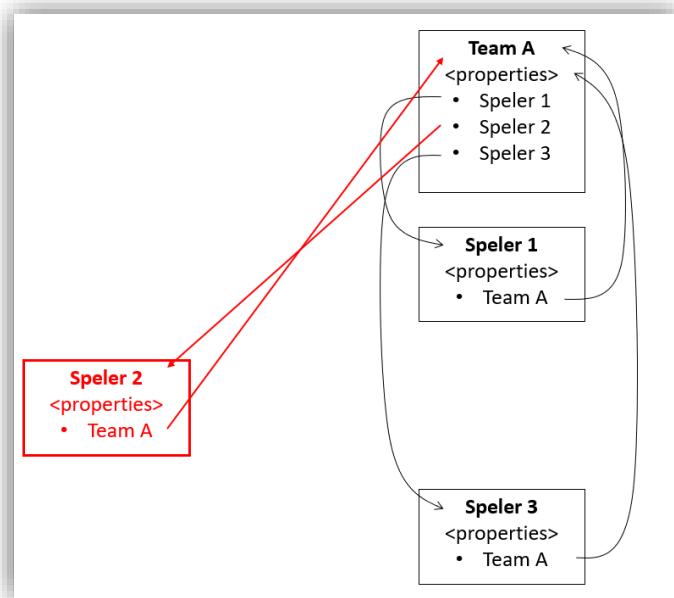
```

We moeten nu nog de SelecteerSpeler-methode implementeren. Hier moeten we eerst eens goed over nadenken, voor de UpdateSpeler-methode is het voldoende om enkel de properties op te halen en is de relatie met Team niet echt nodig. Maar de SelecteerSpeler-methode zal ook nog gebruikt moeten worden bij het registreren van een transfer en daar is het wel degelijk noodzakelijk om de relatie met Team op te nemen. We kiezen dus voor de laatste implementatie waarbij we een speler-object met de relatie naar team gaan opvragen.

De eerste moeilijkheid is het schrijven van het SQL-statement waarmee we in één keer alle info kunnen opvragen. We hebben nodig :

- De eigenschappen van de speler.
- De eigenschappen van het team waartoe de speler behoort (als hij tot een team behoort)
- De spelers die bij dat team behoren.

Schematisch kunnen we het als volgt weergeven, we selecteren een speler (speler 2) die verwijst naar het team-object waartoe hij behoort. Maar dit team-object zelf moet ook alle spelers bevatten die tot dit team behoren.



Het SQL-statement dat daarvoor kan dienen ziet er als volgt uit :

```

select ts.id speleraid,ts.naam spelernaam,ts.rugnummer spelerrugnummer,ts.lengte spelerlengte,
       ts.gewicht spelergewicht,ts.teamid spelerteamid,tt.*
from speler ts
left join (
    SELECT t1.stamnummer,t1.naam as ploegnaam,t1.bijnaam,t2.*
    FROM [dbo].[Team] t1 left join [dbo].[speler] t2 on t1.Stamnummer = t2.teamid
) tt
    on tt.stamnummer = ts.teamid
where ts.id = 3;

```

We gebruiken een subquery die voor elk team alle eigenschappen van het team oplijst en alle spelers die daartoe behoren.

```

SELECT t1.stamnummer,t1.naam as ploegnaam,t1.bijnaam,t2.*
FROM [dbo].[Team] t1 left join [dbo].[speler] t2 on t1.Stamnummer = t2.teamid

```

Het tussenresultaat van deze subquery ziet er zo uit :

stamnummer	ploegnaam	bijnaam	Id	Naam	Rugnummer	Lengte	Gewicht	TeamId
1	Antwerp	The Great Old	3	jos	NULL	185	85	1
1	Antwerp	The Great Old	5	jeff	NULL	177	79	1
7	Gantoise	Buffalo's	1	inge	10	172	69	7
5381	Zulte Waregem	Essevee	NULL	NULL	NULL	NULL	NULL	NULL

We joinen nu de spelers-tabel met de subquery op basis van het stamnummer en het teamid (van speler) en geven via de WHERE aan over welke speler het gaat. Verder gebruiken we nog een aantal aliassen om verwarring met de kolommen te vermijden want uiteindelijk tonen we de speler-tabel twee keer in het resultaat.

spelerid	spelernaam	spelerrugnummer	spelerlengte	spelergewicht	spelerteamid	stamnummer	ploegnaam	bijnaam	Id	Naam	Rugnummer	Lengte	Gewicht	TeamId
3	jos	NULL	185	85	1	1	Antwerp	The Great Old	3	jos	NULL	185	85	1
3	jos	NULL	185	85	1	1	Antwerp	The Great Old	5	jeff	NULL	177	79	1

De SelecteerSpeler-methode zelf bevat dus deze query met als enige parameter de id van de speler.

```
public Speler SelecteerSpeler(int id)
{
    SqlConnection connection = getConnection();
    string query = "select ts.id speleraid,ts.naam spelernaam,ts.rugnummer spelerrugnummer," +
        "+ ts.lengte speleralengte,ts.gewicht spelergewicht,ts.teamid spelerteamid,tt.* " +
        "+ from speler ts " +
        "+ left join (" +
        "+     SELECT t1.stamnummer,t1.naam as ploegnaam,t1.bijnaam,t2.* " +
        "+     FROM[dbo].[Team] t1 left join[dbo].[speler] t2 on t1.Stamnummer = t2.teamid " +
        "+ ) tt on tt.stamnummer = ts.teamid " +
        "+ where ts.id = @id";
    using (SqlCommand command = connection.CreateCommand())
    {
        command.Parameters.Add(new SqlParameter("@id", SqlDbType.Int));
        command.CommandText = query;
        command.Parameters["@id"].Value = id;
        connection.Open();
        try{...}
    }
}
```

Het inlezen/analyseren van de query-output is nu wel een stukje ingewikkelder. We voorzien een speler- en een team-object dat we initieel op null instellen. Het speler-object is het object dat we gaan teruggeven. De manier waarop we de resultaten verwerken is analoog met de manier waarop we dat voor het selecteren van een team hebben gedaan. We doorlopen alle records in de while-lus en daarin onderscheiden we drie code-blokken.

```

try
{
    Team team = null;
    Speler speler = null;
    IDataReader reader = command.ExecuteReader(); //of SqlDataReader
    while (reader.Read())
    {
        if (speler == null)...
        if (team == null)...
        if (!reader.IsDBNull(reader.GetOrdinal("id")))...
    }
    reader.Close();
    return speler;
}
catch (Exception ex)
{
    throw new SpelerRepositoryADOException("selecteerSpeler", ex);
}
finally
{
    connection.Close();
}

```

Het eerste code-blok controleert of er al een speler-object is aangemaakt. De bedoeling is dat we info over de speler slechts één maal gaan verwerken. Van zodra dit object is aangemaakt zal het if-statement er voor zorgen dat we deze code slechts éénmaal moeten uitvoeren. We moeten hier enkel nog rekening houden met eventuele null-waarden in de databank. Indien er geen teamid is voor de speler (speler heeft dus geen team) dan kunnen we ook direct afronden en het speler-object teruggeven.

```

if (speler == null)
{
    int? lengte = null;
    if (!reader.IsDBNull(reader.GetOrdinal("spelerlengte"))) lengte = (int?)reader["spelerlengte"];
    int? gewicht = null;
    if (!reader.IsDBNull(reader.GetOrdinal("spelergewicht"))) gewicht = (int?)reader["spelergewicht"];
    speler = new Speler(id, (string)reader["spelernaam"], lengte, gewicht);
    if (!reader.IsDBNull(reader.GetOrdinal("spelerrugnummer")))
        speler.ZetRugnummer((int)reader["spelerrugnummer"]);
    if (reader.IsDBNull(reader.GetOrdinal("spelerteamid"))) return speler;
}

```

In het volgende blokje verwerken we de team-info, ook hier wensen we deze data maar éénmaal te verwerken (de code is volledig analoog als bij het selecteren van een team).

```

if (team == null)
{
    string naam = (string)reader["ploegnaam"];
    string bijnaam = null;
    if (!reader.IsDBNull(reader.GetOrdinal("bijnaam"))) bijnaam = (string)reader["bijnaam"];
    team = new Team((int)reader["stamnummer"], naam);
    if (bijnaam != null) team.ZetBijnaam(bijnaam);
}

```

In het laatste code-blok verwerken we de data van de verschillende spelers die tot het team behoren. Ook hier is de code analoog als bij het opbouwen van het team-object. Het enige verschil is de laatste lijn code, daar vergelijken we de spelerid van de speler die we nu aan het verwerken zijn met de speler die we aan het opzoeken zijn (onze te zoeken speler behoort namelijk tot dit team). Als de team-speler dezelfde is als de te zoeken speler dan stellen we de te zoeken speler gelijk aan deze speler (op die manier is ook de link naar het team in orde gebracht voor onze te zoeken speler – want dat was nog niet in orde gebracht).

```
if (!reader.IsDBNull(reader.GetOrdinal("id")))
{
    int? lengte = null;
    if (!reader.IsDBNull(reader.GetOrdinal("lengte"))) lengte = (int?)reader["lengte"];
    int? gewicht = null;
    if (!reader.IsDBNull(reader.GetOrdinal("gewicht"))) gewicht = (int?)reader["gewicht"];
    int sid = (int)reader["id"];
    Speler s = new Speler(sid, (string)reader["naam"], lengte, gewicht);
    s.ZetTeam(team);
    if (!reader.IsDBNull(reader.GetOrdinal("rugnummer")))
        speler.ZetRugnummer((int)reader["rugnummer"]);
    if (sid == id) speler = s;
}
```

User story : pas de eigenschappen van een team aan.

Voor het aanpassen van de eigenschappen van een team hadden we reeds de methode UpdateTeam voorzien in de TeamManager-kLASSE waarbij we aan parameter team meegaven.

```
public void UpdateTeam(Team team)
```

De info die we nu krijgen vanuit de UI-laag bevat echter geen Team-object maar een TeamInfo-object. We passen dus de signatuur van de UpdateTeam-methode aan. Verder nemen we nog een aantal checks op, het teamInfo-object mag niet null zijn en de naam van het team mag geen lege string zijn. Ook de BestaatTeam-methode moet worden aangepast, we hebben namelijk geen Team-object ter beschikking, maar wel het stamnummer. Dus passen we de BestaatTeam-methode aan zodat deze met het stamnummer werkt. Deze aanpassing heeft ook een impact op andere methoden (zie verder). Eénmaal we weten dat het team bestaat, kunnen we het team ook opvragen en de nodige aanpassingen uitvoeren. We gaan hier niet direct de update-methode van de repository uitvoeren, maar gebruiken de methodes uit de business-laag om ons team-object aan te passen. Op deze manier zorgen we ervoor dat de business-regels worden gevolgd.

```

public void UpdateTeam(TeamInfo teamInfo)
{
    if (teamInfo == null) throw new TeamManagerException("updateteam - team is null");
    if (string.IsNullOrWhiteSpace(teamInfo.Naam)) throw new TeamManagerException("updateteam - teamnaam is null");
    try
    {
        if (repo.BestaatTeam(teamInfo.Stamnummer))
        {
            Team team=repo.SelecteerTeam(teamInfo.Stamnummer);
            team.ZetNaam(teamInfo.Naam);
            if (!string.IsNullOrEmpty(teamInfo.Bijnaam)) team.ZetBijnaam(teamInfo.Bijnaam);
            else team.VerwijderBijnaam();
            repo.UpdateTeam(team);
        }
        else
        {
            throw new TeamManagerException("updateteam - team niet gevonden");
        }
    }
    catch (TeamManagerException)
    {
        throw;
    }
    catch (Exception ex)
    {
        throw new TeamManagerException("updateteam", ex);
    }
}

```

Een mogelijkheid die we bij de implementatie van ons domein hebben over het hoofd gezien is het verwijderen van de bijnaam. We hebben in de klasse Team enkel een methode ZetBijnaam en deze laat niet toe om een null-waarde in te voeren. Daarom voorzien we een nieuwe (extra) methode in de klasse Team waarbij we specifiek aangeven dat we de bijnaam wensen te verwijderen.

```

public void VerwijderBijnaam()
{
    Bijnaam = null;
}

```

Zoals reeds gezegd passen we de BestaatTeam-methode aan en dus ook de ITeamRepository interface en de RegistreerTeam-methode (klasse TeamManager) die er gebruik van maakt.

```
public interface ITeamRepository
{
    2 references | Tom Vande Wiele, 22 days ago | 1 author, 1 change
    void SchrijfTeamInDB(Team t);
    2 references | 0 changes | 0 authors, 0 changes
    bool BestaatTeam(int stamnummer);
    2 references | Tom Vande Wiele, 16 days ago | 1 author, 1 change
    Team SelecteerTeam(int stamnummer);
    2 references | Tom Vande Wiele, 15 days ago | 1 author, 1 change
    void UpdateTeam(Team team);
    2 references | Tom Vande Wiele, 14 hours ago | 1 author, 1 change
    IReadOnlyList<TeamInfo> SelecteerTeams();
}
```

```
public void RegistreerTeam(int stamnummer, string naam, string bijnaam)
{
    try
    {
        Team t = new Team(stamnummer, naam);
        if (!string.IsNullOrWhiteSpace(bijnaam)) t.ZetBijnaam(bijnaam);
        if (!repo.BestaatTeam(stamnummer))
        {
            repo.SchrijfTeamInDB(t);
        }
        else
        {
            throw new TeamManagerException("RegistreerTeam - team bestaat al");
        }
    }
    catch (TeamManagerException)
    {
        throw;
    }
    catch (Exception ex)
    {
        throw new TeamManagerException("RegistreerTeam", ex);
    }
}
```

De BestaatTeam-methode in de klasse TeamRepositoryADO moet eveneens worden aangepast, maar deze aanpassingen zijn minimaal.

```

public bool BestaatTeam(int stamnummer)
{
    SqlConnection connection = getConnection();
    string query = "SELECT count(*) FROM dbo.Team WHERE stamnummer=@stamnummer";

    using (SqlCommand command = connection.CreateCommand())
    {
        connection.Open();
        try
        {
            command.Parameters.Add(new SqlParameter("@stamnummer", SqlDbType.Int));
            command.CommandText = query;
            command.Parameters["@stamnummer"].Value = stamnummer;

            int n = (int)command.ExecuteScalar();
            if (n > 0) return true;
            else return false;
        }
        catch (Exception ex)
        {
            throw new TeamRepositoryADOException("BestaatTeam", ex);
        }
        finally
        {
            connection.Close();
        }
    }
}

```

User story : voeg een transfer toe aan het systeem.

Ook hier zijn er een aantal aanpassingen noodzakelijk, de invoer van de RegistreerTransfer-methode krijgt nu namelijk een SpelerInfo- en TeamInfo-object mee in plaats van een Speler- en Team-object. We controleren eerst of de spelerInfo in orde is (mag niet null zijn en de id moet ingevuld zijn).

```

public Transfer RegistreerTransfer(SpelerInfo spelerInfo, TeamInfo nieuwTeamInfo, int prijs)
{
    if (spelerInfo == null) throw new TransferManagerException("RegistreerTransfer - speler is null");
    if (spelerInfo.Id == 0) throw new TransferManagerException("RegistreerTransfer - spelerid =0");
    Transfer transfer = null;
    try{...}
}

```

Aan de logica zelf moeten we niet zoveel veranderen, maar doordat we nu geen Team- en Speler-object ontvangen moeten we deze eerst opvragen. Het opvragen van een Team-object hebben we voor zien in de TeamRepositoryADO-klasse (of eender welke klasse die de interface ITeamRepository implementeert). Voor het opvragen van een speler kunnen we dan weer gebruik maken van een klasse die ISpelerRepository implementeert. Dit betekent wel dat de klasse TransferManager niet

enkel over een ITransferRepository variabele moet beschikken, maar ook over ISpelerRepository en ITeamRepository variabele. Deze variabelen worden via de constructor meegegeven.

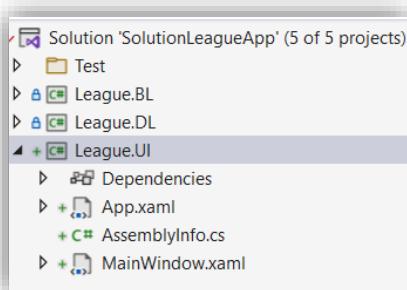
```
try
{
    //speler stopt
    if (nieuwTeamInfo == null)
    {
        if (spelerInfo.Team == null) throw new TransferManagerException("RegistreerTransfer - team is null");
        Speler speler=spelerRepo.SelecteerSpeler(spelerInfo.Id);
        transfer = new Transfer(speler, speler.Team);
        speler.VerwijderTeam();
    }
    //nieuwe speler
    else if (spelerInfo.Team == null)
    {
        Speler speler = spelerRepo.SelecteerSpeler(spelerInfo.Id);
        Team team=teamRepo.SelecteerTeam(nieuwTeamInfo.Stamnummer);
        speler.ZetTeam(team);
        transfer = new Transfer(speler, team, prijs);
    }
    //klassieke transfer
    else
    {
        Speler speler = spelerRepo.SelecteerSpeler(spelerInfo.Id);
        Team team = teamRepo.SelecteerTeam(nieuwTeamInfo.Stamnummer);
        transfer = new Transfer(speler, team, speler.Team, prijs);
        speler.ZetTeam(team);
    }
    return transferRepo.SchrijfTransferInDB(transfer);
}
```

```
public class TransferManager
{
    private ITransferRepository transferRepo;
    private ISpelerRepository spelerRepo;
    private ITeamRepository teamRepo;

    1 reference | 0 changes | 0 authors, 0 changes
    public TransferManager(ITransferRepository transferRepo, ISpelerRepository spelerRepo, ITeamRepository teamRepo)
    {
        this.transferRepo = transferRepo;
        this.spelerRepo = spelerRepo;
        this.teamRepo = teamRepo;
    }
}
```

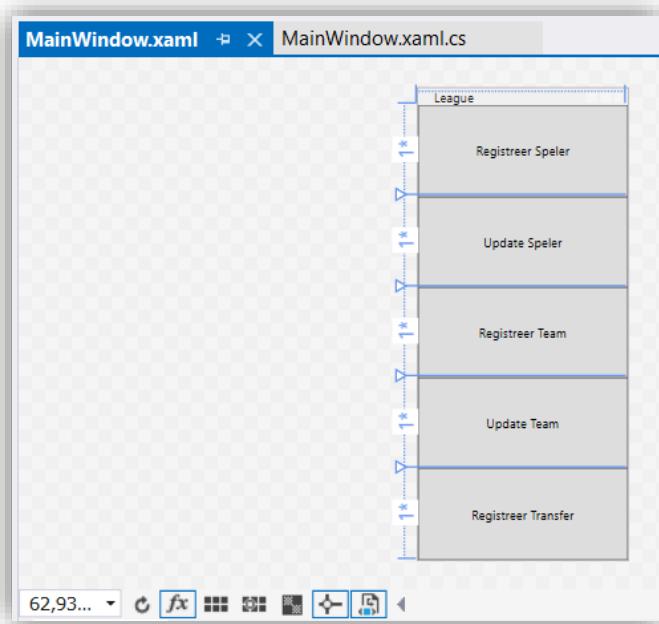
Applicatieontwikkeling – cyclus 3

In dit derde deel ontwerpen en implementeren we de UI, waarbij we gebruik maken van WPF (Windows Presentation Foundation). Dit vormt in onze architectuur een aparte laag en daarom maken we ook een apart project (WPF Application) aan binnen onze bestaande solution.



Hoofdvenster

Het hoofdvenster waarmee onze applicatie opstart is heel eenvoudig en bestaat enkel uit vijf knoppen die telkens overeenkomen met een user story.



Voor de opmaak maken we gebruik van een Grid, waarbij we 5 rijen definiëren met als hoogte “*”, dat wil zeggen dat de beschikbare ruimte volledig wordt gebruikt. Binnen elke rij plaatsen we een knop (Button) die we een duidelijke naam geven en waar we ook een click-event aan koppelen. Dit wil zeggen dat wanneer de knop wordt aangeklikt de bijhorende methode wordt uitgevoerd. Om aan te geven in welke rij de knop moet worden geplaatst, maken we gebruik van de Grid.Row property.

```
Title="League" Height="450" Width="200">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Button Grid.Row="0" Name="RegistreerSpelerButton" Content="Registreer Speler" Click="RegistreerSpelerButton_Click"/>
    <Button Grid.Row="1" Name="UpdateSpelerButton" Content="Update Speler" Click="UpdateSpelerButton_Click"/>
    <Button Grid.Row="2" Name="RegistreerTeamButton" Content="Registreer Team" Click="RegistreerTeamButton_Click"/>
    <Button Grid.Row="3" Name="UpdateTeamButton" Content="Update Team" Click="UpdateTeamButton_Click"/>
    <Button Grid.Row="4" Name="RegistreerTransferButton" Content="Registreer Transfer" Click="RegistreerTransferButton_Click"/>
</Grid>
```

De tekst die op de knop verschijnt geven we mee met de property Content. Verder geven we aan ons venster ook een titel mee en een hoogte en breedte. Het implementeren van de click-methodes stellen we nog even uit tot we effectief weten wat we precies gaan doen.

```

public partial class MainWindow : Window
{
    // ...
    public MainWindow()
    {
        InitializeComponent();
    }

    private void RegistreerSpelerButton_Click(object sender, RoutedEventArgs e)
    {
    }

    private void UpdateSpelerButton_Click(object sender, RoutedEventArgs e)
    {
    }

    private void RegistreerTeamButton_Click(object sender, RoutedEventArgs e)
    {
    }

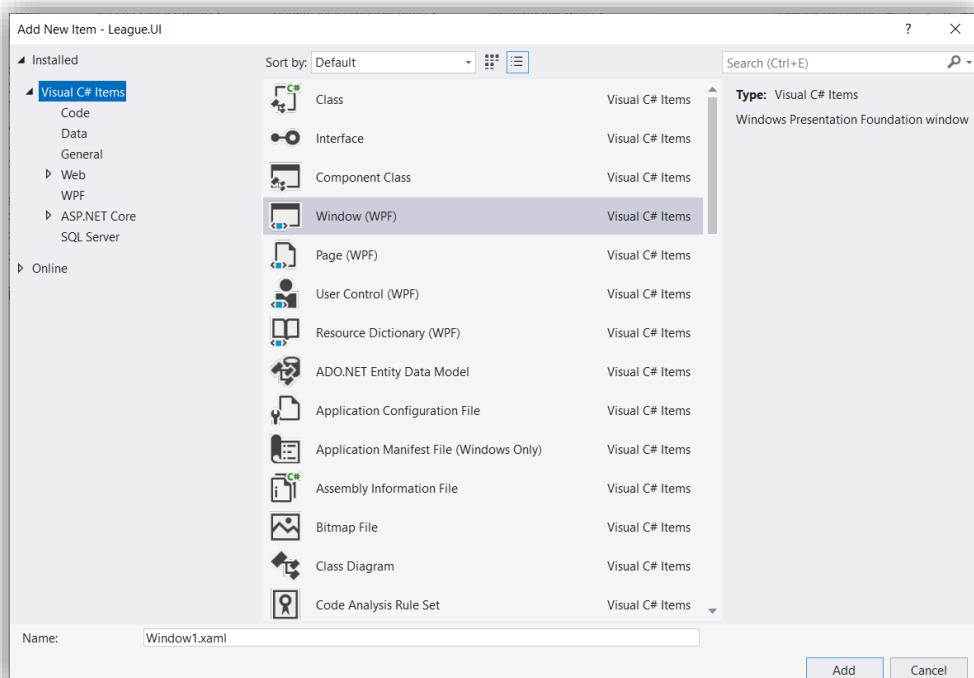
    private void UpdateTeamButton_Click(object sender, RoutedEventArgs e)
    {
    }

    private void RegistreerTransferButton_Click(object sender, RoutedEventArgs e)
    {
    }
}

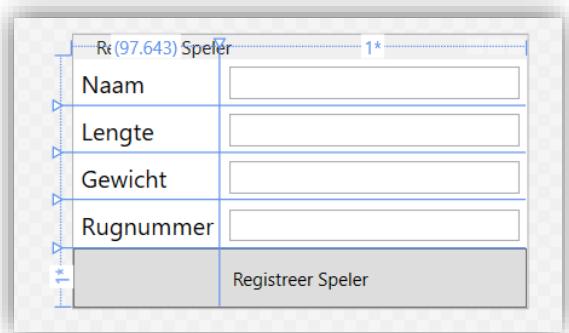
```

Registreer speler

De eerste user story die we gaan implementeren is het registreren van een nieuwe speler. Daarvoor maken we een eenvoudig venster met voor elke property een textbox waar de waarde kan worden ingegeven. Een venster toevoegen kan door een Window (WPF) item te selecteren wanneer we een item toevoegen. Het toevoegen van een venster (RegistreerSpelerWindow) zorgt er voor dat er telkens een xaml- en een bijhorende cs-file worden aangemaakt. In de xaml-file doen we de opmaak en in het cs-bestand schrijven we de code.



Verder voorzien we ook nog een knop om het registreren effectief uit te voeren.



Voor het ontwerp maken we weer gebruik van een grid en deze keer met vijf rijen en twee kolommen. De labels plaatsen we in de eerste kolom en de textboxen in de tweede. De 'Registreer Speler'-knop overspannt beide kolommen en dat kan je aangeven met de property ColumnSpan. Verder geven we voor elke component aan waar in het grid deze moet worden geplaatst doormiddel van de Grid.Row en Grid.Column properties. Aan de registreer-speler-knop koppelen we nog een event die de gepaste methode uit de business-laag zal oproepen.

```
Title="Registreer Speler" Height="180" Width="300">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="auto"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Label Grid.Row="0" Content="Naam" VerticalAlignment="Center" />
    <TextBox Grid.Row="0" Grid.Column="1" Name="NaamTextBox" Margin="5,5,5,5"/>
    <Label Grid.Row="1" Content="Lengte" VerticalAlignment="Center" />
    <TextBox Grid.Row="1" Grid.Column="1" Name="LengteTextBox" Margin="5,5,5,5"/>
    <Label Grid.Row="2" Content="Gewicht" VerticalAlignment="Center" />
    <TextBox Grid.Row="2" Grid.Column="1" Name="GewichtTextBox" Margin="5,5,5,5"/>
    <Label Grid.Row="3" Content="Rugnummer" VerticalAlignment="Center" />
    <TextBox Grid.Row="3" Grid.Column="1" Name="RugnummerTextBox" Margin="5,5,5,5"/>
    <Button Grid.Row="4" Name="RegistreerSpelerButton" Content="Registreer Speler"
           Click="RegistreerSpelerButton_Click" Grid.ColumnSpan="2"/>
</Grid>
```

Om nu ons Registreer-Speler-venster te tonen vullen we de RegistreerSpelerButton_Click-methode aan die zich in de code behind klasse bevindt van het main-window. Hierin maken we een variabele aan van de RegistreerSpelerWindow-klasse en roepen we het venster op met de ShowDialog-methode. Deze methode zorgt ervoor dat ons venster verschijnt en de focus krijgt tot het wordt gesloten.

```
private void RegistreerSpelerButton_Click(object sender, RoutedEventArgs e)
{
    RegistreerSpelerWindow w = new RegistreerSpelerWindow();
    w.ShowDialog();
}
```

Tot nu toe hebben we dus een MainWindow met daarop een registreerspeler-knop die wanneer aangeklikt het registreerspeler-venster toont. De volgende stap is nu de gegevens uit de textboxen halen en de business-laag aanspreken. Dit doen we pas op het moment dat de registreerspeler-knop wordt aangeklikt en de bijhorende methode (RegistreerSpelerButton_Click) wordt uitgevoerd. Om de tekst uit een textbox te lezen kunnen we gebruik maken van de property Tekst. De component zelf is een object die we door middel van zijn naam kunnen oproepen (meegegeven in de xaml-file). Voor elk van de componenten kijken we of er een waarde is ingevuld. Indien de naam niet is ingevuld laten we een MessageBox verschijnen die de gebruiker erop wijst dat de naam verplicht is. Voor de andere velden converteren we de string-waarde naar een getal. Eénmaal alle waarden zijn uitgelezen spreken we de business-laag aan om de user story effectief uit te voeren. Als dat is gelukt laten we dat de gebruiker weten via een MessageBox en sluiten we ons venster.

```
private void RegistreerSpelerButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        if (string.IsNullOrWhiteSpace(NaamTextBox.Text))
            MessageBox.Show("Naam mag niet leeg zijn");
        else
        {
            string naam = NaamTextBox.Text;
            int? rugnummer = null;
            if (!string.IsNullOrWhiteSpace(RugnummerTextBox.Text))
                rugnummer = int.Parse(RugnummerTextBox.Text);
            int? lengte = null;
            if (!string.IsNullOrWhiteSpace(LengteTextBox.Text))
                lengte = int.Parse(LengteTextBox.Text);
            int? gewicht = null;
            if (!string.IsNullOrWhiteSpace(GewichtTextBox.Text))
                gewicht = int.Parse(GewichtTextBox.Text);
            spelermanager.RegistreerSpeler(naam, lengte, gewicht);
            MessageBox.Show($"Speler : {naam}", "Speler is geregistreerd");
            Close();
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, ex.GetType().Name);
    }
}
```

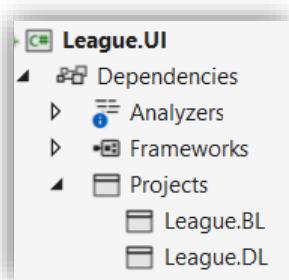
In de voorgaande methode hebben we een SpelerManager-object uit de businesslaag aangesproken. Er zijn verschillende manieren om dit object ter beschikking te stellen, we kunnen deze objecten bijvoorbeeld aanmaken in het MainWindow en ze dan meegeven aan de andere vensters of we kunnen dit object ook aanmaken in het venster waar we het nodig hebben. In deze oplossing kiezen we voor de laatste aanpak. Dit betekent dat we in de RegistreerSpelerWindow-klasse een variabele aanmaken van het type SpelerManager en dat we deze instantiëren in de constructor. De constructor van de SpelerManager verwacht van ons een object dat de ISpelerRepository-interface implementeert. En dit laatste object verwacht dan weer om de connectiestring voor de databank mee te krijgen.

```

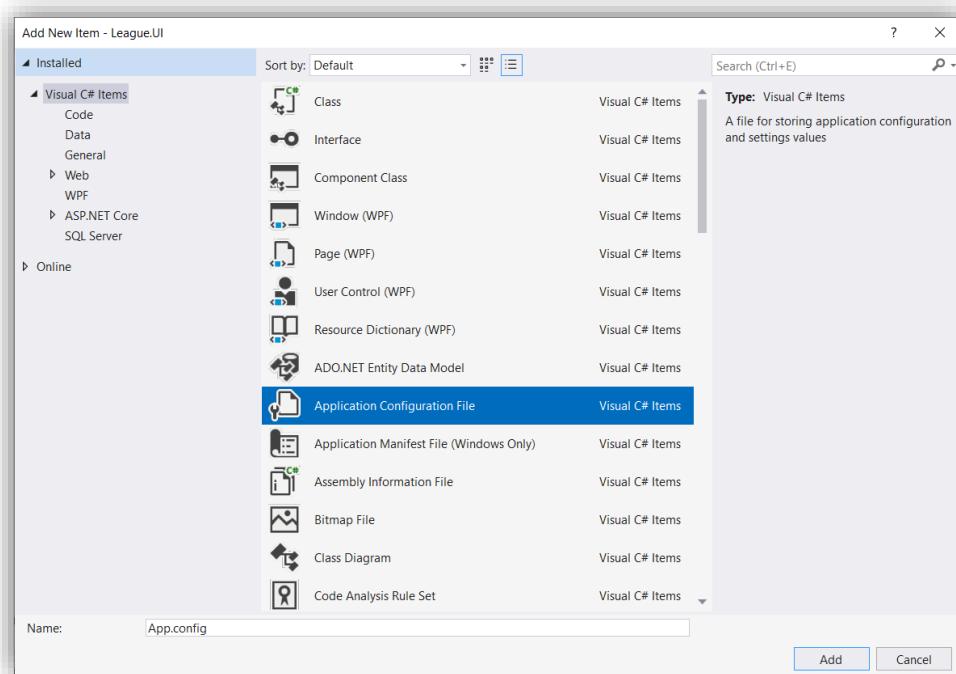
public partial class RegistreerSpelerWindow : Window
{
    private SpelerManager spelerManager;
    1 reference | 0 changes | 0 authors, 0 changes
    public RegistreerSpelerWindow()
    {
        InitializeComponent();
        spelerManager = new SpelerManager(
            new SpelerRepositoryADO(ConfigurationManager.ConnectionStrings["LeagueDBConnection"].ToString()));
    }
}

```

Om dit te realiseren moeten we dus eerst een aantal dependencies in orde brengen, en dat doen we door naar het business-laag-project en data-laag-project te verwijzen. Voor de connectiestring opteren we om deze in te lezen uit een configuratiebestand. Als de string op een correcte manier in het App.config bestand is opgeslagen, dan kunnen we deze via de ConfigurationManager-klasse inlezen.



Het App.config bestand kunnen we toevoegen aan het UI-project door een nieuw item toe te voegen en daarbij te kiezen voor een Application Configuration File.



In het configuratiebestand zelf voegen we dan een connectionStrings-tag toe met daarin dan een extra tag met de connectiestring zelf en de naam die we eraan verbinden. Het is deze naam die we gebruiken bij de ConfigurationManager.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <connectionStrings>
        <add name="LeagueDBConnection" connectionString="Data Source=NB21-6CDPYD3\SQLEXPRESS;" />
    </connectionStrings>
</configuration>
```

Update speler

De volgende user story die we nu gaan afwerken is het updaten van een speler. We starten opnieuw met het aanmaken van een venster (UpdateSpelerWindow). Ook hier maken we gebruik van een grid om de verschillende componenten een plaats te geven. We voorzien de mogelijkheid om een speler op te zoeken op basis van zijn id of naam. Als een speler is gevonden dan worden zijn eigenschappen getoond in de verschillende textboxen. Het team en de spelerid worden ook getoond, maar deze zijn niet aanpasbaar. Het aanpassen van het team moet via een transfer plaatsvinden.

The screenshot shows a Windows application window titled "Update Speler". Inside, there is a grid with four columns. The first column has a header "U1* ate 5 Spele 1*". The second column has headers "Naam" and "Team". The third column has headers "Lengte" and "Rugnummer". The fourth column has a header "Zoek Speler". Below the grid is a button labeled "Update Speler".

```
Title="Update Speler" Height="180" Width="500">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="2*"/>
        <ColumnDefinition Width="2*"/>
        <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>
```

```

<Label Grid.Row="0" Grid.Column="0" Content="SpelerID" />
<TextBox Grid.Row="0" Grid.Column="1" Name="ZoekSpelerIDTextBox" Margin="5,5,5,5"/>
<Label Grid.Row="0" Grid.Column="2" Content="Naam" />
<TextBox Grid.Row="0" Grid.Column="3" Name="ZoekNaamTextBox" Margin="5,5,5,5"/>
<Button Grid.Row="0" Grid.Column="4" Name="ZoekSpelerButton" Content="Zoek Speler" Margin="5,5,5,5"
        Click="ZoekSpelerButton_Click"/>
<Label Grid.Row="1" Grid.Column="0" Content="Speler" />
<Label Grid.Row="1" Grid.Column="1" Content="SpelerID" />
<TextBox Grid.Row="1" Grid.Column="2" Name="SpelerIDTextBox" Margin="5,5,5,5" IsReadOnly="True"/>
<Label Grid.Row="1" Grid.Column="3" Content="Team" />
<TextBox Grid.Row="1" Grid.Column="4" Name="TeamTextBox" Margin="5,5,5,5" IsReadOnly="True"/>
<Label Grid.Row="2" Grid.Column="1" Content="Naam" />
<TextBox Grid.Row="2" Grid.Column="2" Name="NaamTextBox" Margin="5,5,5,5" />
<Label Grid.Row="2" Grid.Column="3" Content="Lengte" />
<TextBox Grid.Row="2" Grid.Column="4" Name="LengteTextBox" Margin="5,5,5,5" />
<Label Grid.Row="3" Grid.Column="1" Content="Gewicht" />
<TextBox Grid.Row="3" Grid.Column="2" Name="GewichtTextBox" Margin="5,5,5,5" />
<Label Grid.Row="3" Grid.Column="3" Content="Rugnummer" />
<TextBox Grid.Row="3" Grid.Column="4" Name="RugnummerTextBox" Margin="5,5,5,5" />
<Button Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="5" Content="Update Speler" Click="Button_Click"/>

```

Zowel aan de zoekspeler-knop als de updatespeler-knop worden events gekoppeld. In ons MainWindow vullen we nu de UpdateSpelerButton_Click-methode verder aan, we maken een UpdateSpelerWindow-object aan en roepen de ShowDialog-methode ervan op.

```

private void UpdateSpelerButton_Click(object sender, RoutedEventArgs e)
{
    UpdateSpelerWindow w = new UpdateSpelerWindow();
    w.ShowDialog();
}

```

In de UpdateSpelerWindow-klasse gaan we gebruik maken van een SpelerManager-object uit de business-laag en dus voorzien we een variabele die we in de constructor initialiseren. Dit is net hetzelfde als bij de RegistreerSpelerWindow-klasse.

```

public partial class UpdateSpelerWindow : Window
{
    private SpelerManager spelerManager;
    1 reference | Tom Vande Wiele, 3 days ago | 1 author, 1 change
    public UpdateSpelerWindow()
    {
        InitializeComponent();
        spelerManager = new SpelerManager(
            new SpelerRepositoryADO(ConfigurationManager.ConnectionStrings["LeagueDBConnection"].ToString()));
    }
}

```

De volgende stap is het zoeken van een speler, zoals reeds gezegd dat kan zowel op basis van zijn id als de naam. Als we zoeken op id dan kan er maximaal één speler worden gevonden en vullen we die ook direct in, zoeken we op naam dan kunnen er meerder spelers gevonden worden en gaan we de gebruiker een keuze laten maken. We zullen dan een nieuw venster openen met daarin de verschillende spelers die zijn gevonden.

Het eerste wat we doen is de textboxen (ZoekNaamTextBox en ZoekSpelerIDTextBox) uitlezen, daarna roepen we de SelecteerSpelers-methode op van de SpelerManager die ons een lijst van spelers teruggeeft. Er zijn nu drie mogelijke resultaten, ofwel is er geen enkele speler gevonden en dan maken we de verschillende textboxen leeg, ofwel is er exact één speler gevonden en dan vullen

we de eigenschappen in of er zijn meerdere spelers en dan openen we een nieuw venster waar de gebruiker dan moet kiezen.

```
private void ZoekSpelerButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        int? spelervId = null;
        string naam = null;
        if (!string.IsNullOrWhiteSpace(ZoekNaamTextBox.Text)) naam = ZoekNaamTextBox.Text;
        if (!string.IsNullOrWhiteSpace(ZoekSpelerIDTextBox.Text))
        {
            spelervId = int.Parse(ZoekSpelerIDTextBox.Text);
        }
        List<SpelerInfo> spelers = (List<SpelerInfo>)spelerManager.SelecteerSpelers(spelervId, naam);
        if (spelers.Count == 0)...
        if (spelers.Count == 1)...
        if (spelers.Count > 1)...
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

```
if (spelers.Count == 0)
{
    NaamTextBox.Text = "";
    SpelerIDTextBox.Text = "";
    GewichtTextBox.Text = "";
    LengteTextBox.Text = "";
    RugnummerTextBox.Text = "";
    TeamTextBox.Text = "";
}
if (spelers.Count == 1)
{
    NaamTextBox.Text = spelers[0].Naam;
    SpelerIDTextBox.Text = spelers[0].Id.ToString();
    GewichtTextBox.Text = spelers[0].Gewicht.ToString();
    LengteTextBox.Text = spelers[0].Lengte.ToString();
    RugnummerTextBox.Text = spelers[0].Rugnummer.ToString();
    TeamTextBox.Text = spelers[0].Team.ToString();
}
```

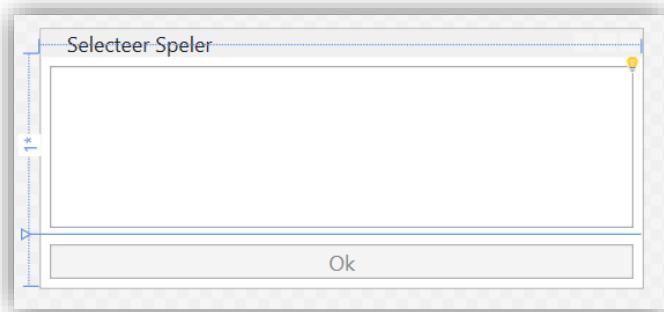
Als er maar één speler is gevonden kunnen we de eerste speler uit de lijst nemen en de waarden invullen. Zijn er meerder spelers dan openen we een nieuw venster (SelecteerSpelerWindow) – dat we nog moeten aanmaken. Aan dit nieuw venster (klasse) geven we dan de lijst van gevonden spelers mee (via de constructor) zodat deze kunnen worden getoond. In de SelecteerSpelerWindow-klasse voorzien we ook een (public) property (SelectedSpeler) die de geselecteerde speler bevat zodat we het resultaat van de selectie kennen. Met het `if (selecteerSpeler.ShowDialog() == true)` statement controleren we of de gebruiker wel een selectie heeft gemaakt, want je kan ook altijd gewoon het venster sluiten natuurlijk. Als een speler is geselecteerd vullen we weer de verschillende textboxen in.

```

if (spelers.Count > 1)
{
    SelecteerSpelerWindow selecteerSpeler=new SelecteerSpelerWindow(spelers);
    if (selecteerSpeler.ShowDialog() == true)
    {
        NaamTextBox.Text = selecteerSpeler.SelectedSpeler.Naam;
        SpelerIDTextBox.Text = selecteerSpeler.SelectedSpeler.Id.ToString();
        GewichtTextBox.Text = selecteerSpeler.SelectedSpeler.Gewicht.ToString();
        LengteTextBox.Text = selecteerSpeler.SelectedSpeler.Lengte.ToString();
        RugnummerTextBox.Text = selecteerSpeler.SelectedSpeler.Rugnummer.ToString();
        TeamTextBox.Text = selecteerSpeler.SelectedSpeler.Team.ToString();
    }
}

```

Het SelecteerSpelerWindow is een heel eenvoudig venster, het bevat enkel een ListBox (met de spelers) en een Ok-knop om de keuze te bevestigen.



```

    Title="Selecteer Speler" Height="150" Width="350">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="auto"/>
    </Grid.RowDefinitions>
    <ListBox Grid.Row="0" Name="SpelersListBox" Margin="5,5,5,5"
             SelectionChanged="SpelersListBox_SelectionChanged"/>
    <Button Grid.Row="1" Name="OkButton" Content="Ok" Margin="5,5,5,5" IsEnabled="False"
            Click="OKButton_Click"/>
</Grid>

```

We voorzien hier twee events, één als we op de ok-knop drukken en één als we onze keuze (selectie) veranderen in de ListBox. De klasse SelecteerSpelerWindow bevat twee variabelen, één voor de lijst van spelers en één voor de geselecteerde speler (deze laat ons toe om gegevens tussen verschillende vensters uit te wisselen). De lijst van spelers toekennen (invullen) in de SpelersListBox kan door de List<SpelerInfo> toe te kennen aan de property ItemSource. Wat we wel nog moeten doen is de ToString-methode van SpelerInfo overschrijven.

```

public override string ToString()
{
    return $"{Id},{Naam},{Team},{Rugnummer},{Lengte},{Gewicht}";
}

```

De event-methodes zelf zijn vrij eenvoudig. Bij het SelectionChanged-event van de ListBox enabelen (aanzetten) we de ok-knop en kennen we het geselecteerde item toe aan de property SelectedSpeler. Dat kan eenvoudigweg door de SelectedItem-property van de SpelersListBox te gebruiken. Aangezien de items van SpelersListBox reeds SpelerInfo-objecten bevat zal ook het SelectedItem een SpelerInfo-object bevatten.

```
public partial class SelecteerSpelerWindow : Window
{
    private List<SpelerInfo> spelers;
    public SpelerInfo? SelectedSpeler=null;
    2 references | 0 changes | 0 authors, 0 changes
    public SelecteerSpelerWindow(List<SpelerInfo> spelers)
    {
        InitializeComponent();
        this.spelers = spelers;
        SpelersListBox.ItemsSource = spelers;
    }
    1 reference | 0 changes | 0 authors, 0 changes
    private void SpelersListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
    {
        OkButton.IsEnabled = true;
        SelectedSpeler=SpelersListBox.SelectedItem as SpelerInfo;
    }
    1 reference | 0 changes | 0 authors, 0 changes
    private void OkButton_Click(object sender, RoutedEventArgs e)
    {
        DialogResult = true;
        Close();
    }
}
```

Bij het aanklikken van de OkButton geven we aan dat alles volgens plan verloopt en dat er dus een speler is gekozen. We stellen de DialogResult-property van het venster in op true (dit wordt in het UpdateSpeler venster gecontroleerd) en sluiten het venster. Het sluiten van het venster wil niet zeggen dat het object direct wordt verwijderd, zolang er een referentie naar bestaat blijft het beschikbaar.

Nadat de speler is opgehaald en de waarden getoond, kan de gebruiker deze wijzigen en uiteindelijk de Update-knop aanklikken. Op basis van de invoertextboxen wordt er dan een SpelerInfo-object aangemaakt dat we aan de UpdateSpeler-methode meegeven van het spelerManager-object. Als de update is gelukt informeren we de gebruiker met een MessageBox en sluiten we het UpdateSpeler-venster.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    try
    {
        int spelerid = int.Parse(SpelerIDTextBox.Text);
        int? gewicht = null;
        if (!string.IsNullOrWhiteSpace(GewichtTextBox.Text))
            gewicht = int.Parse(GewichtTextBox.Text);
        int? lengte = null;
        if (!string.IsNullOrWhiteSpace(LengteTextBox.Text))
            lengte = int.Parse(LengteTextBox.Text);
        int? rugnummer = null;
        if (!string.IsNullOrWhiteSpace(RugnummerTextBox.Text))
            rugnummer = int.Parse(RugnummerTextBox.Text);
        SpelerInfo spelerInfo =
            new SpelerInfo(spelerid, NaamTextBox.Text, TeamTextBox.Text, rugnummer, lengte, gewicht);
        spelerManager.UpdateSpeler(spelerInfo);
        MessageBox.Show($"Speler : {spelerInfo}", "Speler is up to date");
        Close();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```