SharpShuttle Documentatie

Microsoft® Visual C#®

Microsoft® .NET

Inhoudsopgave

1 Introductie

Dit document dient de interne structuur van de gedistribueerde Sharp Shuttle applicatie te verduidelijken. Het doel is om een gekwalificeerde ontwikkelaar in staat te stellen de applicatie te begrijpen en adequaat wijzigingen aan te brengen ofwel uitbreidingen toe te voegen.

Er is gekozen voor een top-down benadering van de applicatie, beginnende bij een abstracte high-level omschrijving, waarna per onderdeel steeds meer in detail wordt gegaan, tot – waar nodig – aan (pseudo-) code-voorbeelden toe.

De documentatie is vanuit de bestaande applicatie ontstaan, maar poogt ook waar nodig enkele gemaakte keuzes in het ontwerp van het eindresultaat te verklaren en zo meer inzicht te verschaffen.

2 Algemene Opzet

2.1 Probleem

S.B. Helios te Utrecht had een systeem/computerprogramma nodig dat de uitvoering van een internationaal studenten badminton toernooi (ISBT) ondersteunt. Dit zijn grote toernooien (met met meer dan 100 deelnemers) waarbij het Zwitserse laddersysteem centraal staat. De belangrijkste taak van de te maken software was het correct beheren en berekenen van de ladder(s), terwijl er voldoende mate aan flexibiliteit voor onvoorziene situaties tijdens een lopend toernooi behouden moest zijn. Daarbuiten was het een vereiste om een gedistribueerde applicatie op te leveren, omdat meerdere personen (aan verschillende computers) tegelijk met het systeem zouden moeten kunnen werken.

2.1.1 Randvoorwaarden

- Het uiteindelijke product moet functioneren op computers die het Microsoft Windows besturingssysteem draaien.
- Niet elke gebruiker mag de volledige functionaliteit gebruiken. Dit om te voorkomen dat onervaren vrijwilligers, die helpen met kleine taken (zoals wedstrijdresultaten invoeren), per ongeluk belangrijke gegevens kunnen overschrijven.

2.2 Oplossing

Er is gekozen voor een gedistribueerde native Microsoft Windows applicatie, bestaande uit twee delen: Een server-applicatie en een client-applicatie.

De server-applicatie fungeert als een dataopslag- en communicatiecentrale. Het is haar taak om alle instanties van de client-applicatie op de hoogte te houden van gewijzigde data, en het meervoudig concurrent opslaan van data te coördineren.

De client-applicatie is de interface naar de gebruiker, en neemt hem/haar het administrative werk van de Zwitserse toernooi-ladders uit handen. In dit onderdeel zit tevens alle intelligentie en kennis verwerkt, die achter de schermen ervoor zorgt dat alle functionaliteit correct volgens verwachtingen en specificaties functioneert. De mate waarin functionaliteit beschikbaar is wordt bepaald door de gebruikersidentiteit waarmee een fysieke gebruiker toegang tot de gegevens van het toernooi krijgt.

2.2.1 Aannames

- De computers waarop Sharp Shuttle moet functioneren draaien het Microsoft Windows besturingssysteem.
- Indien er meer dan één computer tegelijk gebruikt wordt voor de administratie van een toernooi d.m.v. Sharp Shuttle, is er een Local Area Network (LAN) aanwezig waarover de computers kunnen communiceren.
- Er is geen sprake van digitale criminaliteit zoals kwaadaardige Sharp Shuttle-gebruikers met beheerdersrechten of invloeden die opzettelijk data corrupteren.
- De gebruikers van Sharp Shuttle beschikken over basis-computerkennis, en beschikken over een toetsenbord én een computermuis om de applicatie te besturen.
- Het Sharp Shuttle programma draait onder een gebruiker die volledige toegang tot het systeem heeft.

3 Functioneel Ontwerp

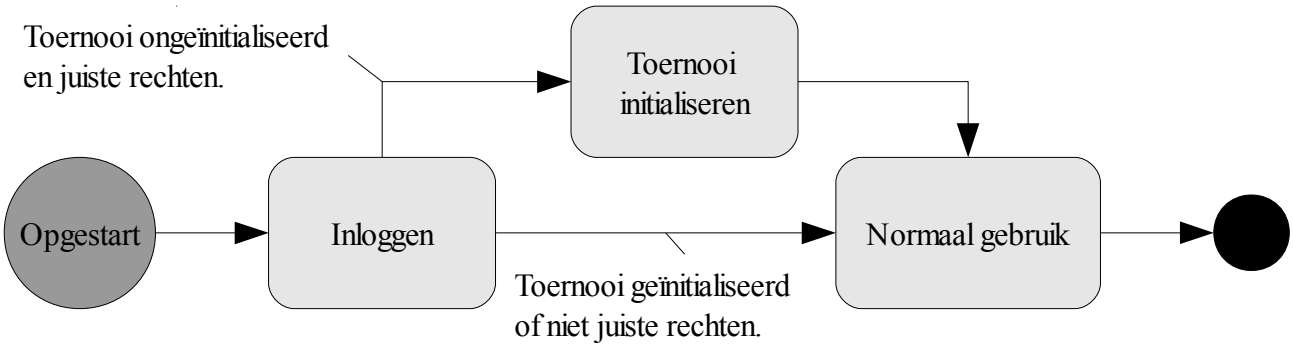
Net als hoofdstuk 2 gaat ook dit hoofdstuk nog niet uitgebreid in op specifieke implementatiedetails of kleine functionele details. Dit hoofdstuk beschrijft het functioneel ontwerp van het programma door de geleverde functionaliteiten en de volgordes waarin die worden uitgevoerd te bespreken.

3.1 Server-applicatie

De server vormt het hart van het systeem. De server beheert de toegang tot de database en vormt ook een communicatiecentrum tussen alle clients. Op de server wordt een toernooi gehost dat beschikbaar is voor de clients. Welk toernooi dat is wordt bepaald door de database XML file. In de server is functionaliteit om een bestaand toernooi te openen of een nieuw toernooi te starten. Zonder deze server zijn de clients onbruikbaar. Meer informatie over de server is te vinden in hoofdstuk 6

3.2 Client-applicatie

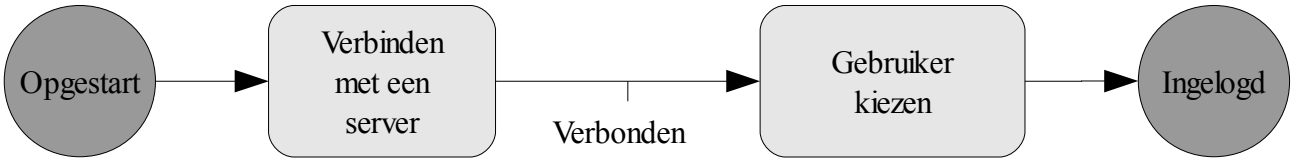
De gebruiker komt het meest in contact met de client-applicatie. De client bied de mogelijkheid om informatie van de server weer te geven en te wijzigen. Het gebruik van de client is te verdelen in drie verschillende stadia. In hoofdstuk 3.2.1 wordt het inloggen van de client besproken. Het opstarten van een nieuw toernooi is beschreven in hoofdstuk 3.2.2 en het gebruik van de belangrijkste toernooi functionaliteiten is te vinden in hoofdstuk 3.2.3. Deze drie delen worden in het programma ook in die volgorde doorgelopen, hoewel het opstarten van een nieuw toernooi niet altijd voorkomt.



Flowchart algemeen

3.2.1 Inloggen

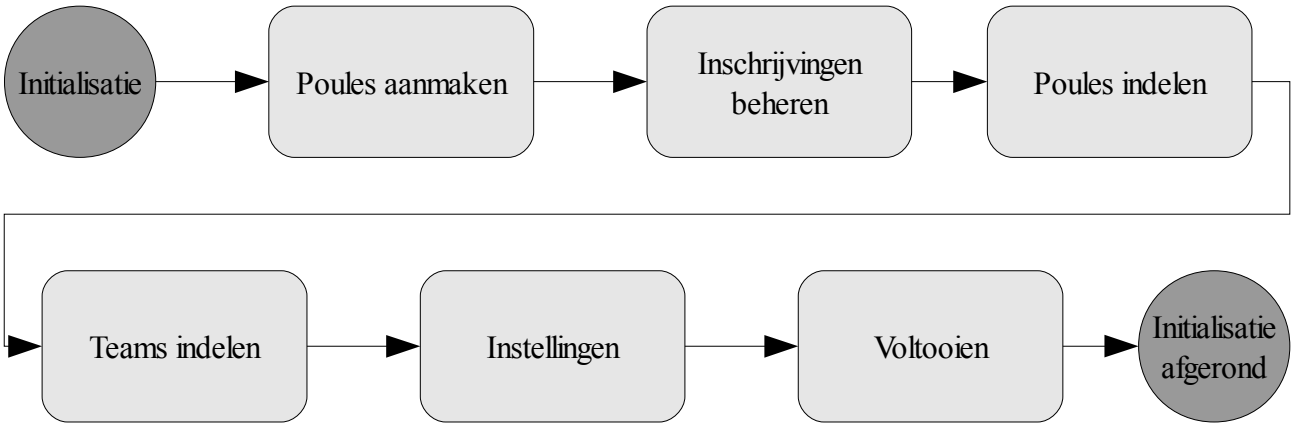
De client werkt met data die bewaard wordt op een server. Daarom is het nodig na het opstarten van de client in te loggen op de server. Dit inloggen bestaat uit twee stappen. Ten eerste moet er verbonden worden met de server. Hiervoor kan een IP adres of hostname worden ingevoerd. Ten tweede moet er ingelogd worden als een bepaald soort gebruiker. Afhankelijk van het type gebruiker zijn functionaliteiten wel of niet beschikbaar. Inloggen kan later vanuit het menu opnieuw gestart worden om (opnieuw) met een server te verbinden.



Flowchart login

3.2.2 Toernooi initialiseren

Voordat een toernooi kan worden gespeeld, moet deze eerst worden geïnitialiseerd. Dit kan maar één keer per toernooi worden gedaan en omdat het zo'n belangrijk onderdeel is heeft de alleen de beheerder rechten om dit te doen. In de initialisatie worden poules, spelers en teams aangemaakt, alsmede de instellingen van het toernooi goed gezet. De verschillende stappen in de initialisatie moeten in de gegeven volgorde worden doorlopen, maar zolang de initialisatie nog niet voltooid is kunnen vorige stappen opnieuw worden gedaan. Na het voltooien van de initialisatie zijn er nog maar enkele wijzigingen in het toernooi mogelijk.



Flowchart initialisatie

3.2.2.1 Poules aanmaken

Alle wedstrijden in Sharp Shuttle worden gespeeld in poules. Een poule bestaat uit een niveau en een discipline. Niveaus zijn door de gebruiker aan te maken. De disciplines staan vast en zijn enkels en dubbels voor zowel vrouwen, mannen als mannen en vrouwen door elkaar. Hiernaast is er nog de mixed dubbel, waar elk team bestaat uit een man en een vrouw. Het is mogelijk meerdere poules van dezelfde niveau en discipline te maken, zolang de poulenaam maar uniek is.

3.2.2.2 Inschrijvingen beheren

In deze stap worden alle spelers van het toernooi getoond. Spelers kunnen worden uitgelezen vanuit een XML bestand. Hiernaast kunnen spelers met de hand worden ingevoerd. Elke speler heeft in ieder geval een naam en een geslacht, maar optioneel zijn ook de vereniging, poulevoorkeuren en opmerkingen toe te voegen.

3.2.2.3 Poules indelen

De ingeschreven spelers moeten aan de verschillende poules worden toegevoegd. Spelers met voorkeuren kunnen automatisch worden ingedeeld. Lege poules worden uiteindelijk verwijdert uit het toernooi. Voordat de volgende stap kan beginnen zijn er een aantal eisen aan de indelingen. Zo moet het aantal spelers in een dubbel poule even zijn, en moet het aantal vrouwen en mannen in een mixed poule gelijk zijn. Zonder deze eisen is het onmogelijk om teams te maken.

3.2.2.4 Teams indelen

In dubbelpoules moeten de spelers per twee aan elkaar gekoppeld worden om een team te vormen. De eerste indeling wordt door een algoritme gemaakt, waarbij gepoogd word spelers van verschillende verenigingen aan elkaar te koppelen. De gebruiker krijgt de mogelijkheid deze indeling te wijzigen.

3.2.2.5 Instellingen

Er zijn een aantal instellingen die voor het hele toernooi gelden. Dit zijn het aantal sets per wedstrijd, variërend van 1 tot en met 3, en het aantal banen dat beschikbaar is op het toernooi. Het aantal sets per wedstrijd is tijdens het toernooi niet meer te wijzigen.

3.2.2.6 Voltooien

Bij het voltooien van de initialisatie wordt het toernooi gestart. De functionaliteiten voor het gebruik van het toernooi zijn daarna beschikbaar. Voor elke poule worden alvast de eerste wedstrijden aangemaakt. Het voltooien van de initialisatie is onomkeerbaar.

3.2.3 Toernooi functionaliteiten

Als er eenmaal is ingelogd op een server waar een geïnitialiseerd toernooi op staat zijn de belangrijkste functionaliteiten pas beschikbaar. Deze zijn ruwweg onder te verdelen in drie schermen. Hiernaast zijn er nog een aantal functionaliteiten bereikbaar vanuit het menu. De indeling en de relevantie daarvan spreken voor zich.

3.2.3.1 Poule informatie

Poule informatie is het belangrijkste venster voor alles wat met poules te maken heeft. De verschillende poules zijn hier in detail te bekijken. Zo zijn teams in een poule met de bijbehorende ranglijst te zien en ook de wedstrijden van een poule zijn hier te vinden. Vanuit dit scherm kunnen nieuwe rondes worden gestart en met de juiste rechten scores worden ingevuld. Wedstrijdbriefjes kunnen opnieuw worden uitgeprint en een nieuw gegeneerde ronde kan nog worden teruggedraaid. Ook kunnen teams hier gewijzigd worden, door bijvoorbeeld een speler te wisselen of het team uit te laten vallen.

3.2.3.2 Veld informatie

Het veld informatie scherm beheert eigenlijk het toernooi zelf. Alle ongespeelde wedstrijden zijn hier te zien, alsmede een overzicht van de verschillende banen. Wedstrijden kunnen op een baan gezet en gestart worden. Er is te zien hoelang een wedstrijd al bezig is en er word gewaarschuwd voor mogelijke conflicten in de wedstrijdindeling, zoals een team indelen waarvan een speler al op het veld staat.

3.2.3.3 Wedstrijdscores

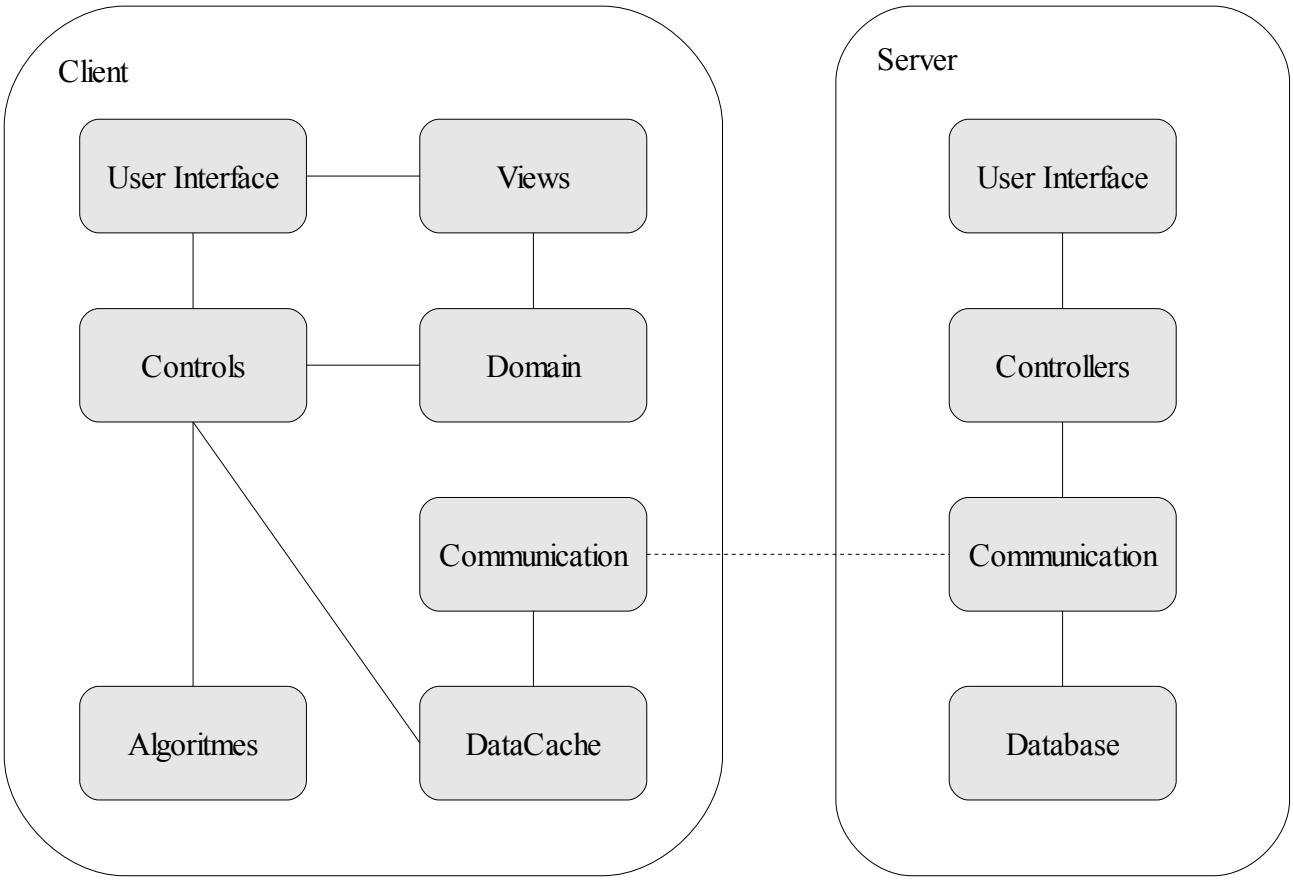
In het wedstrijdscores venster kunnen van alle gestarte wedstrijden de scores ingevoerd worden. Ook is het hier mogelijk om scores van de huidige ronde weer te wijzigen. Dit scherm is vooral bedoeld om de beheerder te ontlasten door een ander persoon de scores in te laten vullen.

3.2.3.4 Menu

Vanuit het menu is het mogelijk de inlog- en initialisatiewizard opnieuw op te starten, bij een nieuwe verbinding of na het per ongeluk sluiten van een scherm. Daarnaast zijn de printer en het aantal banen in te stellen. Ook is het mogelijk om wedstrijdbriefjes uit te printen.

4 Technisch Ontwerp

Sharp Shuttle bestaat uit een Client en een Server, die via een netwerk met elkaar communiceren. De Client en de Server zelf bestaan ook weer uit verschillende onderdelen die met elkaar communiceren.



Structuurdiagram

In deze en de volgende hoofdstukken worden de verschillende onderdelen van het systeem besproken. Hierbij komen de technische aspecten, de structuur van de code en de achterliggende gedachten van de structuur aan bod.

4.1 Programmeertaal, Framework en Ontwikkelingsomgeving

Sharp Shuttle is geschreven in de taal C# en er is gebruik gemaakt van het Microsoft .NET 3.5 framework. De gebruikte ontwikkelingsomgeving is Microsoft Visual Studio 2008 SP1. In deze omgeving wordt Sharp Shuttle een Windows Forms Application genoemd.

Om verder te werken aan Sharp Shuttle kan het VisualStudio 2008 Solution-bestand “Sharp Shuttle.sln” met deze omgeving geopend worden. Deze solution bestaat uit vier projecten – die samen de gehele Sharp Shuttle applicatie vormen – die in de hoofdstukken 5, 6, 7 en 8 nader toegelicht zullen worden. Alle implementatiedetails die niet in die hoofdstukken aan bod komen worden in hoofdstuk 9 besproken.

4.2 Netwerkcommunicatie en Synchronisatie

Sharp Shuttle werkt met een enkele server waarop meerdere clients kunnen connecten. Tussen de clients onderling is geen communicatie, alle berichten gaan via de server. De server beheert de data die alle clients gebruiken in de vorm van een database. Clients kunnen deze data opvragen en verzoeken tot wijzigingen indienen. Om het netwerkverkeer tot een minimum te beperken wordt de data ook op elke client bewaard. Om te weten of de data op een client wel de laatste versie is, heeft

elke dataset een versienummer. Meer over het netwerk is te vinden in Netwerkcommunicatie en informatie over de toegang tot de data staat in Datatoegang.

4.3 Algoritmes

Sharp Shuttle is afhankelijk van drie verschillende algoritmes. Twee hiervan zijn voor het maken van teams en de derde is voor het genereren van nieuwe wedstrijden. De algoritmes van het programma zijn te vinden in de folder Algorithms, in het Shared project. De aanroepen van de verschillende algoritmes zijn gebundeld in de klasse Algorithms.cs. De methoden in deze klasse handelen de verificatie en sortering van de ingegeven data af, houden de administratie van de algoritmes bij en roept de algoritmes zelf aan. De algoritmes zelf staan in de klassen Ladder.cs, Matches.cs en MixedMatcher.cs. Alle algoritmes werken op Domain objecten, maken gebruik van de datastructuren uit Datastructures en leveren een null-waarde op als de invoer incorrect is.

4.3.1 Teamindeling

De algoritmes voor teamindeling verwachten een lijst van spelers en leveren een lijst van teams op. Het aantal spelers moet hierbij even zijn. De algoritmes proberen dan teams samen te stellen, waarbij geprobeerd wordt spelers van verschillende verenigingen bij elkaar te zetten. Dit lukt als niet meer dan de helft van de spelers van dezelfde vereniging is. Het maakt voor de algoritmes niet uit of een speler een vereniging heeft, alle spelers zonder vereniging worden dan gezien als van dezelfde vereniging. Het is aan te raden om wel een vereniging op te geven, aangezien het algoritme zonder deze informatie weinig meer doet dan twee spelers bij elkaar voegen in een team. Aangezien de algoritmes de invoer in een bepaalde volgorde verwachten, wordt de invoer in Algorithms.cs gesorteerd.

4.3.1.1 Standaard teamindeling

Het standaard teamindelingsalgoritme, in Matcher.cs, koppelt spelers aan elkaar onafhankelijk van geslacht. De invoer van het algoritme moet gesorteerd zijn, met de grootste vereniging vooraan in de lijst van spelers. Vervolgens wordt er in elke ronde een speler van de grootste vereniging gekoppeld aan een andere vereniging. De andere vereniging wordt gekozen door een RoundRobin klasse. Als er geen andere vereniging meer over is, wordt er aan dezelfde vereniging gekoppeld. Uiteindelijk zijn alle spelers aan elkaar gekoppeld met zo min mogelijk spelers van dezelfde vereniging bij elkaar.

4.3.1.2 Mixed teamindeling

De teamindeling voor een mixed poule, in MixedMatcher.cs, heeft als extra eis dat het aantal vrouwelijke spelers gelijk is aan het aantal mannelijke spelers. Het algoritme verwacht een gesorteerde lijst, met de mannen voor de vrouwen, en daarna de grootste vereniging voorop. De spelers worden per geslacht en per vereniging. Vervolgens wordt in elke ronde een speler van de grootste verenging gekoppeld aan de grootste verschillende vereniging bij het andere geslacht. Het algoritme kan zichzelf vastzetten zodat in de laatste ronde alleen een koppeling met zichzelf mogelijk is, terwijl een betere oplossing mogelijk is. Daarom wordt, als in het laatste team allebei de spelers van dezelfde vereniging zijn, een van de spelers omgewisseld met het team ervoor. Aan het eind van het algoritme zijn de spelers zo goed mogelijk aan elkaar gekoppeld. In de gegenereerde teams staat de man altijd als Speler 1 en de vrouw altijd als Speler 2.

4.3.2 Ladder

Het ladder algoritme genereert wedstrijden voor een enkele poule. Deze wedstrijden moeten aan een aantal eisen voldoen. Zo moet elke wedstrijd uniek zijn en mag elke ronde ten hoogste 1 team niet spelen. Verder moet volgens het Zwitserse laddersysteem een team gekoppeld worden aan een team dat qua positie op de ranglijst in de buurt staat. Dit is bovenaan de ranglijst belangrijker dan onderaan, omdat de teams bovenaan waarschijnlijk voor prijzen spelen.

Bij het ontwerpen van het algoritme zijn een aantal verschillende methodes geprobeerd. De meeste hiervan vielen af omdat ze niet gegarandeerd met een oplossing kwamen, of geen rekening konden houden met de ranglijst. Een ander probleem was dat als een mogelijke oplossing gevonden was, niet bekend is of er verder gezocht moet worden naar een betere mogelijkheid, omdat er daarvoor een manier nodig is om de goedheid van een oplossing te bepalen. Uiteindelijk is gekozen voor een algoritme dat alle mogelijke oplossingen probeert, dat een goede oplossing oplevert in ruil voor een langere looptijd.

4.3.2.1 Administratie

De invoer van het algoritme is een gesorteerde lijst van teams, met het hoogst gerankte team vooraan, een lijst van gespeelde wedstrijden en het rondenummer waarvoor de wedstrijden gegenereerd worden. Het algoritme zelf, te vinden in Ladder.cs, werkt op een graaf van mogelijke wedstrijden. Deze graaf wordt in de administratie van het algoritme, in Algorithms.cs, gemaakt en goed gezet. Aangezien het algoritme alleen op een even aantal teams werkt, moet hier ook voor worden gezorgd. Als het aantal teams oneven is, wordt een apart ByeTeam toegevoegd. Een team dat in een wedstrijd hiertegen speelt, speelt eigenlijk niet. Dit ByeTeam en de bijbehorende wedstrijd worden naderhand ook uit de resultaten verwijderd. Hierna wordt de graaf aangemaakt uit de lijst van gespeelde wedstrijden, om dubbele wedstrijden te voorkomen. De mogelijkheden van het ByeTeam worden bepaald aan de hand van het aantal gespeelde wedstrijden van elk team.

4.3.2.2 Werking

Het algoritme zelf is een recursief Branch en Bound algoritme. Alle mogelijke indelingen worden stap voor stap geprobeerd en geëvalueerd. Aan elke wedstrijd wordt een score toegekend, afhankelijk van de afstand tussen de twee teams en de hoogte op de ranking. De som van deze scores is de penalty van een indeling. Als tijdens het branchen de penalty stijgt boven de penalty van de beste indeling wordt de branch afgebroken. Als een complete indeling is gemaakt, wordt gekeken of deze voldoet aan een aantal eisen aan de gewijzigde graaf. Zo mag de graaf geen oneven subgroepen hebben of hiertoe leiden in volgende rondes. Als de gevonden indeling beter is dan de beste gevonden indeling dan wordt deze bewaard. Uiteindelijk wordt de beste indeling teruggegeven.

Als er geen indeling wordt gevonden, draait het algoritme opnieuw. Deze keer is het toegestaan om de restricties van de graaf te negeren. Het is dan mogelijk om al gespeelde wedstrijden opnieuw te genereren. Er word geprobeerd dit te minimaliseren door verkeerde keuzes een hogere penalty te geven.

Als er in de gegeven teams uitgevallen teams zitten, moeten deze helemaal achteraan in de lijst staan. De wedstrijden tussen uitgevallen teams (en het ByeTeam) worden in de graaf mogelijk gemaakt, ook als deze al gespeeld zijn. Vervolgens wordt er in het algoritme een hogere penalty toegekend als een actief team aan een inactief team wordt gekoppeld, en heeft het koppelen van inactieve- en byeteams een penalty van nul. Zo wordt gepoogd inactieve teams aan elkaar te

koppelen zodat het aantal niet-spelende teams in een ronde zo klein mogelijk blijft. Er zijn geen garanties over het aantal niet-spelende teams, het geheel werkt onder best-effort. Vanwege uitgevallen teams wordt het aantal mogelijke rondes in een poule verlaagd.

4.3.2.3 Complexiteit

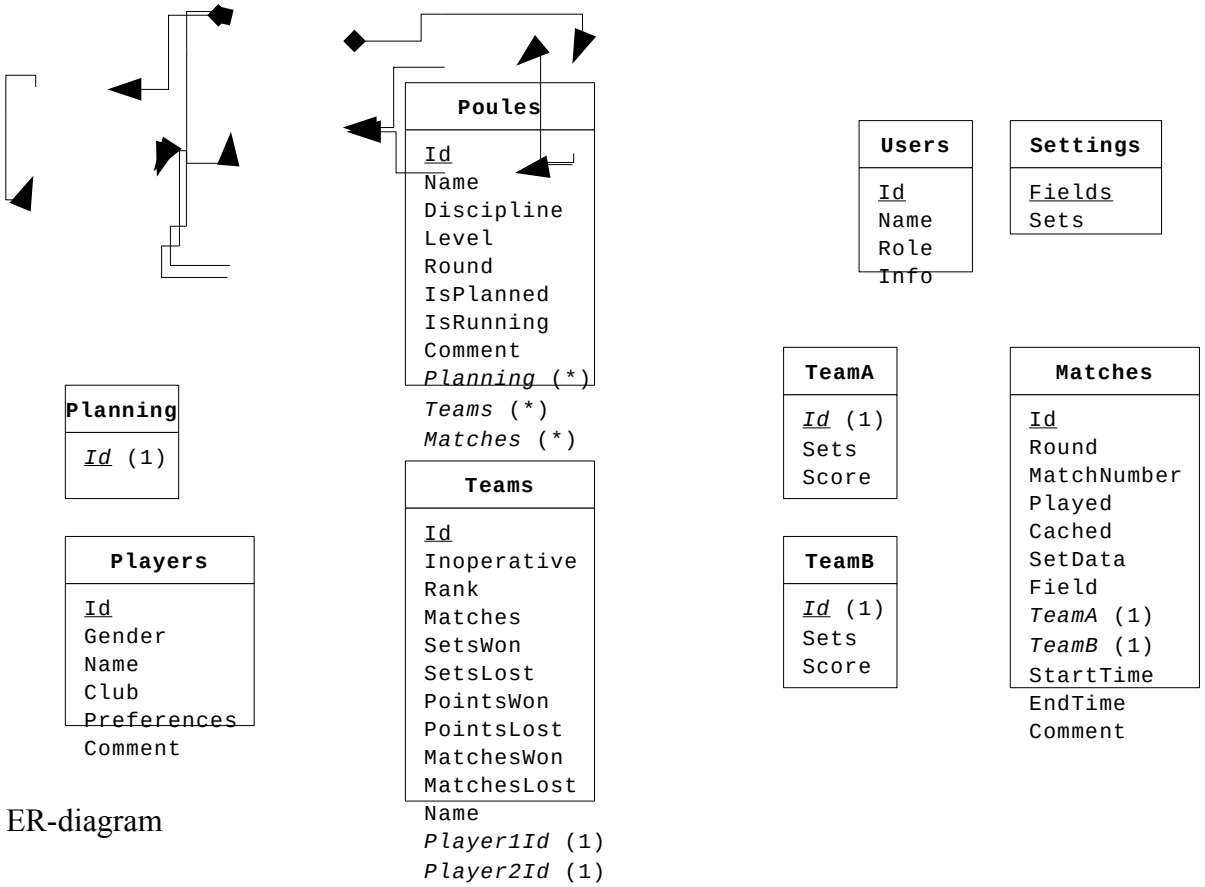
De looptijd van het algoritme is exponentieel in het aantal teams. Vanwege de low-level implementaties van de graaf (snellere 16,32,64 en 128 team versies) en het algoritme zelf is een redelijk aantal teams nog te doen. Meer dan 30 teams in een enkele poule is niet aan te raden. In dat geval is het beter twee verschillende poules aan te maken. Extra versnellingen zoals het hergebruik van objecten bleken geen effect te hebben op de looptijd, deze wordt voor het grootste deel bepaald door het langslopen van alle mogelijkheden.

4.4 Database

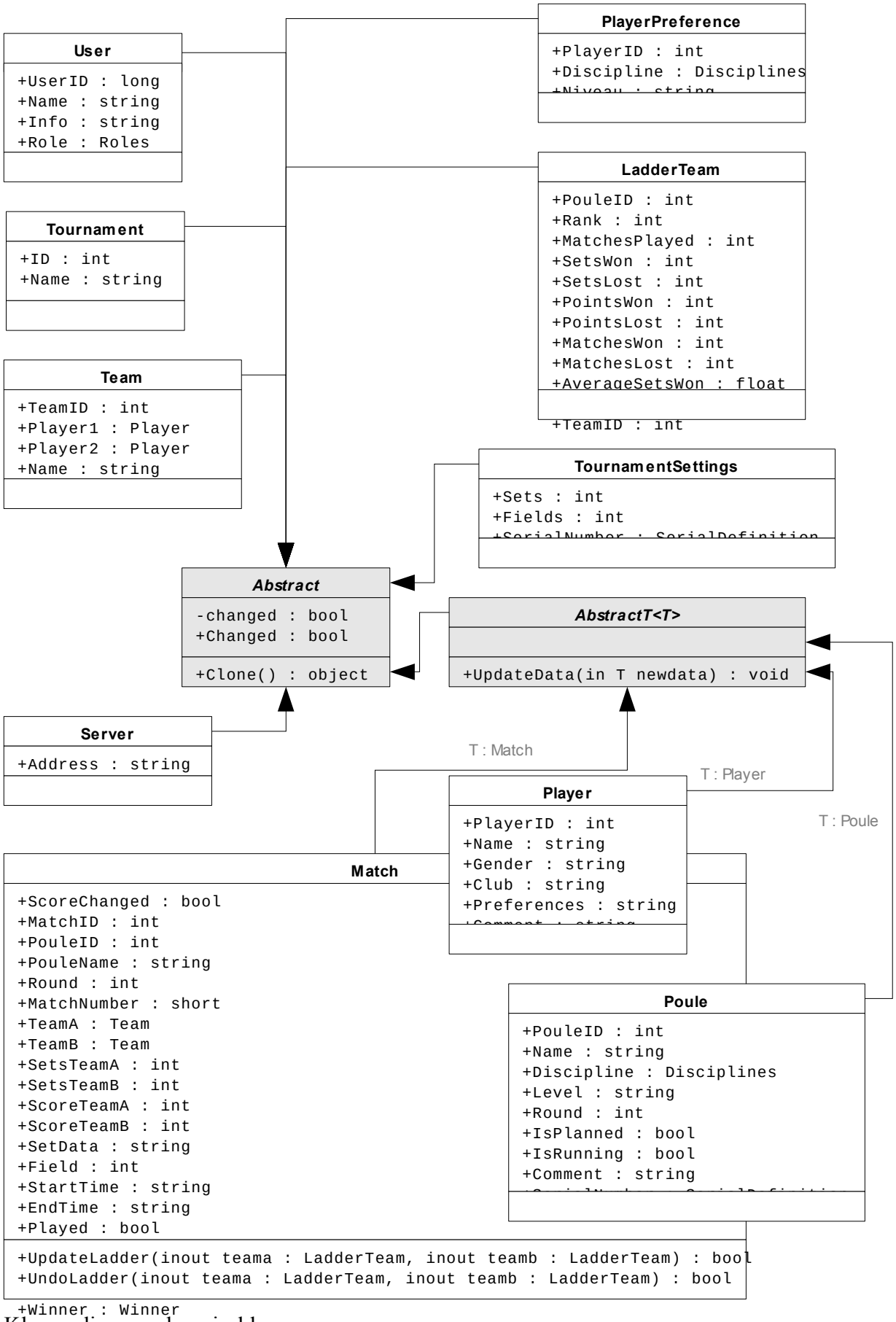
Oorspronkelijk was het de bedoeling om SQLite te gebruiken voor het beheer van de gegevens, maar dit bleek in het kader van de initiële ontwikkeling niet bruikbaar genoeg Daarom is bij de uiteindelijke versie gekozen om de oorspronkelijk als tussenoplossing bedoelde dataopslag met geserialiseerde objecten verder uit te werken. Door het oorspronkelijk voor een relationele database bedoelde ontwerp is het echter wel mogelijk om de objectrelaties in de vorm van een ER-diagram (onder) weer te geven.

De daadwerkelijke opslag gebeurt door de toernooi-data naar XML tekst te converteren en in een tekstbestand op te slaan.

Deze implementatie heeft wel als gevolg dat er geen gebruik gemaakt kan worden van de gemakken die een relationeel database-systeem biedt. Zo moest alle query- en controle-logica handmatig geschreven worden. Dit is gedaan in het Server project.



ER-diagram



Klassendiagram domain-klassen

4.5 Model View Controller (MVC)

Om de code, en de interne structuur in het algemeen, overzichtelijk te houden is de code voor de aan de gebruiker zichtbare onderdelen van Sharp Shuttle onder te verdelen in drie categorieën: Model, View en Controller.

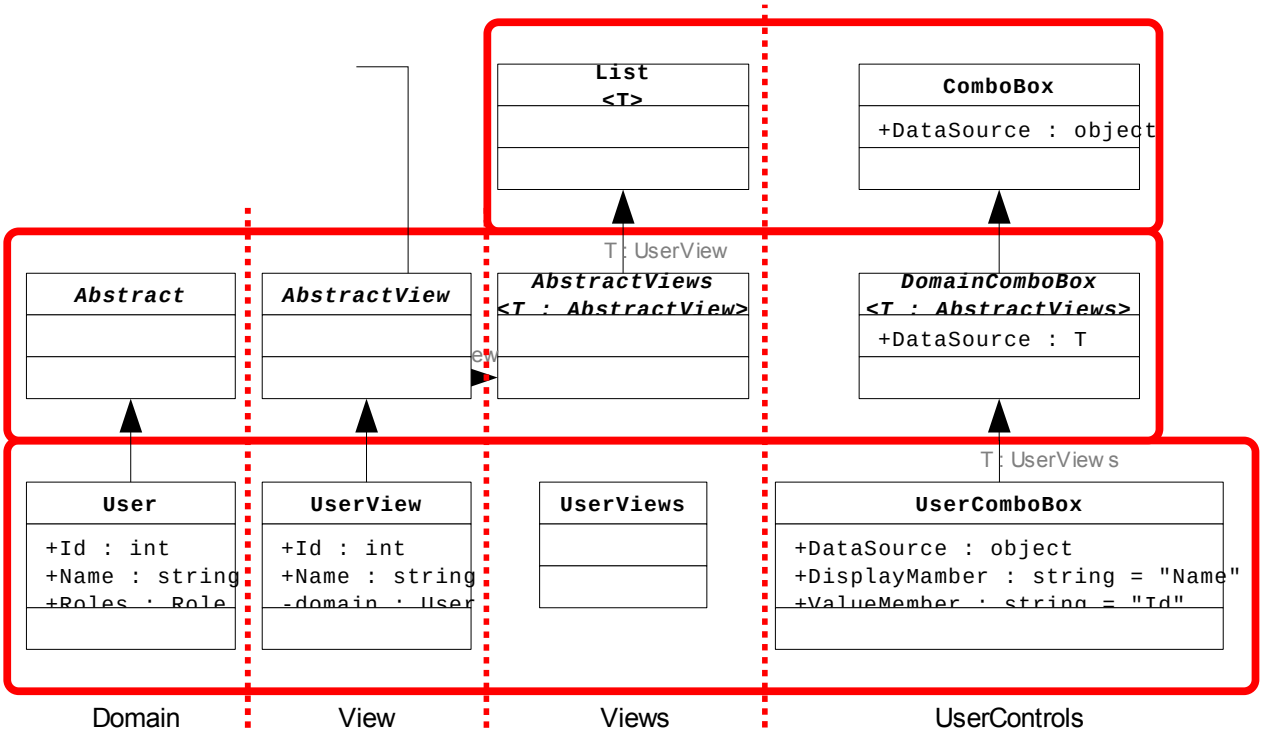
Model staat voor het domeinmodel, en hierin staat beschreven hoe gegevens in elkaar zitten, hoe ze worden opgeslagen en wat de onderlinge relaties zijn. De Model-categorie zal verder worden toegelicht in hoofdstuk 4.5.1.

Tot de View-categorie behoort alles waar de gebruiker direct mee in aanraking komt. Dit zijn over het algemeen vensters en de daaraan verbonden in- en uitvoer componenten, maar ook methodes voor invoervalidatie en methodes voor verhoogd gebruiksgemak horen hierbij. Aan deze categorie is geen aanvullend hoofdstuk geweid, enkele onderdelen hiervan zullen echter wel uitgebreid aan bod komen in hoofdstuk 9.

De derde categorie, Controller geheten, omvat de code die de koppeling tussen de eerste twee groepen vormt. Hier is de business logica geïmplementeerd en vindt de vertaling van ruwe data naar leesbare informatie en v.v. plaats. Uitgebreidere uitleg hierover staat in hoofdstuk 4.5.2. Voor meer uitleg over de genoemde vertaalslag is hoofdstuk 4.5.1 geschikt.

4.5.1 Domain en View klassen

Het domainmodel wordt geimplementeerd met domain klassen. Deze klassen zijn over het algemeen objectgeörinteerde afbeeldingen van de entiteiten in het onderliggende relationele database-model. Een klassendiagram staat hierboven. Voor de interactie met de gebruiker wordt niet rechtstreeks gebruik gemaakt van de domain klassen, maar van zogeheten view klassen. Dit zijn als het ware plug & play-afbeeldingen van domain klassen, die de ruwe, voor rechtstreekse weergave ongeschikte, data converteren naar een vorm die door de vensters rechtstreeks aan de gebruiker getoond kan worden.



Klassendiagram Domain, View, UserControl

Er is dus ook een relatie tussen de domain klassen en de view klassen. Aan de hand van een versimpeld voorbeeld zal hierboven verduidelijkt worden hoe de domain objecten, views en de (gespecialiseerde) usercontrols samenwerken.

De domain en view klassen zijn gedefinieerd in het Shared project.

4.5.2 Controllers

De controllers representeren de use cases die in het Functioneel Ontwerp beschreven staan. Ze komen dan ook alleen in het Client project en het Server project voor. Hier zijn ze te vinden in de namespace genaamd Controls.

In deze klassen staat beschreven hoe de applicatie zijn werk moet doen, en weliswaar het “echte” werk, zoals het uitzoeken welke aanroepen er gedaan moeten worden om gegevens op te halen of weg te schrijven. Vaak houden deze klassen ook bij waar de gebruiker mee bezig is en welke data daarvoor relevant zijn. Vaak hebben controls methodes voor het ophalen en wegschrijven van data, kennen ze validatiemethodes en waarborgen ze de integriteit van de betrokken data.

Contols kunnen niet opzichzelfstaand ingezet worden.

5 Client project

Het client project vormt de kern van het programma, in ieder geval voor de gebruiker. De client bestaat voor een groot deel uit User Interfaces waaraan de benodigde functionaliteiten zijn gehangen. Het project compileert naar een Windows Application genaamd Client.exe. De structuur van het client project ziet er als volgt uit:

Client.csproj – De Project file.
- Controls – De data kant van de verschillende vensters.
- Dockpanel.2.3.1 – De library voor het docken van vensters.
- Forms – Bevat alle vensters aan de client kant.
  - Clientwizard – Alle wizardschermen van de client.
  - Popupwindows – De verschillende popupschermen die vanuit het menu tevoorschijn komen.
- Printers – Al het gebruik van de printer.
- Resources – Files nodig voor het systeem.
- Configurations.cs – Klasse voor het bijhouden van instellingen.
- MainForm.cs – Het belangrijkste venster dat opent bij opstarten.
- Program.cs – Entry point voor het programma.

5.1 MainForm

Het MainForm is het scherm dat geopend wordt bij opstarten van de client. Deze toont standaard een login-wizard, omdat de client zonder verbinding met een server vrij nutteloos is. Na het verbinden met een server zijn de verschillende functionaliteiten beschikbaar vanuit het menu. Welke functionaliteiten beschikbaar zijn hangt af van de gekozen gebruiker en wordt bepaald door de FunctionalityControl. Alle andere schermen die in de client geopend worden vallen onder MainForm, er komen dus geen extra schermen in de taakbalk.

5.2 Dockpanel

Docken is het plaatsen van een scherm binnen een groter geheel. Hiervoor wordt een dockinglibrary

gebruikt. Deze plaatst de geopende schermen in een taakbalk en biedt functionaliteit om, door middel van naar de juiste plek slepen, verschillende vensters naast/boven elkaar te openen.

5.3 Overige delen

Informatie over Printers en Configurations zijn te vinden in respectievelijk hoofdstuk 9.6 en 9.5. Het gebruik van Controls wordt uitgelegd in hoofdstuk 4.5.2. De specifieke delen van de GUI die in sommige Forms gebruikt worden maken deel uit van het UserControls project. Hier valt ook de basis van de verschillende wizards onder.

6 Server project

Het server project is het datacentrum van het systeem. Met een minimale User Interface ligt de nadruk op de database en de verbindingen met clients. Het project compileert naar een Windows Application genaamd Server.exe. De structuur van het server project ziet er als volgt uit:

Server.csproj – De Project file.
- Communication – Regelt alle serverside communicatie.
- Controllers – Geeft de user interface controle over de server zelf.
- Database – Alles wat met de database te maken heeft.
- MainForm.cs – Het enige GUI deel aan de server kant.
- Program.cs – Entry point voor het programma.

Het MainForm van de server biedt de mogelijkheid om bestaande toernooien te laden of een nieuwe op te starten. Verder kan de server gestart en gestopt worden en is er enige monitoring voor het netwerkverkeer. Informatie over Communication staat in hoofdstuk 9.1 en de database wordt uitgelegd in hoofdstuk 4.4.

7 Shared project

Het shared project bevat onderdelen van het systeem die gebruikt worden op zowel de client als de server, samen met onderdelen die eigenlijk niet bij een specifiek onderdeel horen. Het project compileert naar een Class Library genaamd Shared.dll. De structuur van het Shared project ziet er als volgt uit:

Shared.csproj – De Project file.
- Algorithms – Alle algoritmes en de bijbehorende toegangsmethoden.
- Communication – Klassen die te maken hebben met netwerkcommunicatie.
  - Exceptions – De mogelijke exceptions die tijdens de communicatie kunnen ontstaan.
  - Messages – De verschillende berichten die het netwerk ondersteunt.
    - Requests – Berichten van de client naar de server.
    - Requests – Berichten van de server naar de client..
  - Serials – Klassen die de versies van data bijhouden.
- Data – De interface en implementatie voor toegang tot data.
- Datastructures – Datastructuren die vooral voor algoritmes gebruikt worden.
- Delegates – Een aantal standaard delegate methodes.
- Domain – De objecten die over het netwerk verstuurd worden.
- Extensions – Een paar utility methodes.
- Logging – Logger voor het wegschrijven van belangrijke gebeurtenissen zoals fouten.

Parsers – Voornamelijk gebruikt om inschrijvingen in te lezen.
Sorters – Vergelijkingsklasses om bepaalde datasets te ordenen.
Views – De verschillende views die werken over de domein objecten.

7.1 DataStructures

Voor de algoritmes zijn een aantal custom datastructuren geschreven. Hieronder zijn twee versies van counters die atomic ophogen en een heap. Verder zijn er verschillende graaf implementaties, die naargelang hun maximale grootte kleiner is sneller worden. Deze implementeren allemaal de Graph interface. De snelheid van deze datastructuren vormen de basis van voornamelijk het Ladder algoritme.

7.2 Domain

In Domain worden de belangrijkste objecten van het programma gedefinieerd. Dit zijn de objecten waar de views op werken, die over het netwerk worden verstuurd en ook in de database worden opgeslagen. Deze objecten implementeren allemaal de Abstract interface zodat veranderingen in de data kunnen worden bijgehouden. Een bijzonder object is de ChangeTrackingList, die eigenlijk een Collection is, waarin veranderingen zoals verwijderen of toevoegen van elk Abstract object bijgehouden worden.

7.3 Logging

Om enige fouten in het programma terug te kunnen vinden is het belangrijk dat bijzondere gebeurtenissen opgeslagen worden. Hiervoor is een static Logger klasse beschikbaar. Deze schrijft threadsafe logmessages weg naar een tekstfile. De filenaam wordt bepaald door de datum en tijd op het moment van aanmaken, waardoor elke logfile uniek is.

7.4 Parsers

Om spelers te importeren is een parser klasse beschikbaar. Deze parset XML documenten gegenereerd door de MySQL Administrator tool. De meegeleverde parser is alleen geschikt voor de database structuur van S.B. Helios. Voor andere databases moet een aparte parser worden geschreven.

7.5 Sorters

Om lijsten te kunnen sorteren is het altijd handig om een CompareTo functie beschikbaar te hebben. Deze zijn in aparte Comparer klasses gemaakt, zodat ze niet in de objecten zelf gezet hoeven te worden en makkelijk zijn aan te passen. Er zijn sorteerders beschikbaar voor LadderTeams, TeamListViews en kolommen in het algemeen.

7.6 Overige delen

De verschillende onderdelen van Communication staan uitgelegd in hoofdstuk 9.1. Informatie over de datatoegang staat in hoofdstuk 9.2, en in hoofdstuk 4.3 worden de verschillende algoritmes behandeld.

8 UserControls project

Voor de User interface zijn aparte UserControls gemaakt die specifiek werken op de Views van het systeem. Deze controls vervangen in sommige gevallen de standaard C# controls. Het project compileert naar een Class Library genaamd UserControls.dll. De structuur van het UserControls project ziet er als volgt uit:

Server.csproj – De Project file.
- AbstractControls – Definieert een aantal abstracte controls die in het project gebruikt worden.
- AbstractWizard – Definieert een abstracte wizard. Nadere uitleg in hoofdstuk 9.3
- Docking – Standaard form dat gedockt kan worden.
- MatchControls – Controls voor het weergeven van wedstrijden.
- NotificationControls – Control die wijzigingen in data bijhouden en data updaten.
- PlayerControls – Controls voor het weergeven van spelers.
- PouleControls – Controls voor het weergeven van poules.
- Resources – Files nodig voor het project.
- ServerControls – Control voor het weergeven van poules.
- TeamControls – Controls voor het weergeven van teams.
- UserControls – Control voor het weergeven van gebruikers.

De klassen die de standaard C# control vervangen zitten in de PlayerControls, PouleControls, etc. Hun relatie tot de views is verduidelijkt mbv. een klassendiagram in hoofdstuk 4.5.1 (op pagina 13).

9 Implementatie

In dit hoofdstuk komen implementatieaspecten aan bod die nog niet eerder besproken zijn maar wel een groot belang hebben voor het programma.

9.1 Netwerkcommunicatie

De netwerkcommunicatie verzorgt alle communicatie tussen een client en de server en daarmee ook tussen de clients onderling. Hierdoor is het een van de belangrijkste onderdelen van het programma. Het protocol zelf wordt beschreven in hoofdstuk 9.1.1 en het gebruik van het protocol staat in hoofdstuk 9.1.2.

9.1.1 Protocol

Het eigengemaakte berichtenprotocol loopt over een TCP protocol. De client maakt via TCP verbinding met de server op een gegeven IP-adres en poort 7015. De poort is niet instelbaar. Sharp Shuttle moet daarom niet gebruikt worden in combinatie met de programma's City of Heroes, City of Villains, Talon Webserver, RealAudio en LanSafe, aangezien deze poort 7015 ook kunnen gebruiken.

Over het TCP protocol kunnen geserializeerde objecten worden gestuurd. Sharp Shuttle bied alleen ondersteuning voor berichten van het type Message. Messages komen in de vormen placeholder (leeg), PING (ongebruikt), PONG (ongebruikt), REQUEST, RESPONSE, SERIALUPDATE en GOODBYE. Berichten die niet van de juiste vorm of type zijn worden genegeerd.

Het protocol heeft geen beveiliging of encryptie. Er is vanuit gegaan dat hier in een vertrouwelijke LAN-omgeving niet op gelet hoeft te worden.

9.1.2 Netwerkgebruik

Alle code voor het netwerk is te vinden in de map Shared.Communication voor de client en algemene delen, en Server.Communication voor de server. De Communication klassen bevatten de methoden voor het opzetten, stoppen en afhandelen van een verbinding. Zowel aan de server als aan de client kant is er een methode die een bericht verstuurt en een loop op een aparte thread, die door middel van polling inkomende berichten afhandelt.

9.1.2.1 Serials

Serials overlappen enigszins met hoofdstuk 9.2, maar omdat ze bij de netwerkcommunicatie ook aan bod komen, worden ze hier al uitgelegd. In wezen zorgen SerialNumbers voor de synchronisatie van de data.

Elke specifieke verzameling van data die in Sharp Shuttle wordt gebruikt, heeft een SerialNumber. Dit nummer geeft de versie van de data aan. Deze nummers worden zowel op de server als de client bijgehouden in de SerialTracker en kunnen door het gedistribueerde karakter van het programma verschillen, maar het nummer op de server bepaalt welke versie op dat moment de gebruikte is. Aan de hand van dit nummer kunnen de server en de client bepalen of een wijziging in de data doorgevoerd mag worden en bij de client kan worden bepaald of de aanwezige data wel de laatste versie is. Bij een wijziging op de server wordt het serienummer verhoogd. De dataverzamelingen kunnen overlappen, zo kan het zijn dat een enkele wijziging op de server voor meerdere veranderde SerialNumbers zorgt.

9.1.2.2 Berichten

Over het netwerk worden berichten van het type Message gestuurde. Deze klasse zelf is een leeg bericht en moet daarom extended worden. De vier gebruikte types zijn RequestMessage, ResponseMessage en SerialUpdateMessage.

RequestMessages worden door de client naar de server gestuurd om data op te vragen of te wijzigen. Op elke RequestMessage volgt een ResponseMessage van de server naar de client, waarin data wordt opgeleverd of wordt aangegeven of de wijziging gelukt is. Bij een doorgevoerde wijziging wordt ook naar alle clients een SerialUpdateMessage gestuurd, om te laten weten dat van sommige dataverzamelingen een gewijzigde versie beschikbaar is.

Verder wordt een GOODBYE message verstuurd als een programma wordt afgesloten. Hierdoor kan de andere kant van de verbinding de verbinding ook netjes afsluiten.

9.1.2.3 RequestContainers

Bij het versturen van een RequestMessage naar de server wordt gelijk een RequestContainer opgeleverd. Deze container is eerst nog leeg maar zal later het antwoord bevatten. De request zelf wordt in de RequestPool gezet. Hierin staan alle requests die verstuurd zijn, maar nog geen antwoord hebben. Als de ResponseMessage behorend bij een request binnenkomt, wordt het antwoord in de bijbehorende RequestContainer gezet. Als voor die tijd om het antwoord word gevraagd, blijft de opvragende thread wachten op het antwoord of een time-out. Het opvragen van het antwoord uit de container returned altijd, hetzij met het juiste antwoord, hetzij met een exception.

9.1.2.4 Exceptions

In elke ResponseMessage vanaf de server kan een CommunicationException zitten. Dit zijn Exception onstaan bij het opvragen of wijzigen van data op de server, zoals het opvragen van niet bestaande data of het wijzigingen van data met een verouderde versie. Deze exceptions worden alleen doorgegeven en hebben met het netwerk zelf weinig te maken.

Het netwerk zelf heeft drie eigen exceptions, namelijk de NotConnectedException, de CommuncationTimeOutException en de WrongTypedAnswerException. De eerste komt voor in de DataCache als een bericht moet worden verstuurd terwijl er geen verbinding is. De tweede wordt gegooid door de RequestContainer als het antwoord niet snel genoeg geleverd word, en de laatste exception wordt gegooid als het geleverde antwoord niet voldoet aan het verwachte antwoordtype.

9.2 Datatoegang

De data van het programma wordt bewaard in een database op de server. Deze data moet ook op de client beschikbaar zijn om te tonen en te verwerken. Hiervoor is een apart object gemaakt dat data zonder al te veel moeite oplevert en zelf de communicatie met de server regelt. Hierdoor kan er in de client makkelijk data opgevraagd en gewijzigd worden.

9.2.1 IDataCache

IDataCache is de interface die alle operaties op de database beschrijft. Deze operaties worden door het DataCache object geïmplementeerd. In de interface staan parameters, return types en exceptions beschreven. Een belangrijk punt om op te merken is dat GET operaties de gevraagde waarden opleveren en exceptions gooien, terwijl SET operaties het succes van de wijziging opleveren in boolean formaat en exceptions in de meegegeven out parameter opleveren. Dit is gedaan om de vele mogelijke exceptions bij een set operatie makkelijker af te kunnen handelen.

9.2.2 DataCache

DataCache is de implementatie van IDataCache. Het is een singleton object dat niet alleen communicatie naar de server verzorgt, maar ook lokaal data bewaart zodat het niet nodig is voor elke aanroep data op te halen van de server.

Elke aanroep op de DataCache kan maar met 1 tegelijkertijd worden uitgevoerd. Hiervoor is bij elke operatie een Mutex beschikbaar. Bij een aanroep wordt gekeken of de lokale versie nog bruikbaar is. De lokale versie word dan teruggegeven (GET) of de gevraagde wijziging wordt doorgestuurd naar de server (SET). Als de lokale versie niet de laatste versie is, wordt de laatste versie van de server gehaald (GET) of een DataOutOfDateException teruggegeven(SET). Bij communicatie met de server wordt het antwoord in de juiste vorm gecast en het resultaat of de exception teruggegeven aan de aanroeper.

9.3 UserControls.AbstractWizard

AbstractWizard is een stel klassen die het makkelijk maken om een wizard te implementeren. Het voordeel van het gebruik van deze basis is dat bij het ontwikkelen geen moeite gedaan hoeft te worden om de stappen van een wizard in hetzelfde jasje te stoppen en automatisch een consistent uiterlijk en een consistent gedrag gegarandeerd is, zonder dat er onnodig beperkingen opgelegd worden. Er zijn dan ook enkel de minimale ingrediënten voor een werkende wizard aanwezig. Alle

toevoegingen en aanpassingen die afwijkend gedrag tot gevolg moeten hebben, horen door de ontwikkelaar zelf gedaan te worden.

De basiswerking van een wizard wordt als volgt verondersteld:

- Er zijn vier manieren van navigeren: Volgende stap, vorige stap, klaar en annuleren/sluiten.
- Er zijn één of meer stappen.
- Elke stap heeft alleen kennis van zijn eigen in- en uitvoer.
  Er kan natuurlijk d.m.v. een control achter de schermen globaal in- en uitvoer gedaan worden.
- Elke stap bepaalt de toegestane manieren van navigeren in zijn aanwezigheid.
  Welke navigatiemiddelen er zichtbaar zijn, en welke daarvan voor gebruik toegestaan zijn.
- De aparte stappen retourneren de invoer in één object. De wizard retourneert de verzamelde invoer van alle stappen in één object.
  De vorm van geretourneerde data is niet vooraf voor willekeurige gevallen gespecificeerd.

9.3.1 Implementatie

Om een nieuwe wizard te maken zijn twee dingen nodig: Het maken van een nieuwe WizardForm-klasse, afgeleid van AbstractWizardForm en het maken van een nieuwe, van GenericAbstractWizardStep afgeleide, klasse (hieronder vaak WizardStep genoemd). De reden voor het bestaan van de AbstractWizardStep klasse is om automatische instantiëring van WizardSteps te verwezenlijken, maar heeft voor de ontwikkelaar van wizards geen verdere betekenis. Een geïmplementeerde wizardstep is standaard niet te zien in de designer. Om de step visueel te kunnen bewerken, moet de extensie tijdelijk veranderd worden van genericAbstractWizardStep<object> naar UserControl.

Bij de AbstractWizard wordt gebruik gemaakt van generics voor het retourneren van invoer. Als er geen invoer geretourneerd moet worden, is het het meest overzichtelijk om als retour-type het type “object” te gebruiken.

In de reeks afhandelingsmethodes rond de navigatie, is sprake van “Ext” en “Int” methodes. Dit betekent extern respectievelijk intern, en moet gezien worden vanuit het perspectief waarin de methode geschreven is. Bijvoorbeeld WizardStep.validateNextInt() is de stap-interne validatiemethode voor het naar de volgende stap gaan, en staat dus ook in de WizardStep klasse. De WizardStep.validateNextExt() methode (eigelijk is het een event) kan dus gebruikt worden voor validatie buiten de WizardStep klasse om. De methodes die aan dit event gekoppelt worden, zullen in de praktijk veelal in de WizardForm (en dus buiten de WizardStep) staan. Het doel van dit onderscheid is om bijvoorbeeld (intern) primitieve invoervalidatie (zoals het invoerformaat controleren) te kunnen doen, terwijl de correct ingelezen gegevens vervolgens (extern) op een hoger niveau gevalideerd (zoals het controleren op dubbele data) kunnen worden, voordat ze verwerkt worden.

9.3.1.1 Navigatie afhandeling

Voor de afhandeling van de verschillende navigatieopdrachten vanuit de gebruiker zijn een vast aantal methode-aanroepen gegeven. De hier beschreven geneste aanroepen beschrijven het volledig afhandelen van een navigatieopdracht, zonder dat zich bijzondere situaties voordoen. In de praktijk zal het vaak voorkomen dat bijv. de validatiemethodes de reeks opdrachten onderbreken.

Volgende:

F[1].btnNext_Click
    S.Next: (volledige afhandeling binnen S)
        S.validateNext (gezamenlijke noemer)
            S.validateNextInt
            S.read_input
            S.validateNextExt
        F.next (gezamenlijke noemer)
            F.nextInt (wizard-interne post-validatie methode)
            (F.nextExt kan later nog toegevoegd worden)
            Step++ (volgende stap daadwerkelijk tonen)

Vorige:

F.btnPrevious_Click
    S.Previous: (volledige afhandeling binnen S)
        S.validatePrevious (gezamenlijke noemer)
            S.validatePreviousInt
            S.read_input
            S.validatePreviousExt
        F.previous (gezamenlijke noemer)
            F.previousInt (wizard-interne post-validatie methode)
            (F.previousExt kan later nog toegevoegd worden)
            Step-- (vorige stap daadwerkelijk tonen)

Klaar:

F.btnFinish_Click
    S.Finish: (volledige afhandeling binnen S)
        S.validateFinish (gezamenlijke noemer)
            S.validateFinishInt
            S.read_input
            S.validateFinishExt
        F.finish (gezamenlijke noemer)
            F.stepFinishInt (wizard-interne post-validatie methode mbt. invoer van S)
            (F.stepFinishExt kan later nog toegevoegd worden)
            F.completeFinishInt (wizard-interne post-validatie methode mbt. complete invoer van F
            (F.completeFinishExt kan later nog toegevoegd worden)
            F.close (wizard sluiten)

Annuleren/Sluiten:

Hier is geen bijzondere afhandeling voor. Er wordt gewoon het gebruikelijke protocol voor het sluiten van vensters gebruikt.

9.3.1.2 WizardForm

In een WizardForm komen alle onderdelen van een wizard bij elkaar: Het venster, de

1 F betekent Wizard**F**orm; S betekent Wizard**S**tep

navigatiemiddelen en de aparte stappen. Bij het maken van een AbstractWizardForm subklasse moeten een aantal methodes verplicht geïmplementeerd worden:

public Constructor ():

Het zetten van de titel gebeurt hier mbv. een aanroep naar de superklasse en de titel als parameter.

public void Init ():

Deze methode wordt automatisch vanuit de constructor aangeroepen en heeft de taak invulling aan de wizard te geven. Hier worden dus alle WizardSteps aangemaakt, geconfigureerd en in een centraal array opgeslagen. Dit array is een reeds gedeclareerde variable in AbstractWizardForm en hoeft alleen nog geïnitialiseerd te worden.

WizardSteps configureren betekent bijv. onder anderen externe validatiemethodes koppelen.

protected void completeFinishInt (invoer):

Deze methode handelt binnen de scope van de WizardForm de invoer van alle stappen af. Het verzamelen van die invoer moet handmatig geïmplementeerd worden. Deze stap gebeurt net voor de wizard sluit, nadat alle validatie geslaagd is. Als er geen gebruik gemaakt wordt van verzamelde invoer, kan bijv. een null-waarde geretourneerd worden.

protected invoer readCompleteInput ():

Deze methode retourneert slechts de volledige invoer van alle stappen. Het verzamelen moet gebeuren terwijl de gebruiker de wizard doorloopt.

Methodes die niet verplicht ge-override moeten worden staan hieronder:

public int GetStepCount ():

Van essentieel belang voor wizards met een dynamisch aantal stappen.

protected void setForm():

Mede verantwoordelijk voor de tekst in de titelbalk van een wizard.

protected void nextInt (invoer):

Mogelijkheid om de invoer van een WizardStep buiten dat object om te verwerken. Dit gebeurt nadat de interne verwerking is afgesloten.

protected void previousInt (invoer):

Analoog aan nextInt, maar dan voor de “vorige” navigatie-opdracht.

protected void stepFinishInt (stap-invoer):

Analoog aan nextInt, maar dan voor de “klaar” navigatie-opdracht.

protected void completeFinishInt (volledige invoer):

Analoog aan stepFinishInt, maar deze methode verwerkt de volledige invoer van alle stappen (resultaat van readCompleteInput) in plaats van de invoer van de WizardStep.

Volledige invoer verzamelen

Het verzamelen van de volledige invoer moet handmatig geschreven worden, en is voor elke wizard anders. Het meest voor de hand liggende is om in de nextInt, previousInt en stepFinishInt een statement te plaatsen, die de invoerparameter naar het juiste type cast, afhankelijk van de momentele WizardStep, en het zo getypeerde resultaat uit de steps bewaart.

Het bewaren kan echter overbodig worden, door een control in de scope van de WizardForm te gebruiken die de invoer uit de nextInt, previousInt en stepFinishInt methodes verwerkt.

Zelfs het gebruik van een WizardForm-interne control kan overbodig zijn, als de WizardSteps elk voor zich gescheiden use cases bevatten. In dat geval zullen ze veelal zelf controls gebruiken, terwijl het WizardForm helemaal geen besef van invoer heeft. Het is in ieder geval altijd de moeite waard om over het juiste resultaat-type van WizardSteps, als ook WizardForms na te denken, al is het maar voor het overzicht.

9.3.1.3 WizardStep

Een WizardStep is hetgeen dat op een WizardForm getoond wordt, zodra de gebruiker er naartoe genavigeerd is. Een WizardStep is daarmee weliswaar een usercontrol op een venster, maar het WizardStep object heeft ook invloed op zijn omgeving; namelijk de navigatiemiddelen.

Ook voor het maken van een WizardStep zijn een aantal methodes verplicht te implementeren:

public Constructor (navigatie-listener-delegate):

Net als bij de WizardForm is hier de mogelijkheid om een titel in te stellen. Dit kan door de variable “Text” in te stellen. Daarnaast kan een navigatie-listener-delegate aan de ouder-constructor worden meegegeven. Dit is een methode die wordt afgevuurd als er veranderingen zijn omtrent de beschikbare navigatiemiddelen, evenals aan het einde van de constructor. Het is dus aan te raden om in de constructor van nieuwe WizardSteps ook een navigatie-listener-delegate parameter op te geven, ook al is dit niet verplicht.

Daarnaast moet hier ook ingesteld worden welke navigatiemiddelen er initieel beschikbaar zijn. Dit gebeurt door een zestal reeds gedeclareerde booleans van de WizardStep te initialiseren: previous_visible, next_visible, finish_visible, previous_enabled, next_enabled en finish_enabled.

protected invoer readInput ():

Deze methode is bedoeld voor het uitlezen van usercontrols (invoervelden) die op de WizardStep geplaatst zijn. De methode moet de invoer van mogelijk verschillende zulke usercontrols in één object retourneren. Er mag daarbij van uitgegaan worden dat de interne validatie (bijv. validateNextInt) voor de aanroep al geslaagd is.

Methodes niet verplicht overridden/gekoppeld moeten worden staan hieronder:

protected bool validateNextInt ():

Mogelijkheid om de invoer voor het uitlezen te valideren. Dit is een geschikte plek om invoer op hun syntax te controleren.

protected bool validatePreviousInt ():

Analoog aan validateNextInt, maar dan voor de “vorige” navigatieopdracht.

protected bool validateFinishInt ():

Analoog aan validateNextInt, maar dan voor de “klaar” navigatieopdracht.

public ValidatieDelegate validateNextExt:

Ruimte om een externe validatiemethode te koppelen. De te alideren data komen uit de readInput methode.

public ValidatieDelegate validatePreviousExt:

Analoog aan validateNextExt, maar dan voor de “vorige” navigatieopdracht.

public ValidatieDelegate validateFinishExt:

Analoog aan validateNextExt, maar dan voor de “klaar” navigatieopdracht.

Real time (as-you-type) validatie:

De beste manier is om bij de OnChanged events van usercontrols (zoals invoervelden) een methode te koppelen, die de interne validatie aanroept en met het resultaat daarvan de nodige navigatie-variablen (next_enabled, previous_enabled of finish_enabled) zet. Daarna moet enkel nog een melding gemaakt worden dat deze waardes verandert zijn, door FireStepsHaveChanged() van de WizardStep aan te roepen.

Invoer eerder dan WizardForm verwerken:

Hiervoor moeten de Next, Previous en/of Finish methodes van de WizardForm overriden worden (niet vergeten de methodes van de ouder-klasse aan te ropen), of er moet in de nextInt, previousInt en/of finishInt methodes eerst een terugkoppeling naar het juiste WizardStep objekt gemaakt worden.

Een minder nette methode is aan het einde van de stap-interne validatiemethodes het verwerken te laten gebeuren.

Stapsgewijze invoer geheel buiten de wizard verwerken:

Hiervoor zouden de eerder WizardForm.nextExt, .previousExt en .finishExt toegevoegd kunnen worden, die vanuit de WizardForm.nextInt, .previousInt en .finishInt uitgevoerd zouden moeten worden.

9.4 Usercontrols.NotificationControl

De NotificationControl is in het leven geroepen om twee redenen. Ten eerste was er een systeem nodig dat op een makkelijke manier veranderingen van de database op een eenduidige manier aan de gebruiker laat zien. Ten tweede moest er een systeem komen om op een simpele manier in een aparte thread data van de server te kunnen halen of weg te schrijven.

Door deze twee eisen bij elkaar te voegen is een systeem gemaakt dat de status van de data bijhoudt. Via de IserialFilter wordt aangegeven in welke events een scherm geïnteresseerd is. De NotificationControl zal zichzelf dan op deze events registreren. Als een van de events optreed wordt bij de IserialFilter gekeken of de wijziging in de data effect heeft gehad op de data die ook echt gebruikt word. Dit wordt bepaald door de methodes in IserialFilter die true of false kunnen teruggeven.

Als de data veranderd is, en de wijziging heeft een effect, weet de NotificationControl dat de data out of date is. Vanaf dat moment kan het uiterlijk van de control veranderen waardoor de gebruiker weet dat de data op de server veranderd is. De gebruiker kan dan de data refreshen door op de control te klikken. Hetzelfde gebeurt ook de andere kant op. Als de data door de gebruiker gewijzigd is, verandert de control van uiterlijk en functioneer de control als opslaan-knop. Hiervoor is het belangrijk dat er bij lokale wijzigingen de NotificationControl op de hoogte wordt gebracht.

Om te zorgen dat het ophalen of opslaan van de data de GUI niet laat vastlopen, gebeurt dit in een aparte thread. Als de NotificationControl wordt ingedrukt, wordt er gekeken wat de status op dat moment is. Bij een verouderde of gewijzigde status wordt een backgroundworker opgestart, die een delegate methode van het venster aanroept. Hierbij worden twee mogelijke acties meegegeven,

afhankelijk van de status is dat een Reload of een Save. In de delegate wordt bepaald welke acties daarbij worden ondernomen.

Om ervoor te zorgen dat de GUI geen foute dingen kan doen tijdens het opslaan/herladen kan er geregistreerd worden op twee events, WorkStart en WorkFinished. Op deze manier kan de gebruiker zien dat er gewerkt wordt en dat de gebruiker even geduld moet hebben, zonder dat de GUI vastloopt.

9.5 Settings

Voor het bewaren van instellingen is op de client de Configurations klasse beschikbaar. Hierin zijn zowel toernooi- als lokale instellingen makkelijk beschikbaar. Hiermee is er toegang tot veranderende instellingen zonder moeilijk te hoeven doen met file access en database toegang, met de mogelijke exceptions die daar bij komen. Per definitie is er altijd een waarde beschikbaar, deze is alleen niet altijd correct. Configurations zelf is een static klasse die instellingen heeft voor de printer, de laatst verbonden servers, het aantal velden en het aantal sets per wedstrijd.

9.5.1 Lokaal

Lokale instellingen zijn de laatste verbonden servers en de gebruikte printer. Deze verschillen per client. Ze worden opgeslagen in een standaard C# settings file. De inhoud hiervan is aan te passen in Client.Properties.Settings. Op de computer zelf is het bestand te vinden in C:\Documents and Settings\<Username>\Local Settings\Application Data\Client in een bestand genaamd user.config. Dit bestand hoort alleen vanuit het programma aangepast te worden. Om deze instellingen te laden vanaf de file moet bij het opstarten de methode UpdateLocal() worden aangeroepen.

9.5.2 Toernooi

Toernooi instellingen zijn het aantal velden en het aantal sets per wedstrijd. Deze zijn over het hele toernooi gelijk en worden ook opgeslagen in de database. Hiervoor zijn de functies GetSettings en SetSettings beschikbaar in de DataCache, die werken op een TournamentSettings object. Het ophalen van de instellingen gebeurt bij het verbinden met een server en bij een instellingsverandering op de server. De methode ReloadSettings() haalt dan de laatste versie van de settings op. Bij een gewijzigde instelling wordt SaveGlobalSettingsAsync aangeroepen, die de update in een aparte thread doorvoert. Bij een DataOutOfDataException, bijvoorbeeld als er twee updates vlak na elkaar worden doorgevoerd, wordt nog eenmaal geprobeerd de verandering door te voeren op de nieuwe data. Bij andere exceptions of meerdere updates word een wijziging niet doorgevoerd.

9.6 Printers

Sharp Shuttle biedt de gebruiker de mogelijkheid om wedstrijdbriefjes en ranglijsten te printen. Hiervoor wordt gebruik gemaakt van de standaard C# printerklasse PrintDocument. De printer die hiervoor gebruikt wordt kan ingesteld worden via een standaard PrintDialog, bereikbaar vanuit het menu. De gekozen printer is te vinden in Configurations en blijft dus ook na afsluiten van het programma bewaard. Van de gekozen printer wordt aangenomen dat deze A4 print, aangezien allebei de printsoorten daarop werken.

Het printen zelf gebeurt in twee singleton klassen, die luisteren naar de events komend vanuit het PrintDocument. Het grootste verschil tussen de twee ligt in het feit dat RankingPrinter objecten

print, terwijl MatchNotePrinter print vanuit een lijst met MatchNotes. Hierdoor bevat MatchNotePrinter enkele locks om threadingproblemen te voorkomen.

9.6.1 Wedstrijdbriefjes

MatchNotePrinter verzorgt het printen van wedstrijdbriefjes. Bij het aanmaken van een wedstrijd word een wedstrijdbriefje aan de lijst toegevoegd. Vanuit deze lijst worden de briefjes per 4 op een A4 geprint, waarbij er de keus is om de lijst leeg te printen of om alleen volle A4'tjes te printen. Aangezien in de server niet bekend is of een wedstrijdbriefje geprint is en omdat wedstrijdbriefjes soms kwijtraken, is er in PouleInformation een knop beschikbaar om een wedstrijd opnieuw uit te printen.

Op een wedstrijdbriefje staat de naam van de poule, het ID van de match, het rondenummer en de namen van de spelers. Verder zijn er invulvelden voor de eindstand, de score per set tot een maximum van 3 en voor het baannummer. Deze worden op de pagina getekend door methoden die op gegeven plekken een bepaald onderdeel tekenen. Het briefje is verdeeld in 8 rijen, die de verschillende beginhoogtes voor onderdelen aangeven, en een divider, die aangeeft op welk punt de invulboxen beginnen.

Omdat er altijd vier wedstrijdbriefjes op een pagina moeten passen en deze niet aanpasbaar zijn qua grootte, worden de printermarges genegeerd. Dit kan een probleem zijn bij voornamelijk oude printers, die fysiek niet buiten hun marges kunnen printen. Het word aangeraden dit voor een toernooi te testen.

9.6.2 Ranglijsten

Vanuit PouleInformation is de optie beschikbaar om de tussenstand uit te printen, eventueel met de gespeelde wedstrijden erbij. Hiervoor word de RankingPrinter gebruikt. Bij de aanroep word een PrintableRanking object gemaakt, dat vervolgens zichzelf print. In tegenstelling tot MatchNotePrinter print de RankingPrinter op liggende A4'tjes en wordt er rekening gehouden met de printermarges.

Bovenaan de eerste pagina wordt de naam van de poule en de ronde geprint. Hierna worden de teams met hun scores in een tabel onder elkaar gezet. Hierna volgen eventueel nog de wedstrijden. Bij het einde van een pagina wordt de tabel op de volgende pagina voortgezet.

Om met de variabele breedte van de pagina om te gaan is de kolom van de spelernamen van variabele grootte. De beginplaatsen van de verschillende kolommen zijn vast bepaald, waarna de spelerskolom het restant krijgt toegewezen. Deze beginpunten komen uiteindelijk in de starts variabele te staan.

9.7 FunctionalityControl

FunctionalityControl is een singleton klasse in de client, die achter de schermen bijhoudt welke van de bestaande functionaliteiten de gebruiker mag uitvoeren. Ook al is dit het enige centrale punt voor deze informatie, wordt er nergens afgedwongen dat het daadwerkelijke gebruik van de functionaliteit ook is toegestaan. Met andere woorden, de controle of een gebruiker een bepaalde handeling mag uitvoeren is de verantwoordelijkheid van hetgeen (bijv. een button) dat de handeling toegankelijk maakt. Dit (de button) moet dan de FunctionalityControl raadplegen.

Voor het bepalen van de toegestane functionaliteiten worden de gebruikersrollen van de huidige

gebruiker geanalyseerd. Veranderingen treden dan ook enkel bij het in- of uitloggen op. Elke gebruikersrol maakt een aantal functionaliteiten toegankelijk. Deze zijn hardcoded vastgesteld en worden in de hoofdstukken 9.7.1 en 9.7.2 besproken.

9.7.1 Gebruikersrollen

Er zijn zeven gebruikersrollen voor de verschillende onderdelen van Sharp Shuttle. Hieronder staat een overzicht met een korte toelichting van elke rol.

- **Poules**
  Disciplines en niveaus/poules beheren.
- **Players**
  Spelers beheren.
- **Grouping**
  Team- en pouleindelingen maken.
- **Ladder**
  Rondes van poules controleren.
- **Scores**
  Scores invoeren en corrigeren.
- **Courts**
  Velden beheren en wedstrijden plannen.
- **Reader**
  Mag alles lezen, maar niets schrijven.

9.7.2 Functionaliteiten

De functionaliteiten hieronder staan geclusterd voor een beter overzicht. Bij elke functionaliteit is aangegeven welke rol een gebruiker moet hebben om hem uit te kunnen voeren. Waar nodig zijn ook individuele functionaliteiten nader toegelicht.

Ongeclassificeerd:

- **Login** (allen)
- **Logout** (allen)

Poules:

- **NiveauRead** (allen)
  Bestaande niveaus lezen.
- **NiveauWrite** (poules)
- **DisciplineRead** (allen)
  Bestaande disciplines lezen.
- **PouleRead** (allen)
  Bestaande poules lezen.
- **PouleWrite** (poules)

Spelers:

- **PlayerReadBasic** (allen)
  Lezen van minimale spelergegevens.
- **PlayerReadComplete** (players)
  Lezen van volledige spelergegevens.
- **PlayerWrite** (players)

Speler indeling:

- **PlayerPouleRead** (allen)
  Lezen welke speler in welke poules zit.
- **PlayerPouleWrite** (grouping)
  Bepalen in welke poules in speler zit.
- **PlayerTeamRead** (allen)
- **PlayerTeamWrite** (grouping)

Ladder:

- **CloseRound** (ladder)
- **StartRound** (ladder)
- **RestartRound** (ladder)
- **DisableTeam** (grouping)
- **MatchRead** (allen)
  Wedstrijdgegevens lezen.
- **MatchWrite** (ladder)

Scores:

- **ScoreRead** (scores, ladder)
- **ScoreWrite** (scores)

Velden:

- **CourtRead** (allen)
- **CourtWrite** (courts)

Overige:

- **PrintRanking** (ladder)
  Ladder en wedstrijden van huidige ronde printen.
- **PrintMatchPaper** (ladder)
  Wedstrijdbriefjes printen.
- **SetsRead** (allen)
  Instelling hoeveel sets een wedstrijd heeft lezen.
- **SetsWrite** (ladder)

9.7.3 Toepassing

Feitelijk komt de FunctionalityControl pas in beeld als een venster toegang biedt tot veel verschillende functionaliteiten, die voor verschillende gebruikersrollen toegankelijk zijn, zoals het MainForm. Daarom heeft dit venster een eigen control dat de toegankelijkheid van functionaliteiten stuurt: MainFormActionControl. Deze control is de schakel tussen het MainForm en de FunctionalityControl, waardoor geen venster-specifieke code in de FunctionalityControl, en ook geen grote hoeveelheden functionaliteit-gerelateerde code in MainForm opgenomen hoeft te worden.

9.8 Deployment

9.8.1 Codesigning

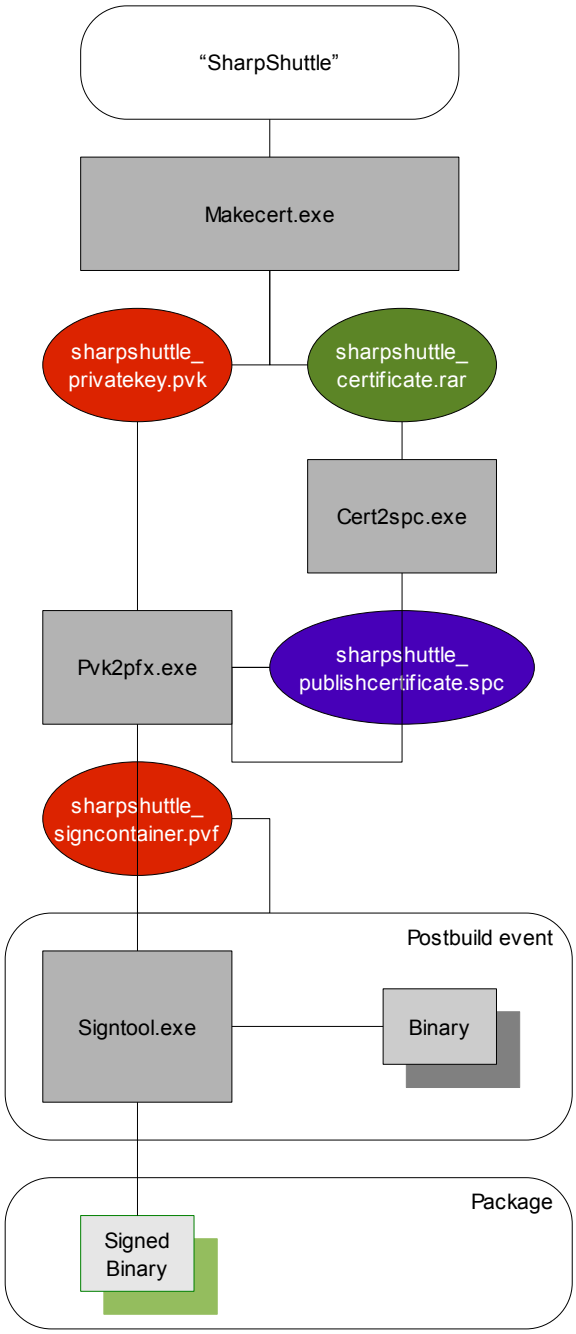
Tegenwoordig is het zeer belangrijk om en computer te beschermen tegen kwaadaardige programma's. Daarom maken ontwerpers steeds meer systemen om bepaalde software te weigeren op een computer. Een van de oplossingen is dat men tegenwoordig vereist dat programma's ondertekend zijn met een certificaat.

Dit ondertekenen heeft als voordeel dat niemand je programma kan aanpassen zonder het certificaat ongeldig te maken. Op deze manier is een programma dat met een vertrouwd certificaat is ondertekend ook automatisch vertrouwd. Er is alleen één probleem; een vertrouwd certificaat kost per jaar meer dan honderd dollar en moet elk jaar vernieuwd worden. Voor MKB ontwikkelaars wordt het dus moeilijker gemaakt om goede software op de markt te brengen.

Bij PC systemen met een Microsoft Windows besturingssysteem kan men duidelijk merken wanneer software wel of niet geldig wordt bevonden. Op het moment dat men software installeert zonder een geldig certificaat heeft het desbetreffende programma minder rechten op het systeem. Op deze manier kan men er dus niet standaard van uit gaan dat netwerkgebruik en file IO mogelijk is.

Tevens kan de gebruiker zelf aangeven of hij wel of niet de software en het certificaat vertrouwd. Van het feit dat de gebruiker zelf mag aangeven dat hij een ongeldig certificaat vertrouwd wordt door Sharp Shuttle gebruik gemaakt.

Het hele signing process is gebaseerd op zelfgemaakte certificaten die niet zijn ondertekend door een Certificate Authority. Deze certificaten zullen standaard dus niet vertrouwd zijn, maar kunnen door de gebruiker wel als vertrouwd worden gekozen.



Codesigning diagram

He genereren van een Code Signing Certificate kan met behulp van de OpenSSL library of met tools uit de Visual Studio omgeving. In deze documentatie zal aangenomen worden dat er een certificaat wordt gemaakt door de tools uit Visual Studio.

De eerste stap is het genereren van een certificaat en een bijbehorende privesleutel. Hiervoor wordt de tool MakeCert gebruikt. Dit is onderdeel van de Microsoft DSK en deze moet dan ook geïnstalleerd zijn. Door "makecert.exe" aan te roepen met de volgende argumenten zal er een privé sleutel (.pvk) en een certificaat met een publieke sleutel (.cer) worden gegenereerd:

- makecert.exe -sv Sharp Shuttle_privatekey.pvk -n "CN=Sharp Shuttle" Sharp Shuttle_certificate.cer

Bij het signeren wordt de gebruiker gevraagd om de privesleutel te beveiligen met een wachtwoord. Dit wachtwoord heb je vervolgens elke keer nodig als je een binary wilt signeren. Het wachtwoord dat voor het certifiaat van Sharp Shuttle gebruikt word, is te vinden in de Signing folder op de CD.

Omdat het signeren van binaries tegenwoordig niet meer met standaard certificaten gaat, maar met speciale certificaten, zullen we een aantal transformatiestappen uitvoeren om vervolgens tot een bestand te komen wat we de signeer container noemen. Deze container bevat de gecodeerde privésleutel en een certificaat met de bijbehorende publieke sleutel.

De volgendee stap is het het omzetten van het certificaat (.cer) naar een codesigning certificaat (.spc). Merk op dat bij deze stap geen privé informatie wordt gebruikt, er zal dan ook niet om een wachtwoord worden gevraagd. Het volgende commando maakt van het desbetreffende certificaat een codesigning certificaat:

- cert2spc.exe Sharp Shuttle_certificate.cer Sharp Shuttle_publishcertificate.spc

Nu rest ons nog de stap om de privésleutel en het codesigning certificaat samen te voegen in een signeer container:

- pvk2pfx.exe -pvk Sharp Shuttle_privatekey.pvk -pi <wachtwoord hier> -spc Sharp Shuttle_publishcertificate.spc -pfx Sharp Shuttle_signcontainer.pvf -po <wachtwoord hier>

Dit commando gebruikt privé informatie en dus zal er weer gevraagd worden om een wachtwoord. Tenslotte kan met met de po argument een nieuwe wachtwoord ingesteld worden om de container te beveiligen.

Nu is er een niet vertrouwd certificaat waarmee we binaries kunnen signeren. Het signeren kan men handmatig doen met "signtool.exe". Door het volgende commando uit te voeren zullen we een fictieve binary helloworld.exe signeren:

- signtool.exe sign /f Sharp Shuttle_signcontainer.pvf /p "wachtwoord hier" /t “http://timestamp.comodoca.com/authenticode” /v "helloworld.exe"

Om een goede flow in het ontwikkelproces te hebben kan automatische signing gebruik worden. De signing directory bevat een tooltje sign.cmd die met de juiste parameters een binary gaat signen met het certificaat. Hieronder zal een samenvatting gegeven worden van de vereisten om te signeren:

- We willen niet zomaar signeren, alleen binaries die aangemerkt zijn als Release. (Optie in Visual Studio). De eerste parameter geeft aan wat voor een uitvoer het is. Signing gebeurd alleen als dit gelijk is aan Release
- De tweede parameter geeft aan welke bestand er gesigneerd moet worden. Deze parameter is het pad naar de binary.

Vervolgens moeten we er ook voor zorgen dat het tooltje zelf goed is ingesteld. De volgende instellingen in sign.cmd moeten goedgezet worden.

- %timestampurl% - verwijst naar een timestamp url zodat er een geldige timestamp in de binary komt.
- %signtoolpath% - deze variabele moet het volledige pad naar signtool.exe bevatten (zonder \signtool.exe).
- %pfxcontainer% - deze variabele moet het volledige pad naar de pfx signeer container bevatten.
- %private_key_password% - Deze variabele staat niet ingesteld in sign.cmd deze rede hiervoor is dat er voorzichtig met deze informatie moet worden omgegaan. om deze variabele te gebruiken moet men naar de systeem eigenschappen gaan en daar de varibelen aanmaken (Computer->Eigenschappen->Geavanceerde Systeem instellingen->Geavanceerd->Omgevings variabelen en vervolgens een nieuwe Systeem variabele aanmaken (private_key_password, wachtwoord))

Als laatste moet in elk project een post-build event worden gemaakt. Deze is te vinden in de properties van een project, onder Build Events. Onder post-build event command line moet staan:

- CD "$(SolutionDir)..\Signing"
- "sign.cmd" $(ConfigurationName) "$(TargetPath)"

Als al deze instellingen goed staan zal Visual Studio na het build proces (postbuildevent) alle binaries signeren.

9.8.2 Installers

Voor gemakkelijke lancering van nieuwe versies, is een installer het meest geschikt. Op die manier is met een verwisselbare schijf (bijv. een CD-rom) de applicatie eenvoudig te plaatsen op een willekeurige computer.

Om een installer te genereren biedt VisualStudio de mogelijkheid om een setup-project aan de solution toe te voegen. Ook bij de geleverde ontwikkelingsbestanden zit zo een project, genaamd “Sharp Shuttle”. In dit projekt zijn alle instellingen gemaakt voor het genereren van een gebruiksvriendelijke en robuuste installer. Aangezien de meeste instellingen zeer voor de hand liggend zijn, zal hier niet verder op ingegaan worden.

10 Mogelijke verbeteringen

Het Sharp Shuttle programma bevat een redelijke hoeveelheid functionaliteiten. Er zijn echter nog genoeg mogelijkheden om het programma te verbeteren. Om dit aan te geven hebben wij een aantal mogelijke verbeteringen bedacht waarmee Helios verder mee kan gaan. Dit zijn natuurlijk niet alle mogelijkheden, maar ze geven wel een goed beeld van wat er nog met het programma kan gebeuren.

- **Streepjescodes:**

  Het opzoeken van een wedstrijd in het programma om de bijbehorende score in te voeren kost wat tijd. Door een scanner op de PC aan te sluiten en op elk wedstrijdbriefje een unieke streepjescode te printen is het mogelijk om bij het scannen van een wedstrijdbriefje het

bijbehorende invulscherm in het programma te openen. Zo gaat het invoeren van scores nog makkelijker, zeker op toernooien met een groot aantal wedstrijden.

- **Wijzigen voor andere sporten:**

  De manier waarop de ranglijst bepaald word is vrij makkelijk aan te passen. Het aantal sets in een wedstrijd is gelimiteerd op 3 vanwege de ruimte op de wedstrijdbriefjes. Als deze worden weggelaten of aangepast is het mogelijk om meer dan 3 sets per wedstrijd te spelen. Het programma is dan ook bruikbaar om Zwitserse ladder toernooien te organiseren voor andere sporten, zolang deze maar onder te verdelen zijn in 3 scoreniveaus (wedstrijd, set, punt/rally).

- **Display toevoegen:**

  Bij de oorspronkelijke opdracht was sprake van een scherm waarop de deelnemers van het toernooi konden zien welke wedstrijden bezig waren en welke wedstrijden daarna gespeeld gaan worden. Wegens tijdsgebrek is dit niet in het uiteindelijke systeem gekomen. Om dit alsnog toe te voegen moeten de aankomende wedstrijden een volgorde krijgen vanuit het Veld informatie scherm. Deze volgorde moet in de database bewaard worden. Behalve deze aanpassing is het toevoegen van een display verder een makkelijke taak. Op deze display zouden ook nog foto's van het toernooi getoond kunnen worden.

- **Inschrijvingsscherm:**

  Aan het begin van een toernooi zonder inschrijvingen moeten alle spelers door de beheerder ingeschreven worden. Een idee is om een aparte gebruiker en een apart scherm te maken voor het inschrijven van een speler. Deze kan dan in de week voor het toernooi op de vereniging worden neergezet, waarna mensen zichzelf kunnen inschrijven. De beheerder hoeft de inschrijvingen daarna alleen uit te zoeken.

- **Wachtwoorden:**

  De keuze van het soort gebruiker waarmee ingelogd word bepaalt welke rechten er zijn in het programma. Dit om te voorkomen dat mensen zonder de juiste kennis of toegang geen verkeerde wijzigingen kunnen doen. Op het moment zijn de gebruikers echter niet met wachtwoord beveiligd, waardoor iedereen als beheerder in zou kunnen loggen. Een mogelijke oplossing is op de gebruikers met rechten een wachtwoord in te stellen.

- **Statistieken:**

  Cijfers zijn altijd leuk om te hebben, en daarom kunnen statistieken van het toernooi bijgehouden worden. Denk aan aantal wedstrijden en gemiddelde score. Door ook de lengte van elke wedstrijd bij te houden, zou een voorspelling kunnen worden gemaakt van de looptijd van aankomende rondes.

- **Zaalsplitsing:**

  Het toernooi zou over verschillende zalen gespeeld kunnen worden. Het is dan handig om per zaal een laptop te hebben voor het beheren. Dit kan op zich al door per zaal een aantal verschillende poules te spelen, maar netter is het om het idee van zalen in het programma op te nemen.