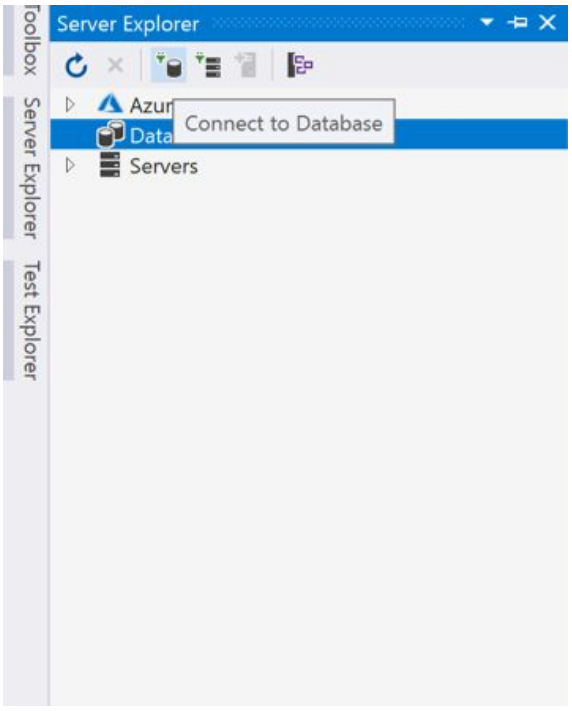
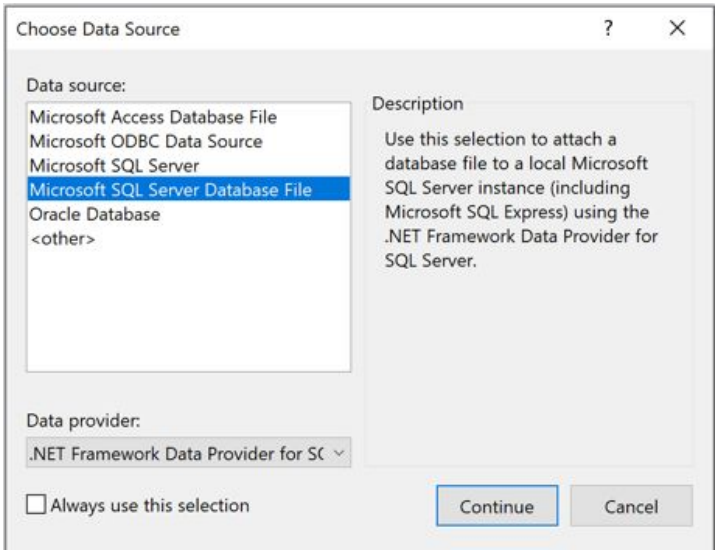
SWK/WEA Projekt-Dokumentation

Installationsanleitung

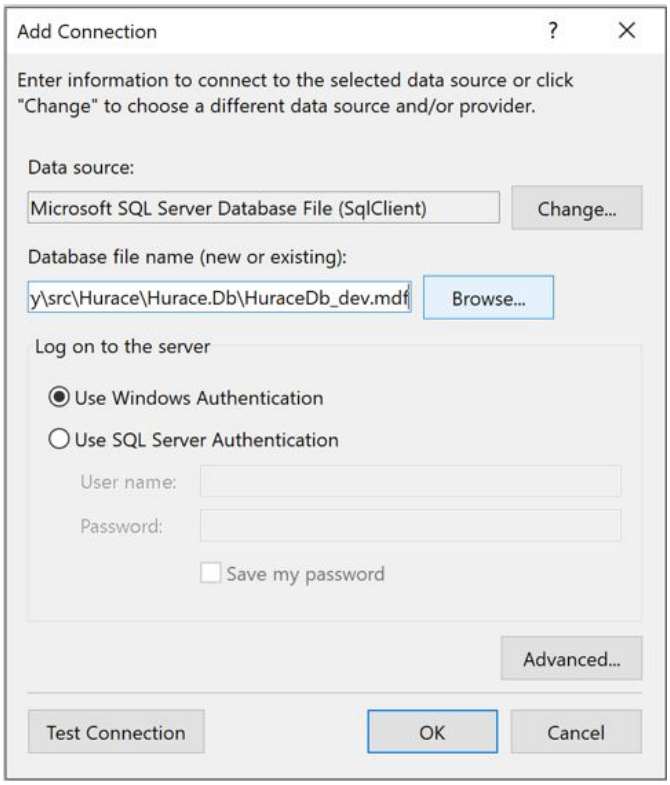
Zuerst im Server-Explorer *Connect to Database* drücken.



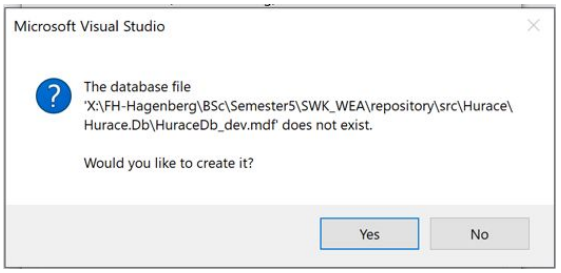
Microsoft SQL Server Database File auswählen und *Continue* drücken.



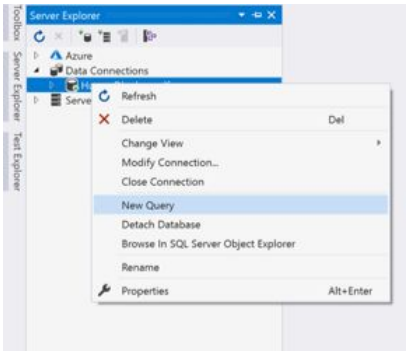
Im nächsten Schritt wird eine neue LocalDB-Instanz angelegt, geben Sie also bitte den Pfad an, wo die neue Datenbank gespeichert werden soll. (Wir haben diese üblicherweise unter *Hurace.Db* abgelegt.
Danach auf *OK* drücken.



Visual Studio erkennt, dass die Datenbank noch nicht existiert, drücken Sie also bitte *Yes* um die Datenbank zu ersellen.

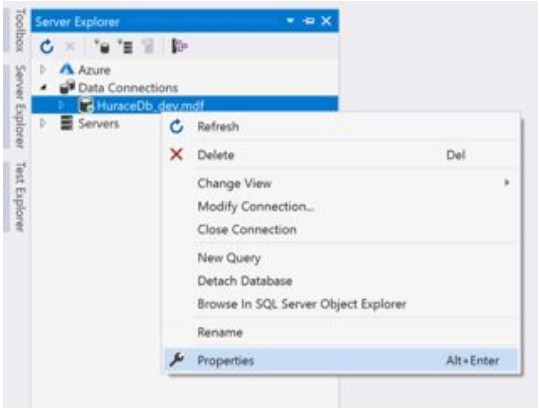


Dann im Server Explorer rechsklick auf die neu angelegte Datenbank-Instanz und *New Query* drücken.

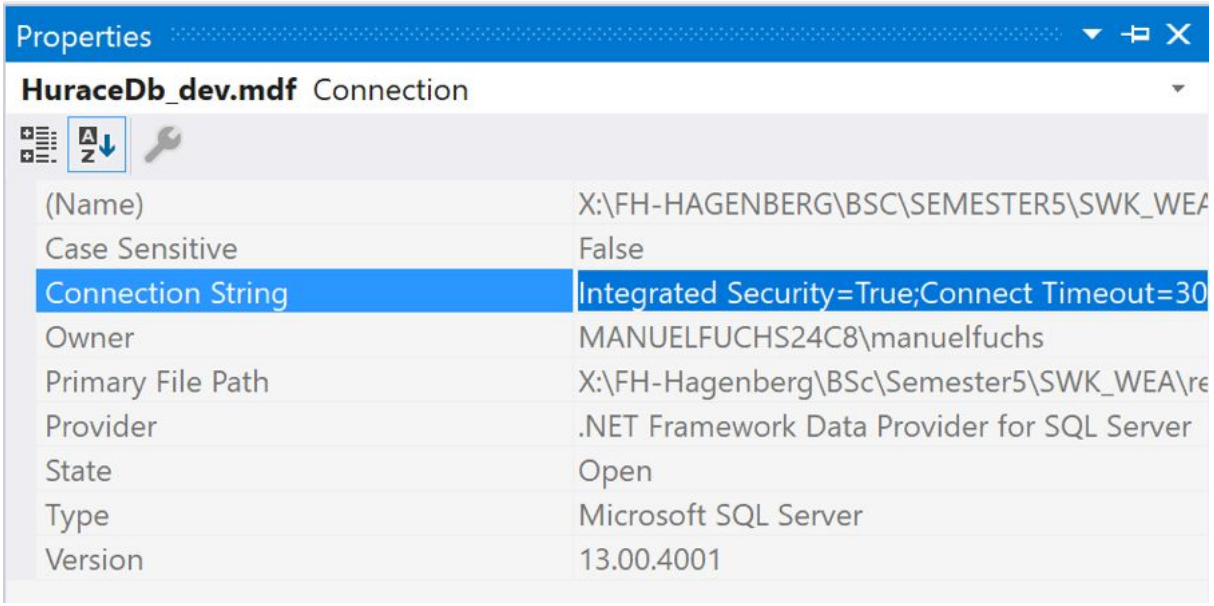


Dann kopieren Sie bitte den Inhalt des Skripts *create_script.sql* in das geöffnete Fenster und führen es aus. Danach wiederholen Sie dasselbe mit dem *insert_script.sql* Skript.

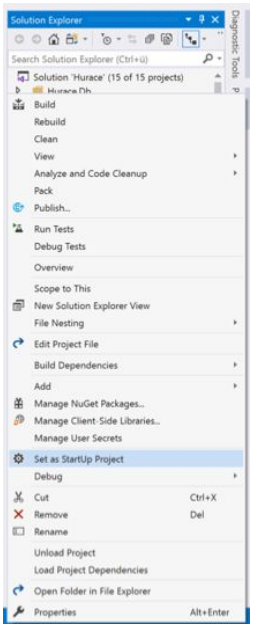
Danach noch einmal rechtsklick auf die nun befüllte Datenbank und diesmal *Properties* auswählen.



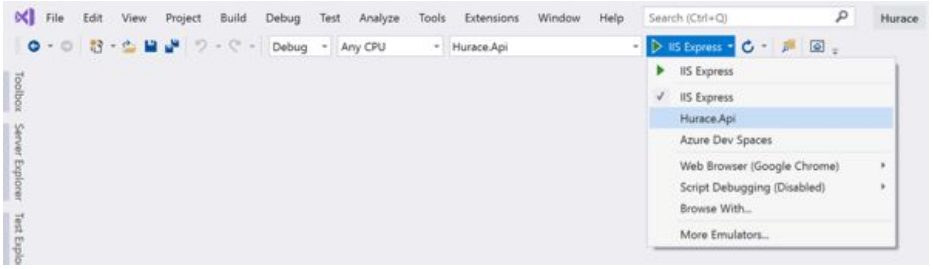
Das Property-Fenster öffnet sich, nun bitte Connection String kopieren und in den JSON-Dateien unter *HuraceDbConnectionString.ConnectionString* eintragen in den Dateien *Hurace.Api/AppSettings.json* und in *Hurace.RaceControl/AppSettings.json*.
Damit alle Integration-Tests erfolgreich ausgeführt werden können, wird die Datenbank im Zustand nach Ausführung des *insert_script.sql* erwartet, deshalb ist es ratsam eine 2.Datenbank mit den oben beschriebenen Schritten erneut anzulegen und mit Skripts zu befüllen. Der Connection-String dieser 2.Datenbank muss dann unter Hurace.Core.Tests/AppSettings.json eintragen werden.



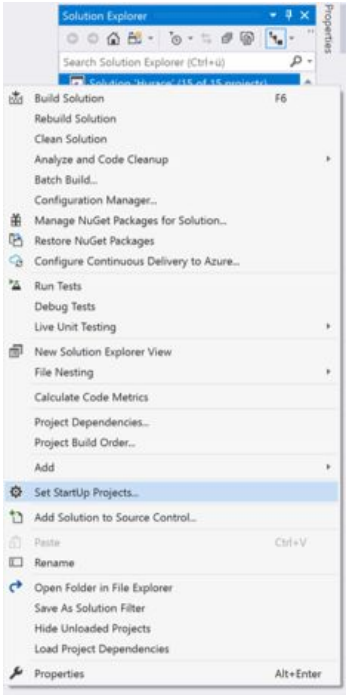
Dann mit rechtsklick auf Hurace.Api dieses Projekt als Startprojekt auswählen.



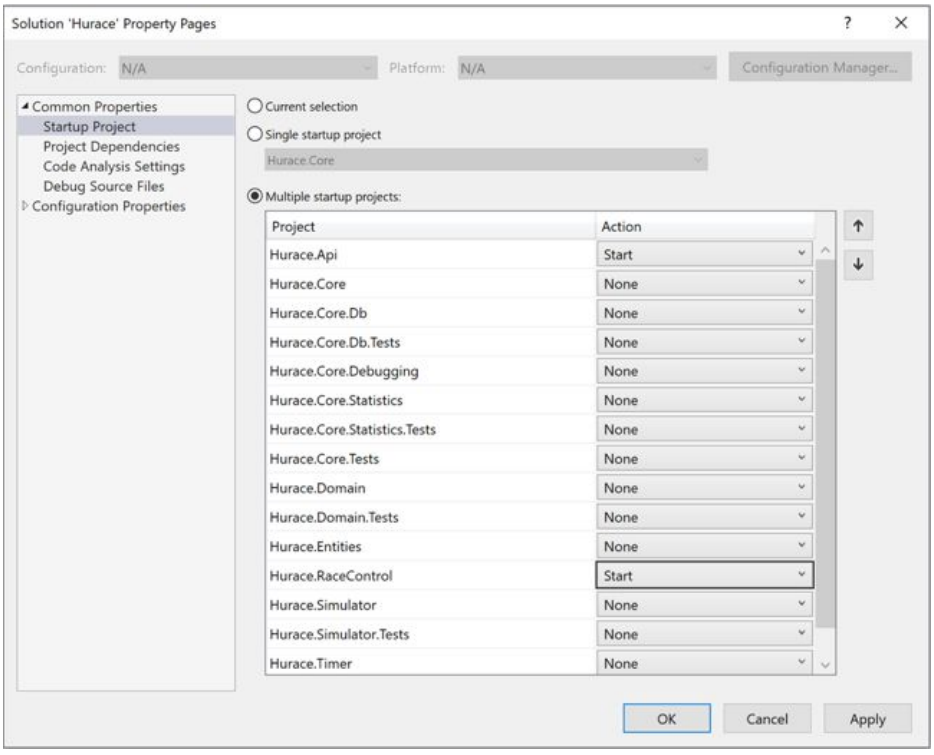
Dann bei Hurace.Api als Startmethode wählen, um Kestrel statt IIS Express als lokalen Entwicklungsserver zu verwenden.



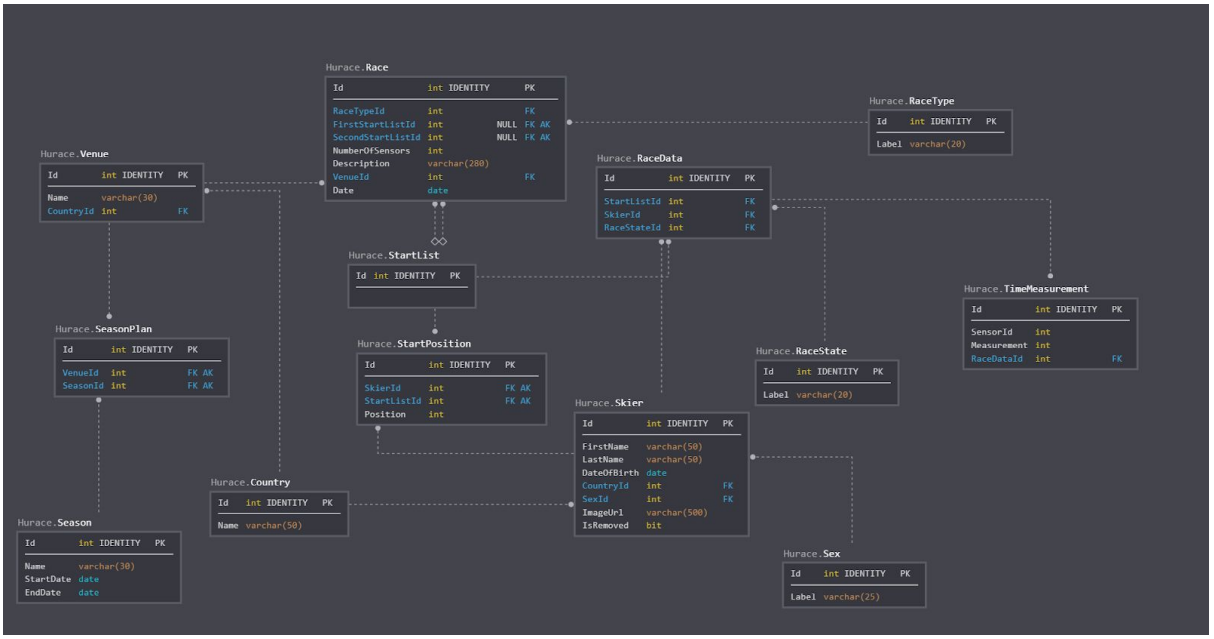
Danach mit rechtsklick auf die Solution die eigentlichen Startprojekte auswählen.



Dann *Multiple startup projects* auswählen und *Hurace.Api* und *Hurace.RaceControl* als Startprojekte wählen und mit *OK* bestätigen.



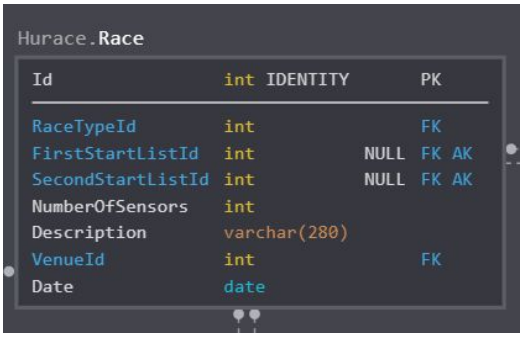
Datenbankmodell



Country

Enumeration für Länderkürzel (z.B.: “AUT”, “GER”, ...)

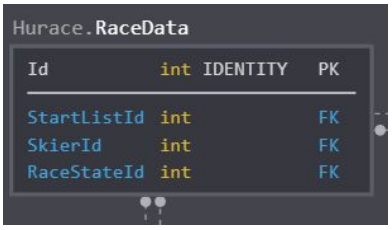
Race



In Race werden alle Relevanten Daten zu einem Rennen gespeichert.

RaceTypeId: Referenz auf den Renntyp
FirstStartListId: Referenz auf Startliste für ersten Durchgang
FirstStartListId: Referenz auf Startliste für zweiten Durchgang
NumberOfSensors: Anzahl der im Rennen verwendeten Sensoren
Description: Rennbeschreibung
VenueId: Referenz auf einen Veranstaltungsort
Date: Veranstaltungsdatum

RaceData

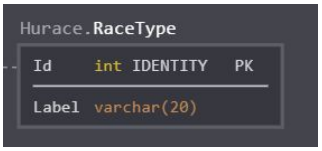


Mit RaceData wird gespeichert, welchen Zustand der Antritt eines Schifahrers bei einem Rennen hat.

RaceState

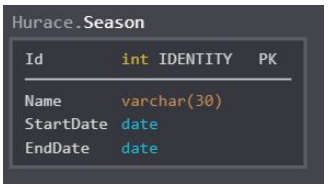
Enumeration für Rennstati (z.B.: Disqualifiziert, Abgeschlossen,...)

RaceType



Enumeration für Renntypen (z.B.: Slalom, Riesentorlauf,...).

Season



In der Seasontabelle wird Name und Dauer gespeichert, Maßnahmen um zu verhindern dass Saisonen gleichzeitig auftreten können wurden nicht getroffen da dies nicht gefordert wurde. In den Testdaten wurden 2 ganzjährliche Saisonen abgebildet, eine in 2017 und die andere in 2018.

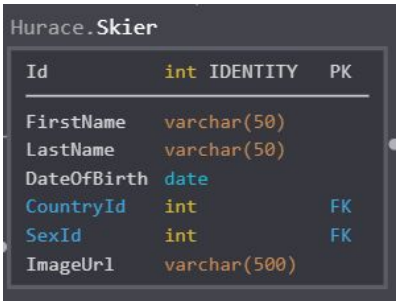
SeasonPlan

SeasonPlan bildet die Beziehung zwischen Saison (*Season*) und Austragungsort (*Venue*) ab.

Sex

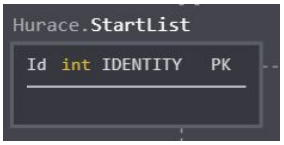
Das Geschlecht einer Person als Enumeration abzubilden, wurde entschieden um auf noch unbekannte und mögliche Änderungen hinsichtlich der gesellschaftlichen Einstellung zum sexuellen Geschlecht, vorbereitet zu sein.

Skier



Enthält alle persönlichen Daten zu den Schifahrern.

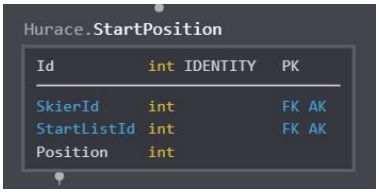
StartList



Fortlaufende Nummer für die Startlisten. Der Grund, warum diese Entität überhaupt existiert, ist die Angabe. Dort ist gefordert, dass eine Entität mit Namen StartList gespeichert werden soll.
Unserer Meinung nach ist es semantisch-gesehen falsch, wenn man die Position direkt im StartList-Tabelle speichert, da ja der Name der Tabelle suggeriert, dass ein Eintrag darin eine eigenständige Startliste darstellt.
In unserem Datenmodell setzt sich somit eine einzelne Startliste nicht durch viele einzelne StartList-Datensätzen zusammen, sondern bildet nur eine einzige StartListe ab.
Die Information über einzelne Startnummern wurde in die StartPosition-Tabelle verschoben.

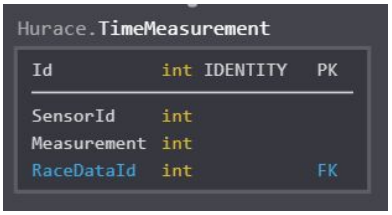
Andererseits lässt sich auch darüber streiten, ob eine Tabelle mit synthetischem Primärschlüssel und keinen weiteren Spalten überhaupt Sinn macht.

StartPosition



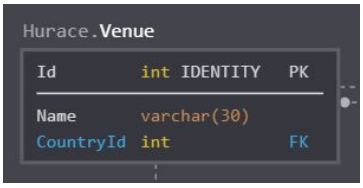
Bildet die Beziehung zwischen Schifahrer und Startliste ab, also an welcher Position ein Schifahrer in einer Startliste aufgelistet ist.

TimeMeasurement



Speichert alle gültigen Zeitmessungen eines Renn-Durchlaufs.

Venue

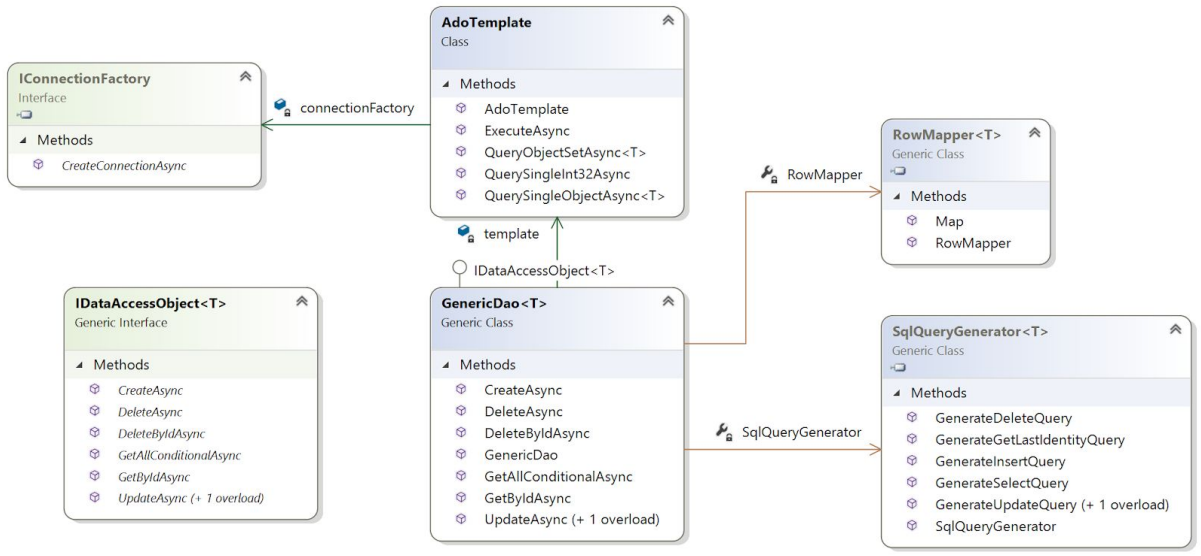


Austragungsorte der Rennen.

Testdatenset

52	Countries
74	Races
7414	RaceData
5	RaceStates
2	RaceTypes
148	StartLists
7414	StartPositions
2	Seasons
62	SeasonPlans
521	Skiers
2	Sexes
37544	TimeMeasurments
31	Venues

Aufbau der Datenzugriffsschicht



SqlQueryGenerator

Der SqlQueryGenerator bietet Methoden zur Generierung simpler SQL-Querys abhängig vom Typ der beim Erstellen übergeben wird.
Um SQL-Injection zu verhindern, werden Parameter nicht direkt in die Query eingebettet, sondern über das DbCommand sanitized.

RowMapper

Der RowMapper ist generisch implementiert und kann einzelne Zeilen aus der Datenbank auf Objekte eines Typs mappen, sofern diese kompatibel zueinander sind.

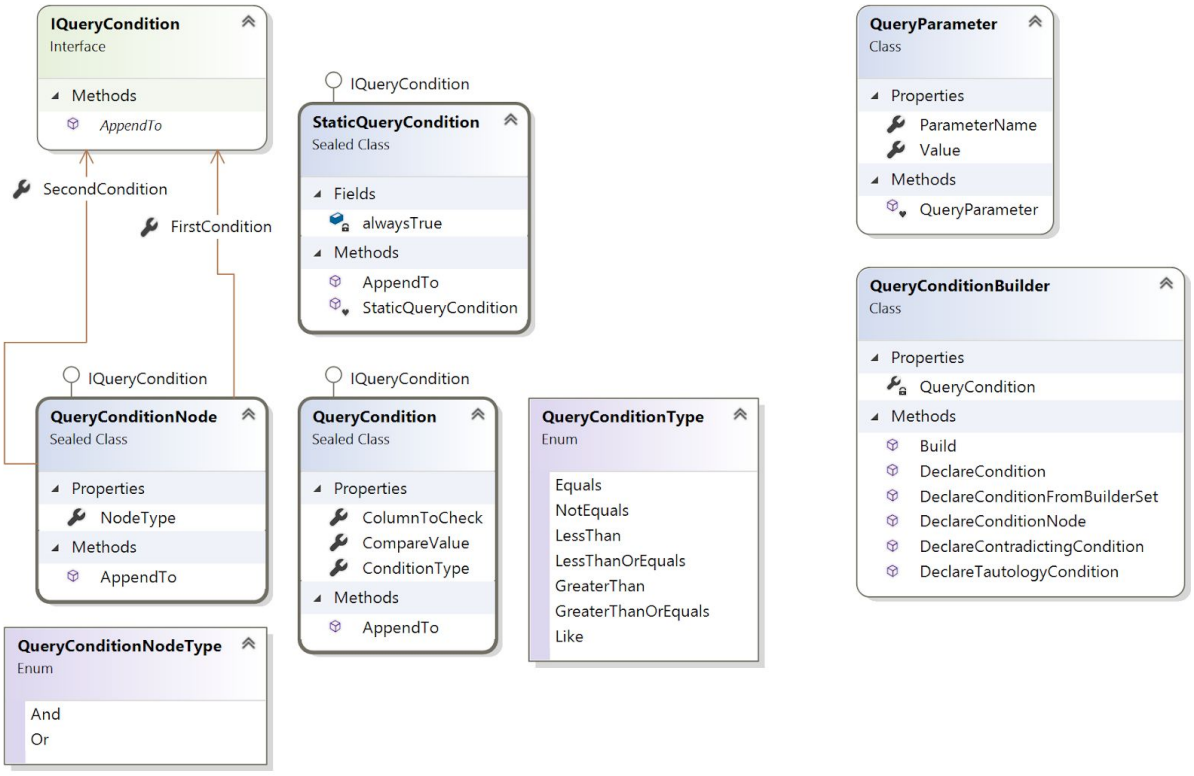
AdoTemplate

Der beim Ausführen von SQL-Abfragen entstehende Boilerplate-Code wird mittels mehrerer Template-Methods im Falle von ADO.NET mit dem AdoTemplate gekapselt.

GenericDao

Das GenericDao verbindet die Funktionalität von *SqlQueryGenerator* und *RowMapper* und bietet dem Verwender die Möglichkeit, generierte Abfragen auf die Datenbank auszuführen, die dann automatisiert auf Entitäten gemapped werden.
Diese Klasse ist nicht abstrakt, da die angebotenen Queries so generisch gehalten sind dass keine Anpassung auf spezielle Domaintypen notwendig ist. Queries können dann von außerhalb über *QueryConditions* optimiert werden (zum Beispiel ein Update auf mehrere Zeilen, bei denen eine *QueryCondition* zutrifft).

Query-Conditions



Was ist eine QueryCondition? Eine QueryCondition ist eine Abstraktion der WHERE-Klausel bei einer SQL-Abfrage.
Wozu werden QueryConditions benötigt? Um der Business-Logik, die Implementierungen von *IDataAccessObject* verwendet, zu ermöglichen, die Abfragen zu optimieren. Zum Beispiel kann eine GetAll-Abfrage somit auf wenige Einträge begrenzt werden.

Wie funktioniert die Abstrahierung? Wenn man die WHERE-Klausel einer SQL-Abfrage betrachtet, fällt einem auf, dass diese nur aus einzelnen Bedingungen besteht die durch logische Operatoren verknüpft werden. Diese Erkenntnis hat folgende Abbildung motiviert. Einzelne Bedingungen sind vom Typ *QueryCondition*, logische Verknüpfungen die zwei zwei einzelne Bedingungen miteinander verbinden sind vom Typ *QueryConditionNode*.
Auf der anderen Seite gibt es dann noch *StaticQueryCondition*s, die immer wahr oder falsch ergeben. Dies mag zwar komisch wirken, aber es gab Fälle in der Business-Logik, da war so eine Implementierung hilfreich.
Diese Architektur ist durch das *Composite-Pattern* motiviert, wobei die klassisch abstrakte Basisklasse hier ein Interface ist.

Wenn diese QueryCondition nun verwendet werden möchte im SqlQueryGenerator, muss dieser nur AppendTo aufrufen. Diese Methode nimmt einen StringBuilder (wird zum generieren der SQL-Query verwendet) und eine Liste von QueryParametern als Parameter. Dies bewirkt, dass die einzelnen Vergleiche die über die QueryConditions abgebildet sind, in den übergebenen StringBuilder gegeben werden. In dieser Query stehen aber nicht direkt

die in den QueryConditions angegebenen Werte, da dies zu SQL-Injection führen würde. Stattdessen werden die übergebenen Werte in einen QueryParameter eingebettet und diese werden dann in die übergebene Liste eingefügt.

Die simple WHERE-Klausel “WHERE Name = ‘Austria’ AND Date = ‘<aktuelles Datum>’” kann nun also so nachgebildet werden.

```
var condition = new QueryConditionNode
{
    FirstCondition = new QueryCondition
    {
        ColumnToCheck = nameof(Domain.Country.Name),
        CompareValue = "Austria",
        ConditionType = QueryConditionType.Equals
    },
    NodeType = QueryConditionNodeType.And,
    SecondCondition = new QueryCondition
    {
        ColumnToCheck = nameof(Domain.Race.Date),
        CompareValue = DateTime.Now.Date,
        ConditionType = QueryConditionType.LessThan
    }
};
```

Das Problem bei der Initialisierung von QueryConditions auf diesem Wege ist jedoch, dass die Überprüfung einer korrekten Initialisierung der QueryConditions erst beim Verwenden bei AppendTo passieren kann.
Um diesen Umstand zu verbessern, wurde ein QueryConditionBuilder implementiert, der beim Bauen einer QueryCondition bereits Überprüfungen durchführen kann.
Ein weiterer Vorteil ist, dass durch diesen Schritt die Klassen QueryConditionNode, QueryCondition und StaticQueryCondition nur interne Sichtbarkeit haben müssen und somit ein Verwender außerhalb des Hurace.Core.Db-Projektes nun somit gezwungen ist, den QueryConditionBuilder zu verwenden.

Die zuvor beschriebene QueryCondition kann nun so über den Builder initialisiert werden:

```
var builtCondition = new QueryConditionBuilder()
    .DeclareConditionNode(
        QueryConditionNodeType.And,
        () => new QueryConditionBuilder()
            .DeclareCondition(nameof(Domain.Country.Name), QueryConditionType.Equals, "Austria"),
        () => new QueryConditionBuilder()
            .DeclareCondition(nameof(Domain.Race.Date), QueryConditionType.LessThan, DateTime.Now.Date))
    .Build();
```

Des weiteren wird dann zusätzlich die Extension-Method AddQueryParameter für Variablen vom Typ *IList<QueryParameter>* benötigt, da es sein kann, dass in einer UPDATE-Query bereits zuvor andere QueryParameter hinzugefügt wurden, die die aktualisierten Werte beinhalten. Diese Situation kann zu Namenskonflikten führen, da jeder QueryParameter ja eindeutig benannt werden muss.
Deshalb sollte beim hinzufügen von neuen QueryParametern immer diese Extension-Method verwendet werden, da diese die Funktionalität bietet um Namenskonflikte zu beheben.

Um große Kaskaden an QueryConditionNode Verkettungen zu vermeiden, hat der QueryConditionBuilder die Methode DeclareConditionFromBuilderSet. Dieser Methode kann eine Menge von QueryConditionBuildern übergeben werden und die Information, wie die resultierenden QueryConditions zu verknüpfen sind mit QueryConditionNodeTypes. Dies

funktioniert aber nur, wenn die Art der Verknüpfung über allen Builder-Instanzen homogen ist.
Ein Beispiel dazu aus der Businesslogik:

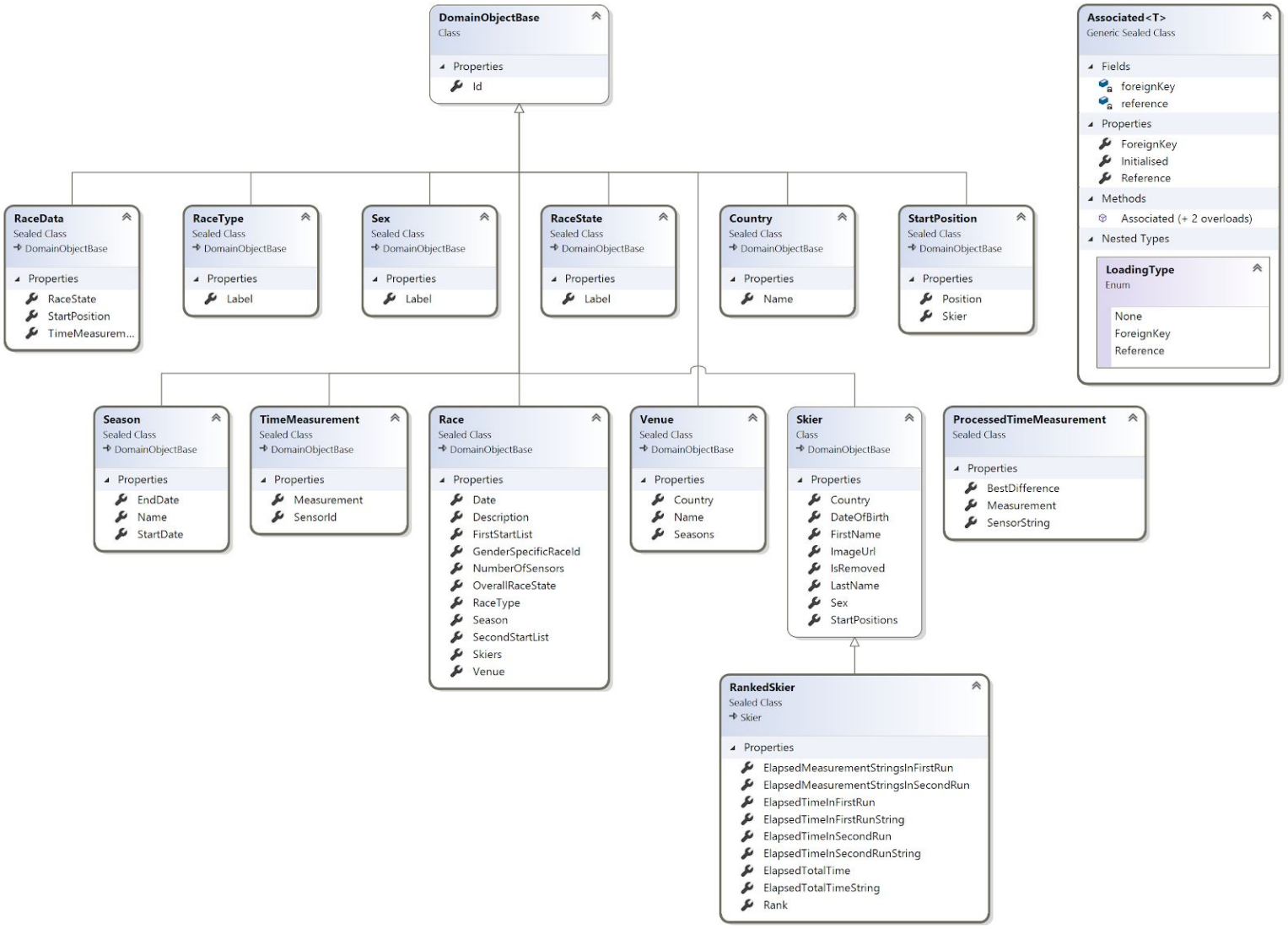
```
var seasonEntSet = await seasonDao.GetAllConditionalAsync().ConfigureAwait(false);
var seasonPlanEntSet = (await seasonPlanDao.GetAllConditionalAsync().ConfigureAwait(false))
    .Where(sp => seasonIdSet.Any(seasonId => seasonId == sp.SeasonId));

seasonConditionBuilder = !seasonPlanEntSet.Any()
    ? new QueryConditionBuilder()
        .DeclareContradictingCondition()
    : seasonConditionBuilder = new QueryConditionBuilder()
        .DeclareConditionFromBuilderSet(
            QueryConditionNodeType.Or,
            seasonPlanEntSet.Select(sp => new QueryConditionBuilder()
                .DeclareConditionNode(
                    QueryConditionNodeType.And,
                    () => new QueryConditionBuilder()
                        .DeclareCondition(nameof(Entities.Race.VenueId), QueryConditionType.Equals, sp.VenueId),
                    () => new QueryConditionBuilder()
                        .DeclareConditionNode(
                            QueryConditionNodeType.And,
                            () => new QueryConditionBuilder()
                                .DeclareCondition(
                                    nameof(Entities.Race.Date),
                                    QueryConditionType.GreaterThanOrEquals,
                                    seasonEntSet.First(s => s.Id == sp.SeasonId).StartDate),
                            () => new QueryConditionBuilder()
                                .DeclareCondition(
                                    nameof(Entities.Race.Date),
                                    QueryConditionType.LessThanOrEquals,
                                    seasonEntSet.First(s => s.Id == sp.SeasonId).EndDate)))));
```

Erklärung: Wenn die Menge seasonPlanEntSet leer ist, erzeuge eine QueryCondition die immer falsch ist. (StaticQueryCondition im Hintergrund verwendet)
Ansonsten erzeuge eine QueryCondition von einer Menge an Buildern, die mit einem logischen OR verknüpft werden sollen.
Die einzelnen Builder können nun über eine Linq-Abfrage erzeugt werden.

Das einzige was Design-mäßig noch zu Problemen bei der Verwendung führt, ist, dass der angegebene Spaltenname beim .DeclareCondition-call über nameof angegeben wird und dies ist nicht typesafe. Der Verwender kann hier leider irgendeine Klasse angeben. Es wäre besser gewesen hier das Property zu übergeben, und dann den Namen über Reflection in der Method auszulesen.

Domänenobjekte

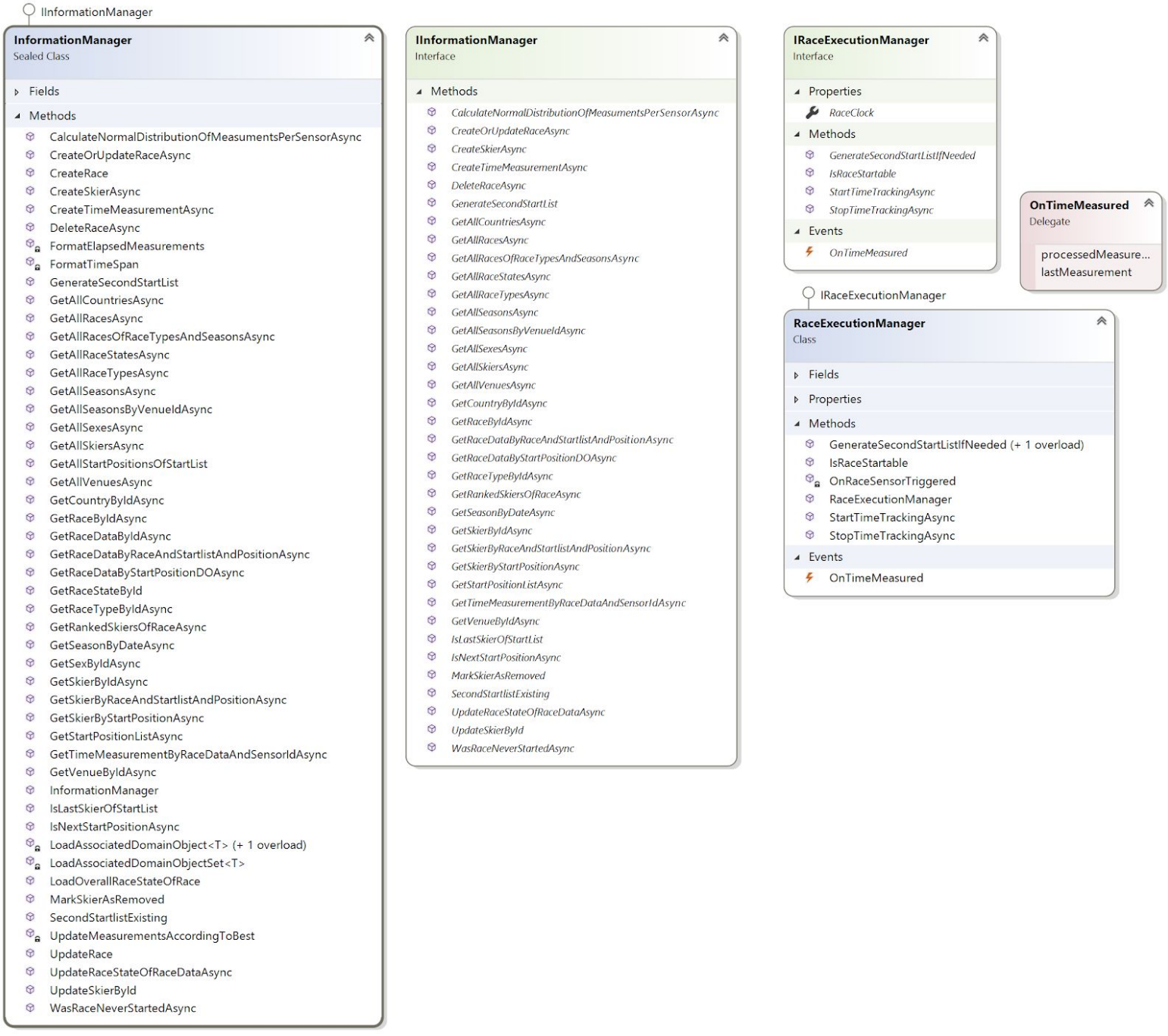


Beim Vergleich der Domänenobjekt-Hierarchie mit dem Datenbank-Modell fällt auf, dass hier zum Beispiel die StartListe nicht mehr vorhanden ist, beziehungsweise dass eine Associated-Klasse eingeführt wurde.

Warum existiert die StartListe nicht mehr? Die Startliste wird nur auf der Datenbank benötigt, um den Anforderungen der Angabe zu entsprechen.
In den Domänenobjekten wird diese Abstraktion nicht mehr benötigt und die Startliste kann vollkommen weggelassen werden. Eine Menge von StartPositions erfüllt genauso seinen Zweck.

Was ist die Associated-Klasse? Eine Associated-Klasse kapselt die Abhängigkeit einer Domänenklasse zu einer anderen Domänenklasse. Hinter einer Associated-Instanz kann dann entweder ein Fremdschlüssel oder die konkret geladene Referenz zu der assoziierten Domänenklasse stehen.
Wozu benötigt man eine solche Abstraktion? Dies kommt beim verwenden der Business-Logik zugute, indem der Aufrufer erst dann entscheiden muss, was er zu diesem Zeitpunkt wirklich benötigt, indem er an die Businesslogik eine Ausprägung der Enumeration *Associated.LoadingType* übergibt. Der *LoadingType* hätte nicht als nested-type implementiert werden müssen, jedoch macht dass den *LoadingType* dann weniger typsicher. In dieser Implementierung ist eine Ausprägung *Associated<Race>.LoadingType.ForeignKey* ungleich zu *Associated<Venue>.LoadingType.ForeignKey*. Dies macht natürlich die Implementierung der Business-Logik komplexer, jedoch die Verwendung dieser weniger Fehleranfällig.

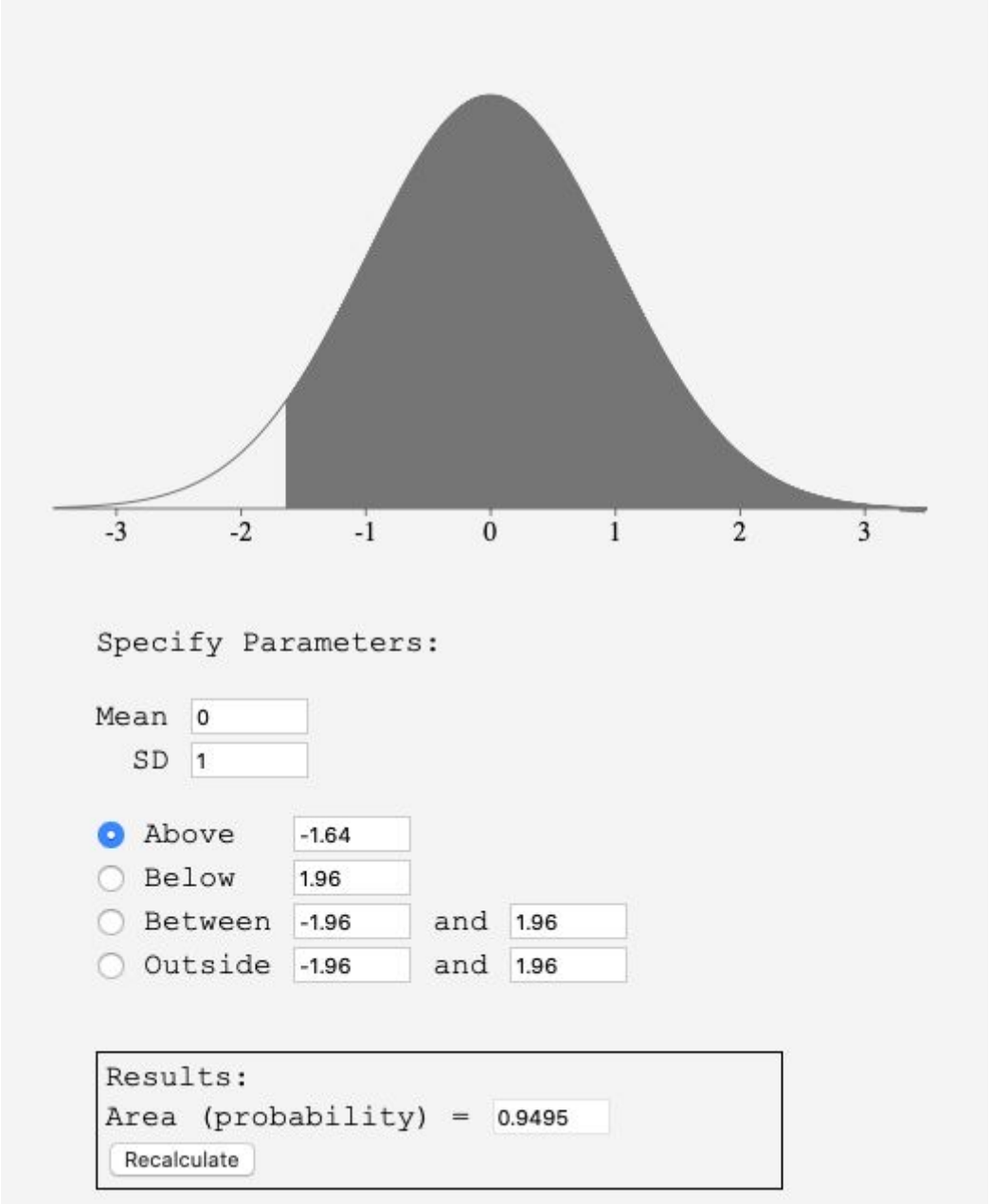
Architektur der BL

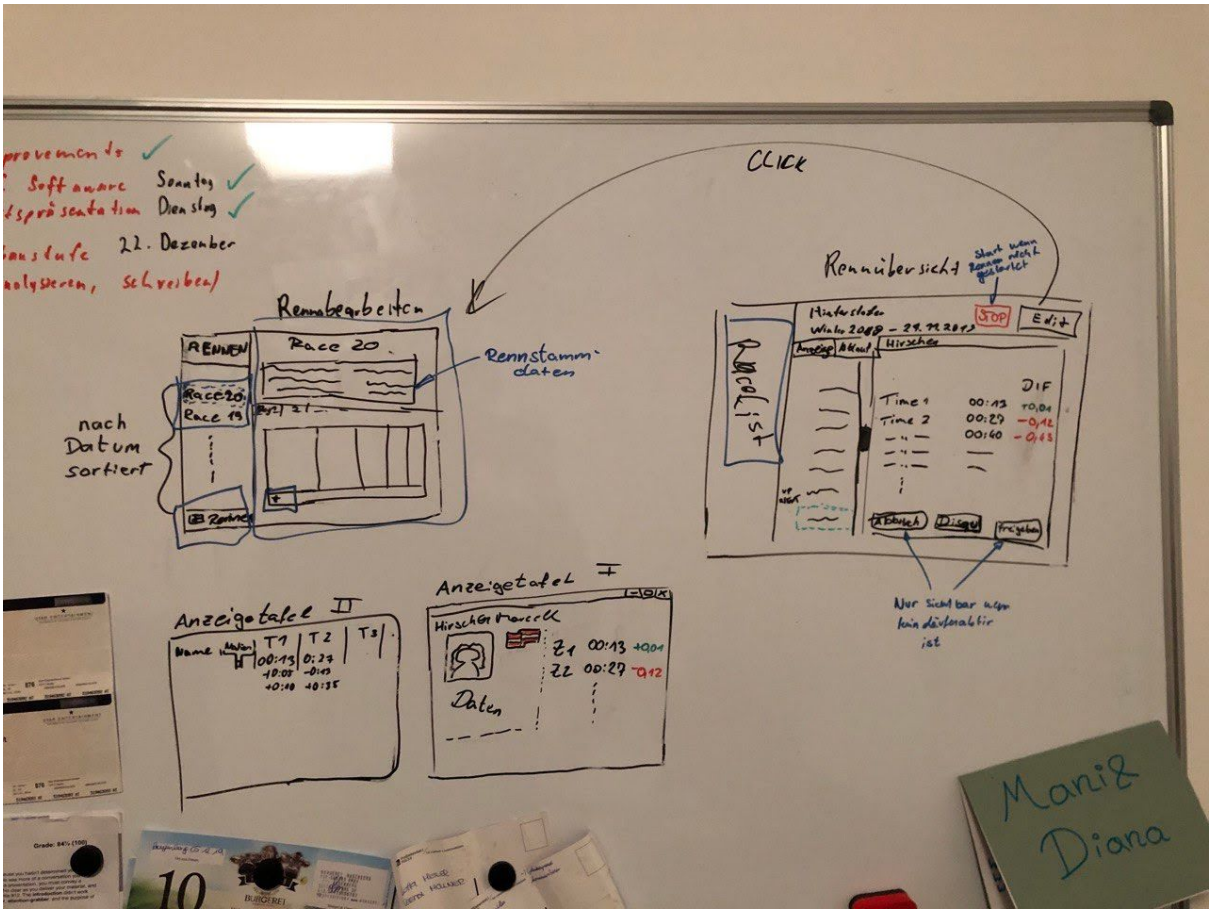


Grundsätzlich ist die BusinessLogik in 2 verschiedene Komponenten unterteilt. Einerseits in den *InformationManager* und andererseits den *RaceExecutionManager*.
Der InformationManager bietet alle Operationen, die benötigt werden um Informationen aller Domainobjekte zu verwalten, währenddessen der *RaceExecutionManager* die Funktionalität kapselt, Zeitmessungen über Hurace.Timer.IRaceClock zu nehmen.
Falls eine gültige Zeitmessung erkannt wurde, wird ein *OnTimeMeasured-Event* gefeuert um *Hurace.RaceControl* zu benachrichtigen, dass eine neue Zeit verfügbar ist.

Wie wird die Gültigkeit eines Sensorimpulses bestimmt?
Zu aller erst können Impulse gefiltert werden, zu deren Auslösezeitpunkt noch nicht der initiale Startsensor ausgelöst worden ist. Des weiteren wird der letzte Sensor immer als gültig erkannt, sofern der Startsensor zuvor ausgelöst worden ist.

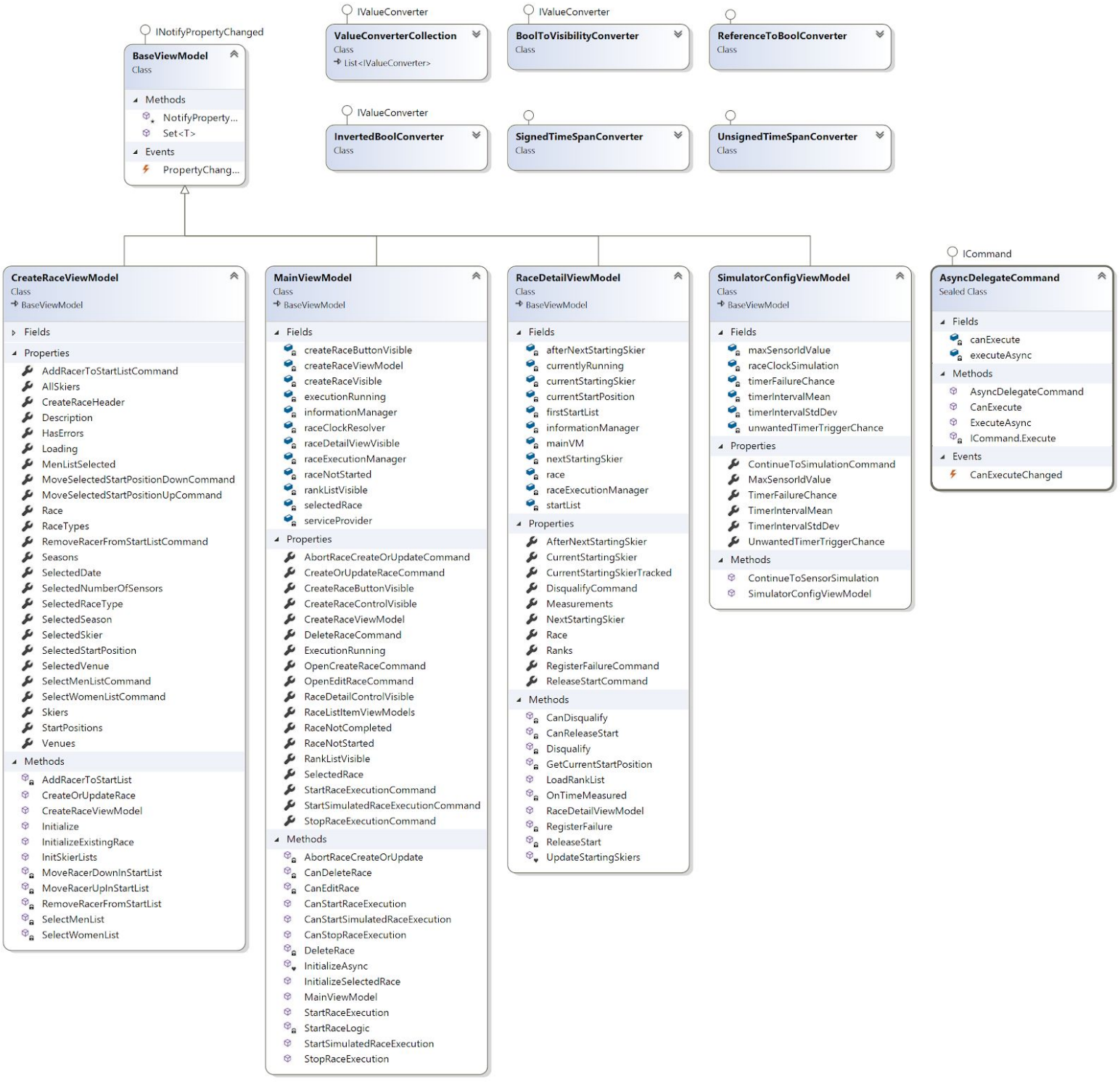
Dann müssen nur noch die dazwischen auftretenden Sensoren gefiltert werden.
Falls Rennen existieren mit dem selben Renntyp, die am selben Veranstaltungsort stattgefunden haben, werden maximal die letzten 15 genommen und von deren Zeitmessungen die Normalverteilung pro Sensor berechnet.
Beim auftreten eines Sensorimpulses wird dann nur noch ermittelt, ob sich die Zeitmessung in einem 95%-Intervall befindet. Aus Sicht der Standardnormalverteilung wäre das somit jeder Wert, der größer/gleich -1.64 ist.





Sketch der geplanten UI

Architektur von RaceControl

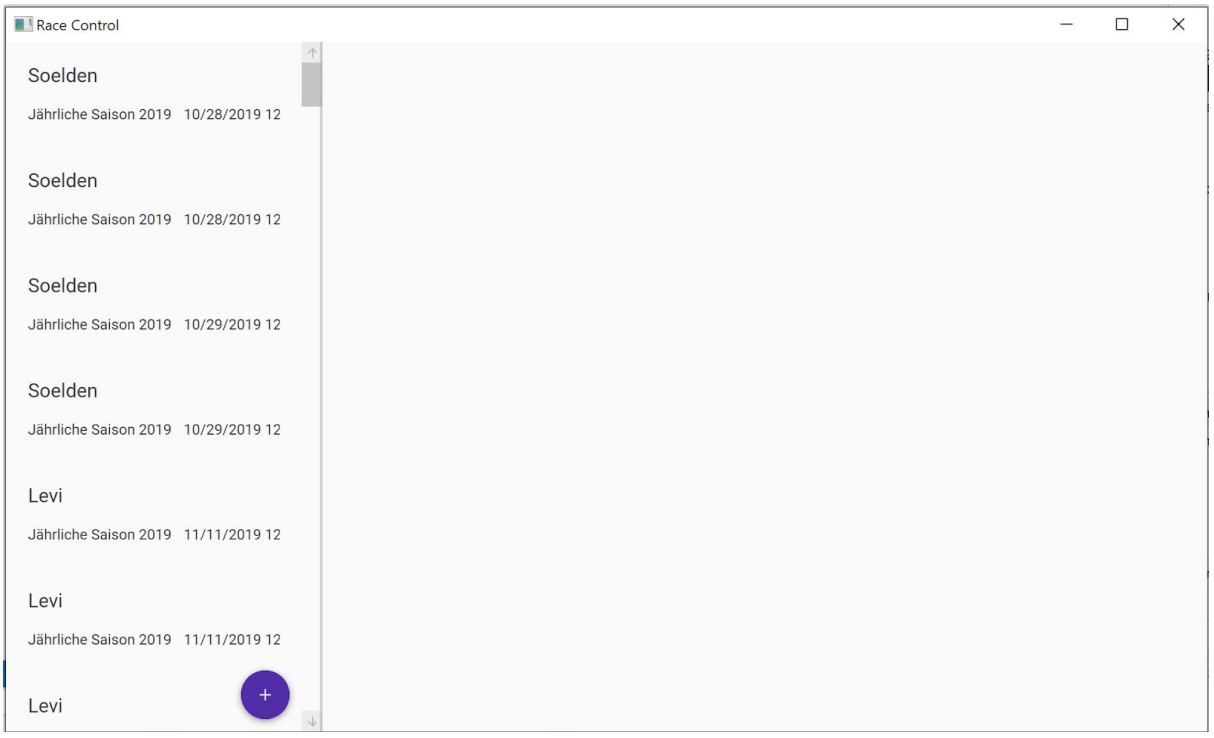


BaseViewModel: Das BaseViewModel implementiert Set, diese Methode Prüft bei aufruf ob sich ein Property geändert hat und benachrichtigt die Views welche sich auf die Properties binden.

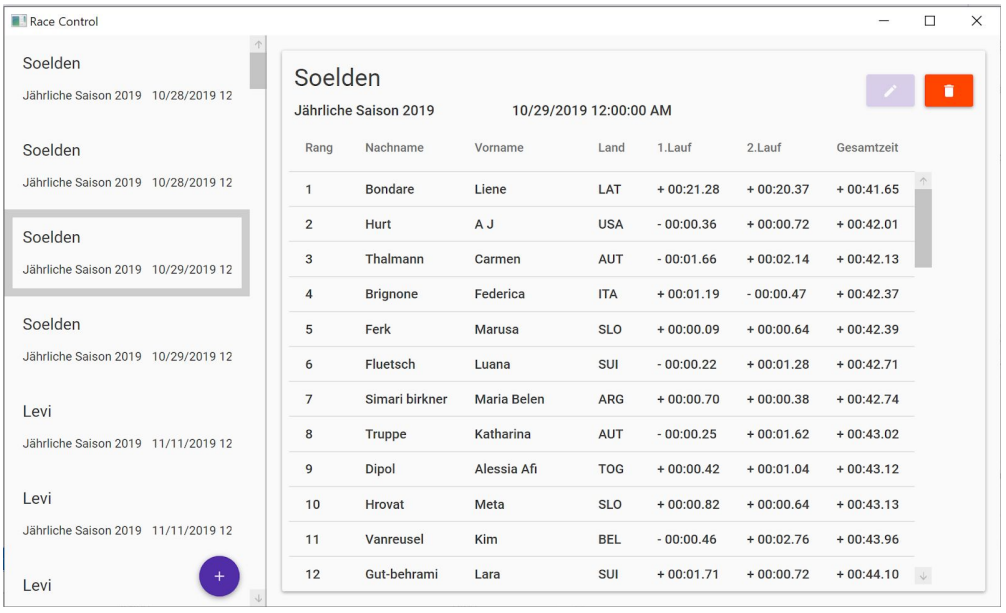
Das MainViewModel ist der Kern der WPF anwendung. Über Dependency injection erhält sie die BusinessLogic objekte und kann über diese Daten abfragen und schreiben. Beim initialisieren werden alle Rennen Asynchron geladen um sie danach im MainWindow anzuzeigen. Für jedes Rennen wird ein RaceDetailViewModel des entsprechenden Rennes angelegt. Im RaceDetailViewModel kann das Rennen gestartet und gestopt werden, hier wird alles Geladen was zum ausführen der Rennen notwendig ist.
Beim erstellen eines neuen Rennen wird im MainViewModel ein neues CreateViewModel angelegt, in diesem werden alle Daten die zum erstellen eines Rennens notwendig sind zum Anzeigen geladen. Über die UI kann der Anwender die erforderlichen Daten eintragen welche validiert werden. Über das MainViewModel wird signalisiert ob das Rennen nun angelegt werden soll, beim anlegen werden die Daten mittels BL in die Datenbank geschrieben. Das SimulatorConfigViewModel bietet eine Schnittstelle um die Konfiguration des Simulators zu ändern.

Die *ValueConverterCollection* erbt von der generischen BasisKlasse *List<IValueConverter>* und implementiert andererseits das Interface *IValueConverter*.
Dies ermöglicht den Verwender dieser Klasse, mehrere bereits bestehende ValueConverter miteinander zu verketten. Somit muss nicht 10 mal derselbe <Anything>ToVisibiltyConverter implementiert werden.

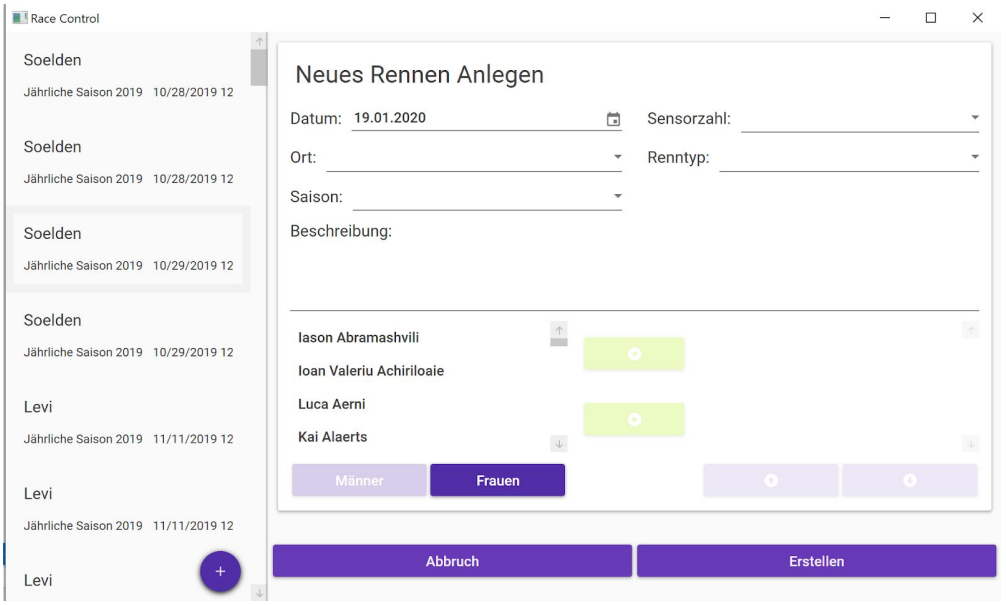
Beim Start der Anwendung landet der Benutzer in einer Ansicht, in der links alle Rennen aufgeführt sind.



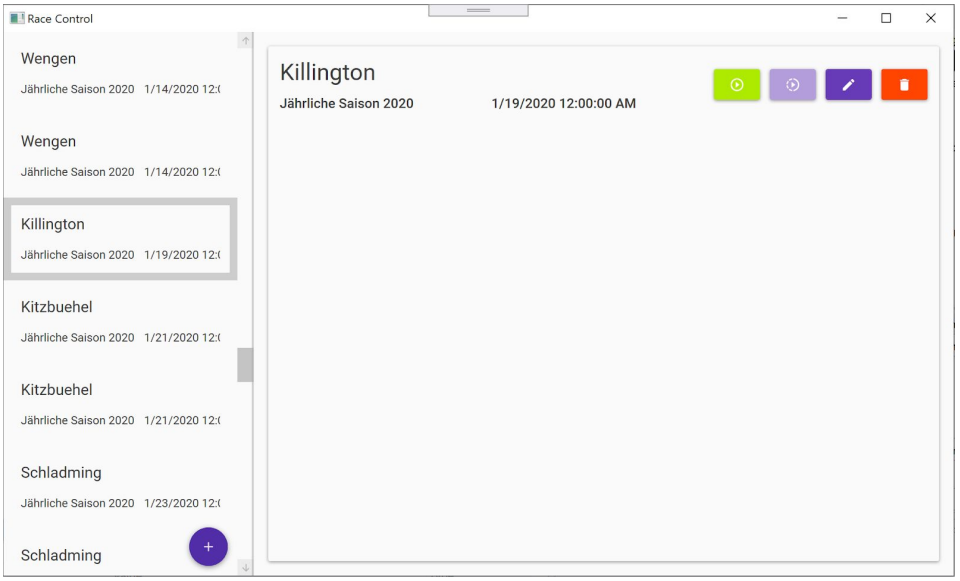
Wenn der Benutzer auf ein Rennen klickt und dieses abgeschlossen oder bereits gestartet wurde, wird eine Rangliste angezeigt. In diesem Fall werden die Buttons zum Starten der Rennausführung ausgeblendet, da das Rennen bereits abgeschlossen ist.
Rennen können dann auch über den Mistkübel-Button gelöscht werden.
Rennen können nur editiert werden, wenn diese noch nie gestartet wurden.



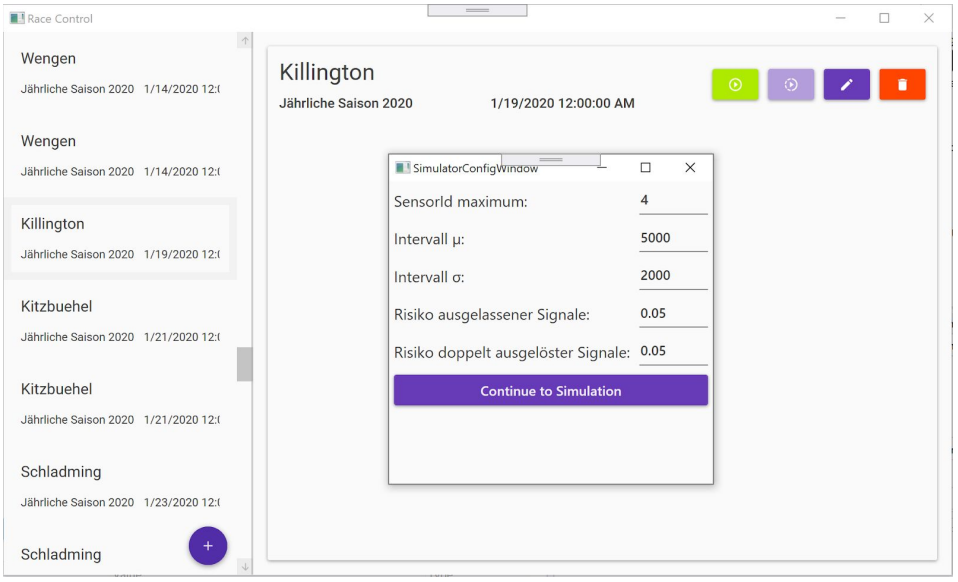
Rennen können dann über den Plus-Button erstellt werden.
Die Startliste kann unten über die Buttons verwaltet werden. Links ist eine Liste aller Schifahrer, rechts eine Liste der Startliste.
Wenn links ein Schifahrer ausgewählt wird, kann dieser über die links/rechts Buttons nach rechts in die Startliste verschoben werden. Schifahrer rechts können dann mit den Buttons unten nach oben oder unten verschoben werden.



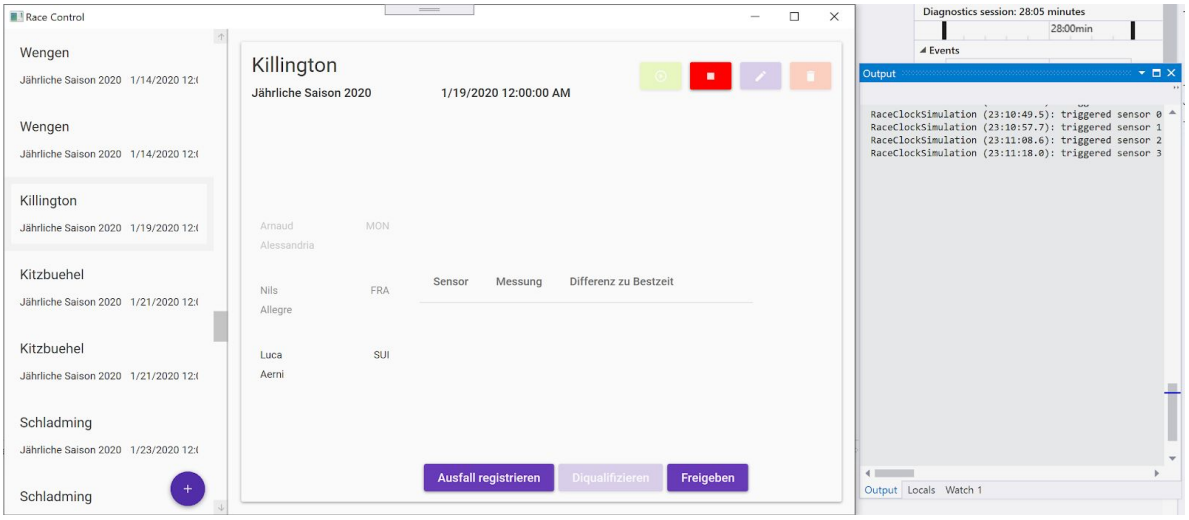
Nachdem ein neues Rennen angelegt wurde und dieses links in der Liste ausgewählt wird, kommt man auf diese Ansicht. Die Buttons zum starten der Rennausführung sind nun eingeblendet. Diese sind sichtbar, wenn es Rennfahrer in der Startliste gibt, die noch nicht gestartet sind. Der grüne Startbutton ist für die eigentliche Rennsimulation gedacht, hinter der die von Ihnen übergebene IRaceClock-Implementierung verwendet werden sollte. Jedoch ist dieser Button nur dummy-mäßig implementiert und löst keine Rennausführung aus.
Eine Rennen kann mit der von uns entwickelten Rennsimulation über den leicht ausgegrauten Startbutton ausgelöst werden.



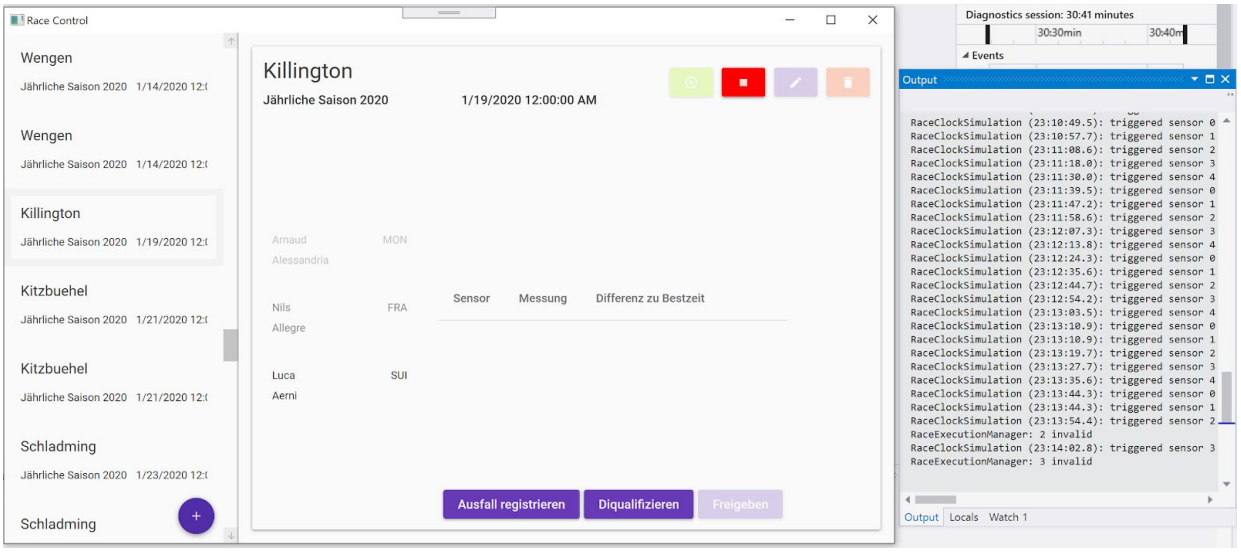
Nachdem der Simulationsstartbutton ausgewählt wurde, öffnet sich ein Fenster zum konfigurieren der Simulation.



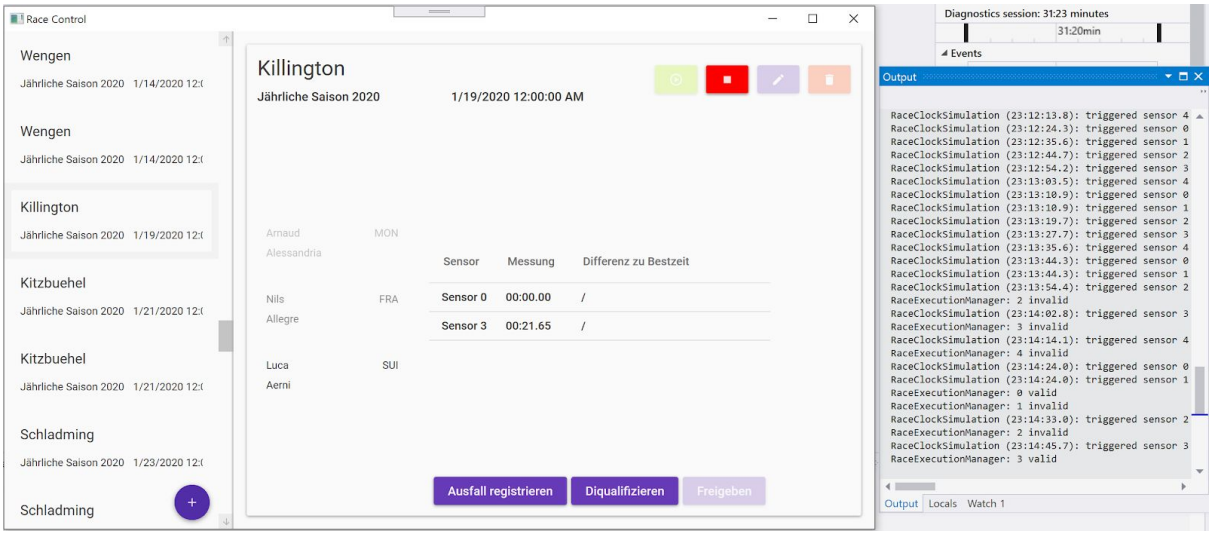
Nach dem der Dialog beendet wird, landet man in dieser Anzeige. Links wird der aktuelle Rennfahre, der nächste und übernächste Rennfahrer (mit immer weniger Opacity) dargestellt. In der Mitte befindet sich eine Liste der Zeitnehmungen des aktuellen Rennfahrers.
Wie rechts im Output-Window erkennbar ist, die Simulation löst bereits Sensorimpulse aus aber der RaceExecutionManager reagiert noch nicht darauf, da er sich noch nicht auf das Event subscribed hat. Der Benutzer hat nun die Möglichkeit, den Lauf über Freigeben zu starten.



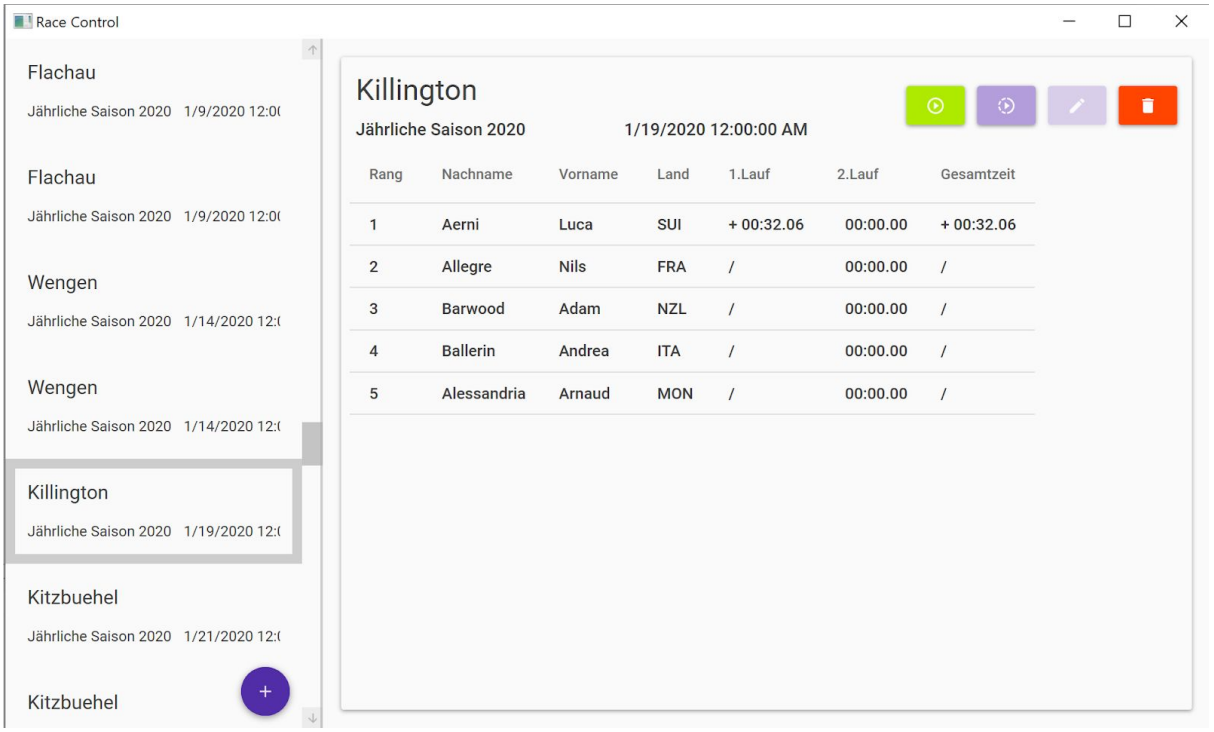
Die Sensorwerte sind laut RaceExecutionManager erst gültig, wenn der initiale Sensor erkannt wurde.



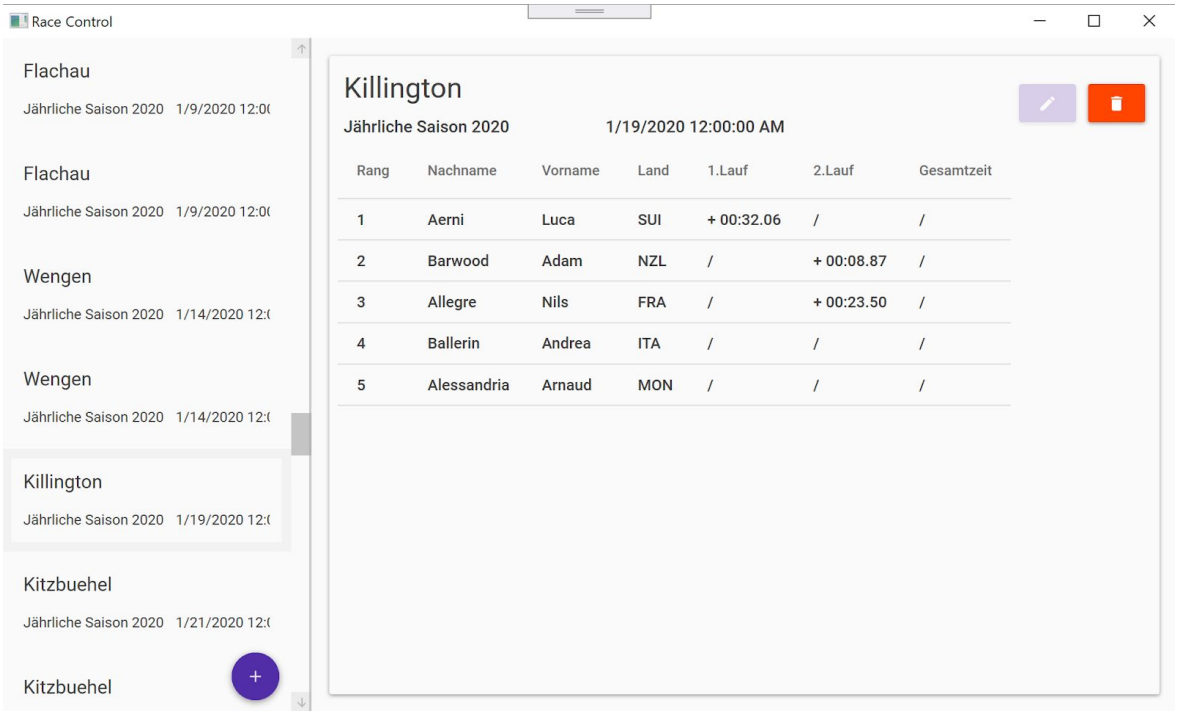
Der Sensor mit Id 0 bis 3 wurden erkannt, jedoch wurden die Sensoren 1 und 2 zu bald ausgelöst und durch die Normalverteilungsregel als ungültig empfunden.
Nach einer Rennfreigabe ist es nun auch möglich, den Skifahrer zu disqualifizieren.



Wenn man das Rennen nun über den Stop-Button beendet, sieht man eine temporäre Rangliste.

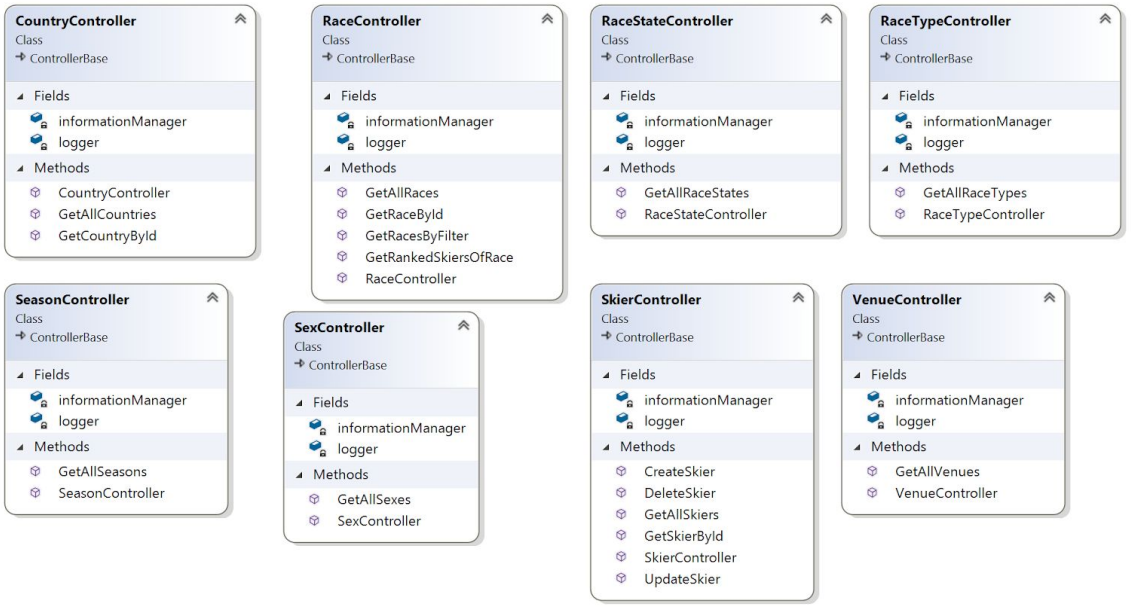


Und nach Abschluss des Rennens kommt man wieder auf diese Hauptansicht der Rangliste.

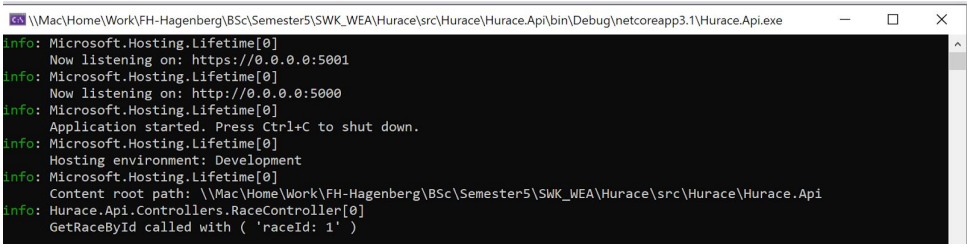


API

Die API stellt Methoden für den Zugriff auf die Businesslogik über einen Rest-Service zur verfügung.



Alle Methoden sind nur Weiterleitungen an die dementsprechenden Methoden des InformationManagers. Bevor die Aufrufe durchgeführt werden, wird aus Testbarkeitsgründen im Debug-Modus der Aufruf auf die Konsole ausgegeben.




```
        [HttpGet("{raceId}")]
        [ProducesResponseType(StatusCodes.Status200OK)]
        [ProducesResponseType(StatusCodes.Status404NotFound)]
        [ProducesDefaultResponseType]
        [OpenApiOperation(nameof(GetRaceById))]
        1 reference | 0 changes | 0 authors, 0 changes
        public async Task<ActionResult<Domain.Race>> GetRaceById(int raceId)
        {
#if DEBUG
            logger.LogCall(new { raceId });
#endif

            var race = await informationManager.GetRaceByIdAsync(
                    raceId,
                    venueLoadingType: Domain.Associated<Domain.Venue>.LoadingType.Reference,
                    seasonLoadingType: Domain.Associated<Domain.Season>.LoadingType.Reference,
                    raceTypeLoadingType: Domain.Associated<Domain.RaceType>.LoadingType.Reference)
                .ConfigureAwait(true);

            return race == null
                ? NotFound($"Invalid raceId: {raceId}")
                : (ActionResult<Domain.Race>)Ok(race);
        }
```

Durch die flexible Implementierung der Business-Logik ist diese hier nun sehr vielfältig einsetzbar. Einzelne Methoden können dann mittels Loading-Type parametriert werden.

Tests

Für die 3 Hauptkomponenten des DataAccessLayer, das GenericDao, den RowMapper und den SimpleSqlQueryGenerator wurden umfassende UnitTests erstellt.
Es wurden 25 UnitTests mit unterschiedlichen Parametrierungen entwickelt, wodurch insgesamt 113 unterschiedliche Testdurchläufe erzielt werden können.

Ein guter UnitTest ist unabhängig von anderen UnitTests. Dies ist aber bei Tests für die Datenzugriffsschicht nicht selbstverständlich, da eine Datenbank persistent ist.
Es ist auch keine Lösung, die Datenbank transient zu machen, da wir in einer weiteren Abfrage die Ergebnisse auf der Datenbank verifizieren können möchten und bei einer transienten Datenbank die Ergebnisse unmittelbar verloren wären.
Desweiteren ist es auch keine Lösung, das Dao im UnitTest zum Aufsetzen der Testsuite zu verwenden, wenn das Dao selbst getestet werden soll. Das selbe gilt dafür den Boilerplate-Code zum inserten/deleten in die Testsuite zu kopieren, da dieser Code ja auch wieder separat getestet werden müsste, bzw. angenommen werden würde, dass der Code einfach funktioniert.

Um dieses Problem zu lösen, haben wir uns dazu entschieden, einen TransactionScope am Beginn eines jeden Unittests der mit der Datenbank zutun hat, anzulegen und nach jedem Unittest diesen TransactionScope wieder zu disposen.
Ein TransactionScope kann mehrere unterschiedliche Transaktionen zu einer großen Transaktion zusammenfassen und wenn eine der einzelnen Transaktionen fehlschlägt, wird für alle zuvor durchgeführten Transaktionen ein *rollback* durchgeführt.
Wenn wir nicht explizit beim TransactionScope angeben, dass diese Transaktion erfolgreich war (SaveChanges), werden alle darin geschehenen Änderungen zurückgenommen, was uns genau das gewünschte Verhalten bei UnitTests liefert - Persistenz während des UnitTests und transientes Verhalten zwischen unterschiedlichen UnitTests.

Damit dieses Konzept sich nicht auf die lokale Entwicklung auswirkt (bei dem Persistenz in jeglicher Hinsicht erwünscht ist), werden zwei separate Datenbanken verwendet.
Eine Datenbank ist zum lokalen Entwickeln, dort werden alle Änderungen persistiert. Dies ist auch gleichzeitig eine lokale Replikation des späteren Produktivsystems.
Gleichzeitig gibt es auch eine andere Datenbank, auf der nur UnitTests durchgeführt werden sollen und bei der Änderungen nie länger als die Lebensdauer eines TransactionScopes existieren. Dort existiert nur der Datenzustand, der mit dem Insert-Script beschrieben ist.

Um BusinessLogik Komponenten wie *InformationManager* und *RaceExecutionManager* gut testen zu können ohne Abhängigkeit von Datenbankzugriffen, haben wir das Mocking-Framework FakeItEasy verwendet, um gemockte Implementierungen von *IDataAccessObject<AnyDomainObject>* an die BL weiterzureichen.

Da wir durch die generische Implementierung unseres DataAccessLayer eine sehr allgemeine Formulierung der Schnittstellen haben, ist es sehr aufwendig komplexere Business-Logik Operationen zu testen (der Setup-Code ist sehr komplex).
Einerseits wurden dadurch UnitTests implementiert, die sehr sehr stark vom Code der Business-Logik abhängen, also in welcher Reihenfolge welche Methoden des DataAccessLayer aufgerufen werden. Beim UnitTest für die Rennausführung brauchte es ca. 600 Zeilen (!) nur an SetupCode. Nachdem wir den RaceExecutionManager nocheinmal überarbeiten mussten, wurde die Reihenfolge der Aufrufe des DataAccessLayer verändert und der UnitTest funktioniert leider nicht -> darum wurde er entfernt. Solche UnitTests haben keinen großen Mehrwert, da dann sowieso der Code gar nicht mehr angegriffen werden darf, was ja durch den Einsatz von UnitTests möglich sein sollte ohne die korrekte Funktionsweise anderer abhängigen Komponenten zu verlieren.