

# Hurace

SWK Projekt – WS 19/20

Andreas Hahn – S17010307015

David Strauss – S17010307042

# Dokumentation

# 1 Ausbaustufe 1

## 1.1 Datenmodell

Bei der Auswahl der Datenbank wurde auf Grund der guten Integration auf eine Microsoft SQL Express LocalDB gesetzt. Die Administration der Datenbank kann somit integriert in der Visual Studio IDE erfolgen. Das Datenmodell wurde bestmöglich nach Anforderung normalisiert. Es werden, wo sinnvoll erscheinend, Views und Stored Procedures eingesetzt, um die Operationen auf der Datenbank bestmöglich zu optimieren. Siehe Abb. 1.1 des Datenbank-Schemas.

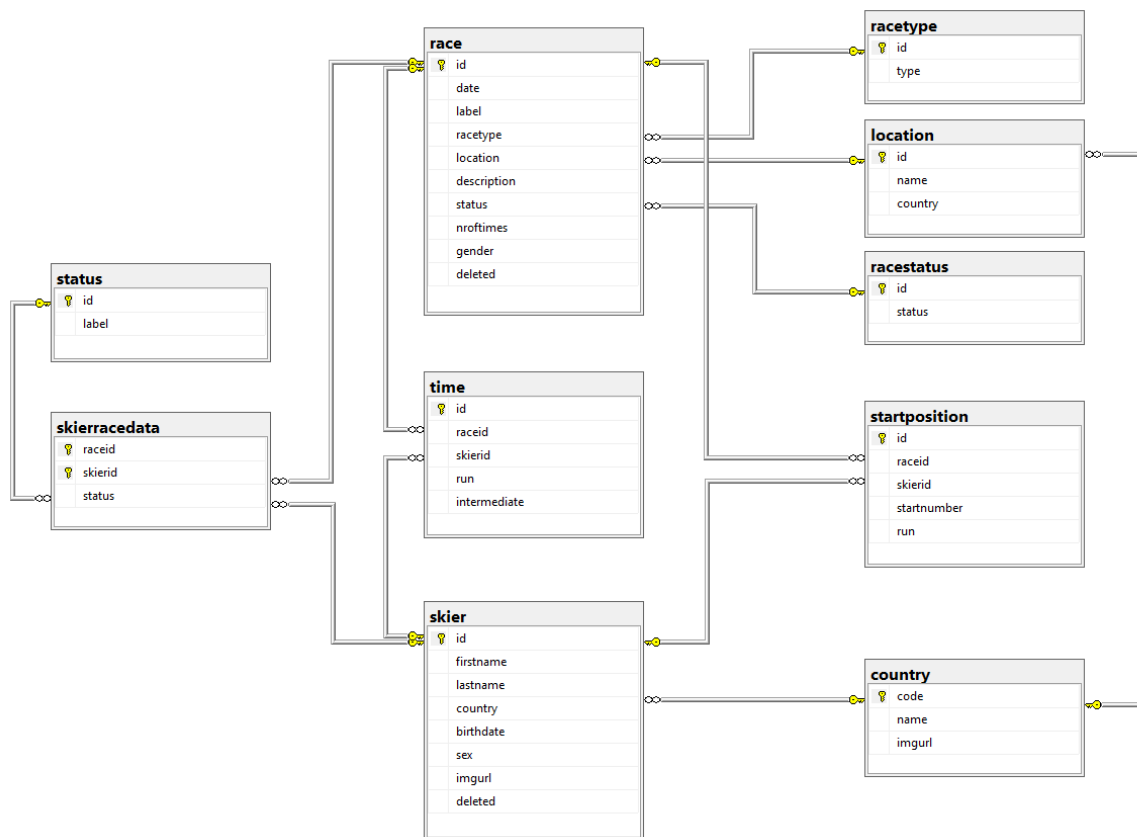


Abb 1.1: Datenbankschema der Hurace Applikation

## 1.2 Data Access Layer

Der Data Access Layer wurde generisch, unabhängig der darunter verwendeten Datenbank, implementiert. Die Entitäten der Datenbank wurden in Domänen-Klassen überführt. Zugriff auf die Daten der Datenbank erfolgt ausschließlich über Data Access Objekte (DAO). Es wurde versucht Schnittstellen und Implementierung sauber zu trennen. Generische Datenbankoperationen sind in ein eigenes Projekt ausgelagert, sind somit kein direkter programmspezifischer Teil und können daher einer Wiederverwendung in anderen Projekten zugeführt werden.

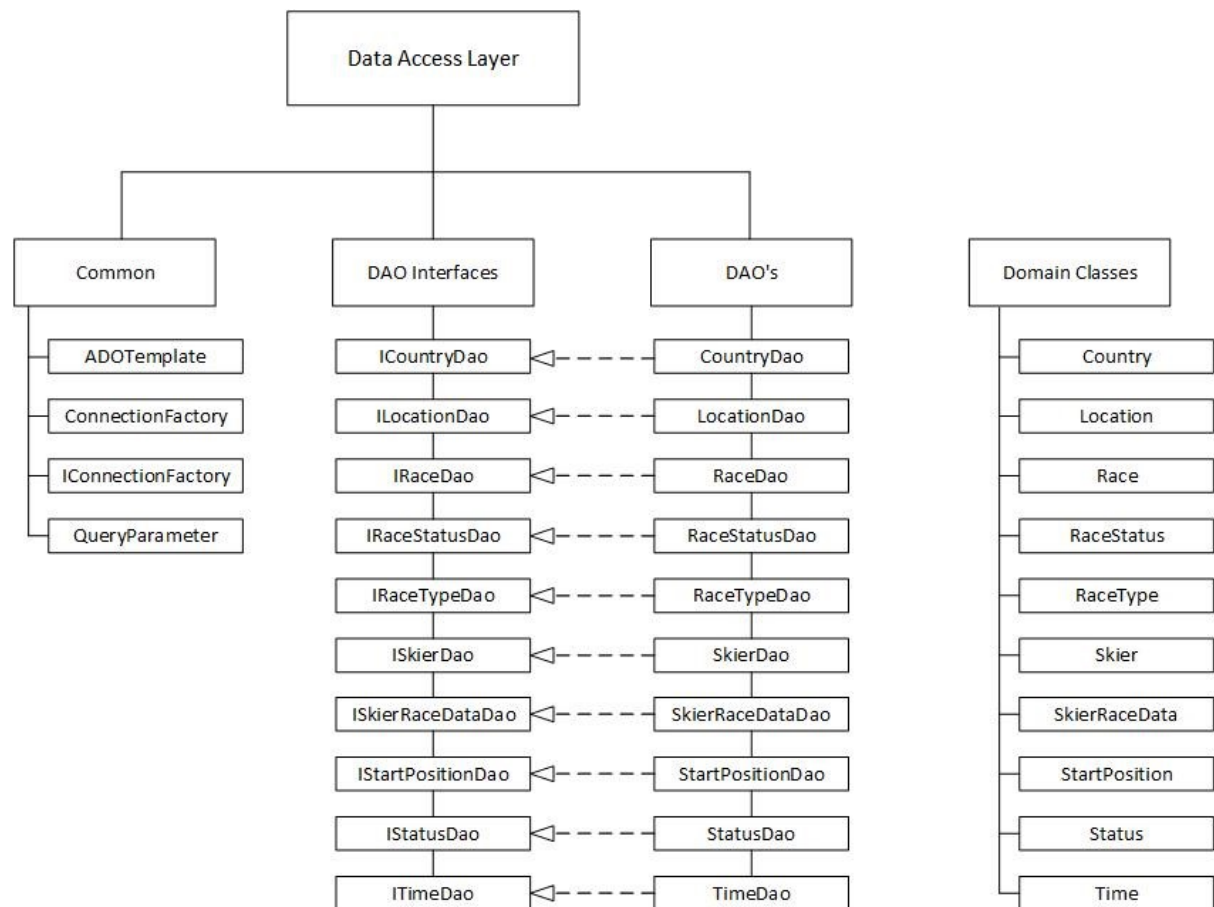


Abb 1.2: Aufbau Data Access Layer

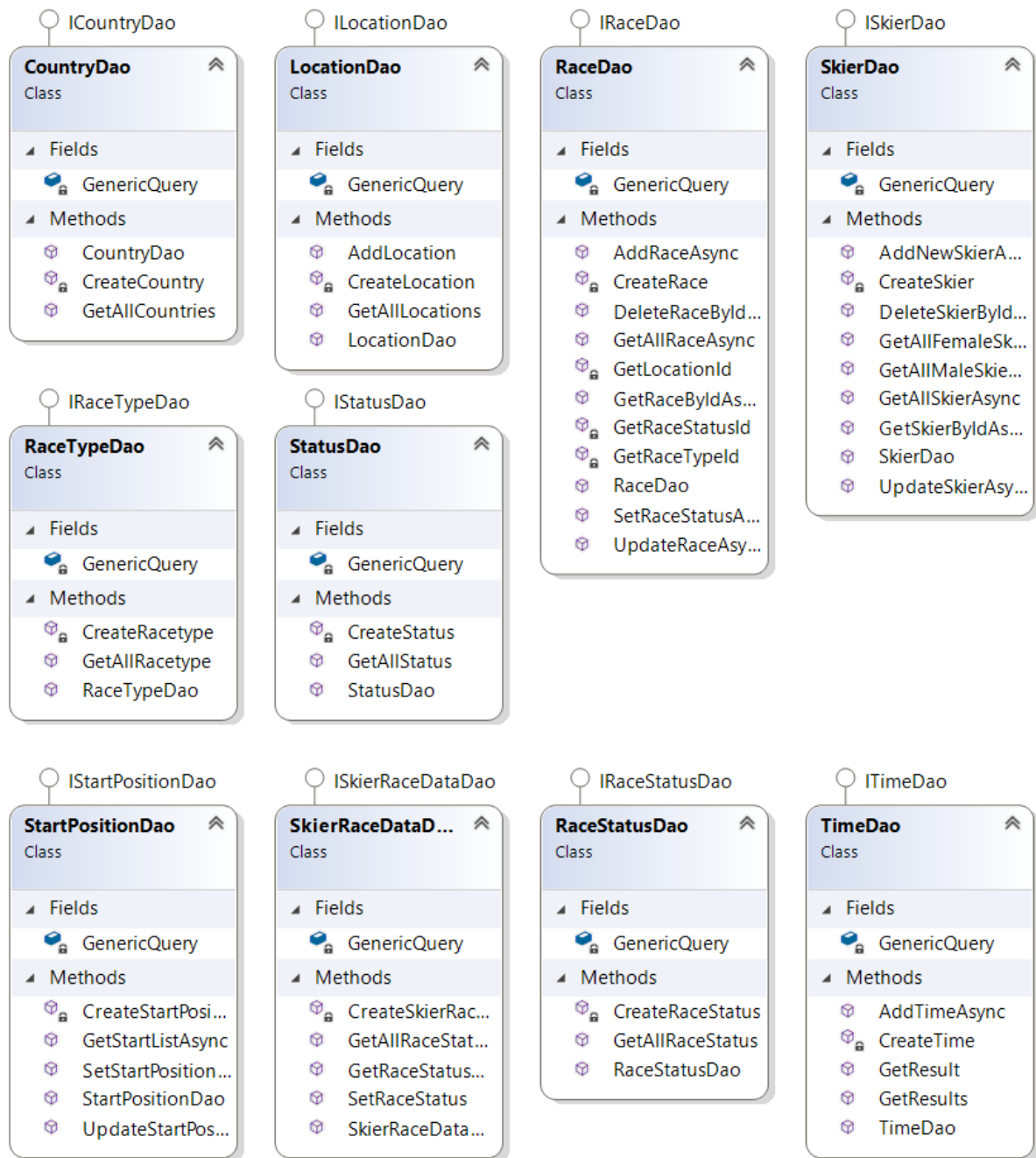
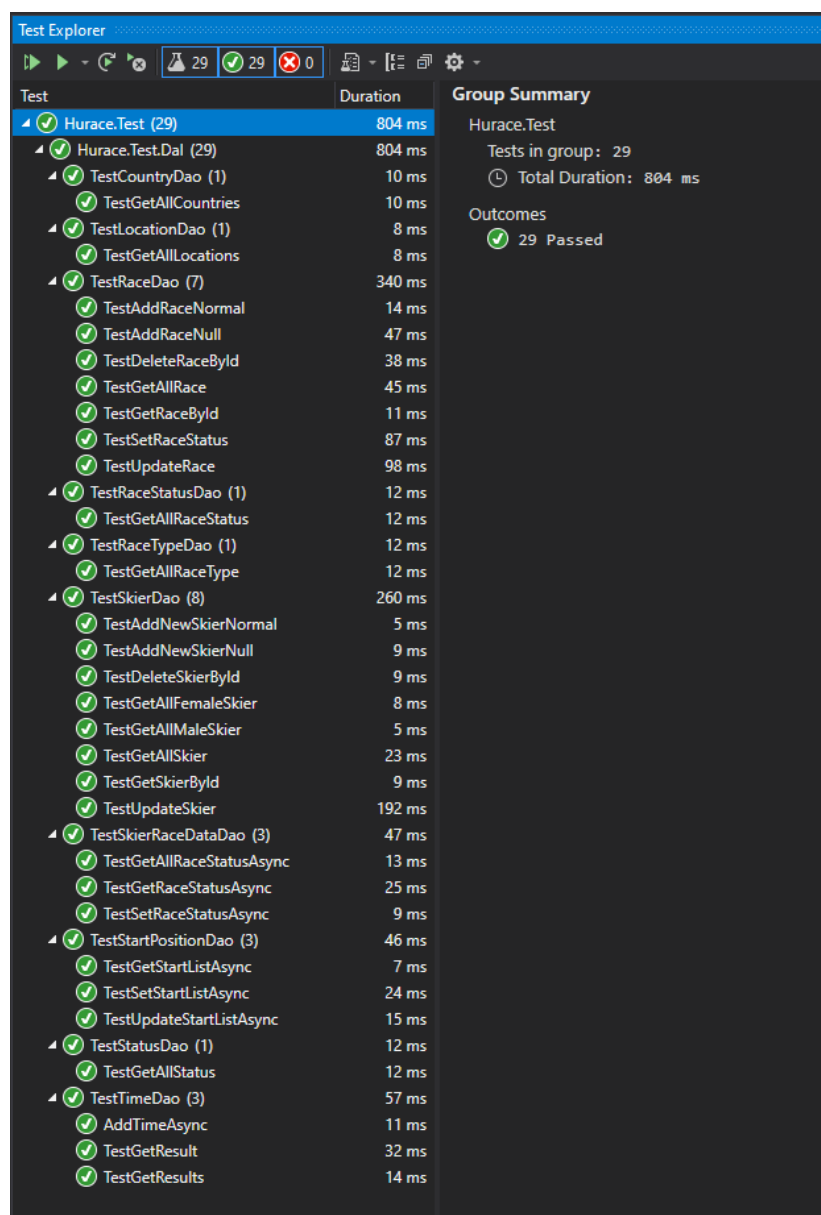


Abb 1.3: UML Klassendiagramm der Data Access Objekte

### 1.3 Unit Tests

Die Unit Tests wurden mit dem xUnit Framework implementiert. Die DA-Objekte werden mit einer tatsächlichen Datenbankverbindung mit Testdaten überprüft. Im Gegensatz zu einem Datenbankmock, können so bessere Live-Bedingungen simuliert werden. Für die Tests muss jedoch von einer konsistenten Datenbank ausgegangen werden. Für die einzelnen Testabschnitte wird die Datenbank immer wieder in Initial-Zustand versetzt, um diese Integrität zu gewährleisten. Mittels *Collection Fixtures* kann die Datenbank als Ressource über alle Tests geteilt werden und im Konstruktor vor jedem Test auf Initialzustand gebracht werden. Weiters wurde bei der Erstellung des einzelnen Tests auf eine möglichst hohe Unabhängigkeit sowie eine hohe Codeabdeckung geachtet.



Test	Duration	Group Summary
✓ Hurace.Test (29)	804 ms	Hurace.Test
✓ Hurace.Test.Dal (29)	804 ms	Tests in group: 29
✓ TestCountryDao (1)	10 ms	⌚ Total Duration: 804 ms
✓ TestGetAllCountries	10 ms	Outcomes
✓ TestLocationDao (1)	8 ms	✓ 29 Passed
✓ TestGetAllLocations	8 ms	
✓ TestRaceDao (7)	340 ms	
✓ TestAddRaceNormal	14 ms	
✓ TestAddRaceNull	47 ms	
✓ TestDeleteRaceById	38 ms	
✓ TestGetAllRace	45 ms	
✓ TestGetRaceById	11 ms	
✓ TestSetRaceStatus	87 ms	
✓ TestUpdateRace	98 ms	
✓ TestRaceStatusDao (1)	12 ms	
✓ TestGetAllRaceStatus	12 ms	
✓ TestRaceTypeDao (1)	12 ms	
✓ TestGetAllRaceType	12 ms	
✓ TestSkierDao (8)	260 ms	
✓ TestAddNewSkierNormal	5 ms	
✓ TestAddNewSkierNull	9 ms	
✓ TestDeleteSkierById	9 ms	
✓ TestGetAllFemaleSkier	8 ms	
✓ TestGetAllMaleSkier	5 ms	
✓ TestGetAllSkier	23 ms	
✓ TestGetSkierById	9 ms	
✓ TestUpdateSkier	192 ms	
✓ TestSkierRaceDataDao (3)	47 ms	
✓ TestGetAllRaceStatusAsync	13 ms	
✓ TestGetRaceStatusAsync	25 ms	
✓ TestSetRaceStatusAsync	9 ms	
✓ TestStartPositionDao (3)	46 ms	
✓ TestGetStartListAsync	7 ms	
✓ TestSetStartListAsync	24 ms	
✓ TestUpdateStartListAsync	15 ms	
✓ TestStatusDao (1)	12 ms	
✓ TestGetAllStatus	12 ms	
✓ TestTimeDao (3)	57 ms	
✓ AddTimeAsync	11 ms	
✓ TestGetResult	32 ms	
✓ TestGetResults	14 ms	

Abb 1.4: Unit Tests Data Access Layer

## **2 Ausbaustufe 2**

### **2.1 Geschäftslogik**

Bei der Implementierung der Geschäftslogik wurde darauf geachtet, dass alle Zugriffe der darüber liegenden Schichten (wie etwa die Benutzerzugriffsschicht) über die Logikschicht laufen. Dies resultiert aus dem Grundgedanken, dass eine Schicht nur mit der darüber und darunter liegenden Schicht kommuniziert. In weiterer Folge ermöglicht diese Architekturentscheidung einen einfacheren Austausch von Schichten bzw. Komponenten. Die Geschäftslogik übernimmt somit für die darüber liegende Benutzerschicht, die Kommunikation mit der Datenzugriffsschicht. Die Geschäftslogik muss weitere Funktionen zur Verfügung stellen, um den Ablauf eines gesamten Rennens zu betreuen. Weiters muss die Geschäftslogik über eine Komponente verfügen, die den aktuellen Zustand des Programms zu jederzeit eindeutig kennt.

### **2.2 Benutzeroberfläche**

Die Benutzeroberfläche soll einfach und möglichst intuitiv zu bedienen sein, sodass Fehleingaben und Fehlverwendung ausgeschlossen werden kann. Der Benutzer soll hier Rennen anlegen und verwalten können sowie die dazugehörigen Startlisten anlegen können. Weiters sollen Rennen inklusiver beider Läufe (bei Slalom und Riesenslalom) betreut werden können. Dies inkludiert eine einfache Freigabe des Rennens sowie der einzelnen Läufer. Während eines Rennens können einzelne Läufer auch manuell durch den Rennleiter auf den Status „Disqualifiziert“ oder „Nicht gestartet“ gesetzt werden. Über die Oberfläche kann weiters die Simulation einer Zeitnehmung gestartet werden, mit deren Hilfe ein gesamter Rennlauf durchgespielt werden kann. Auch die Visualisierung der Live-Bildschirme kann über die Oberfläche gestartet werden. Hierbei wurde darauf geachtet, dass weitere Bildschirme (realisiert als WPF-Usercontrols) durch einfaches hinzufügen in eine Liste, nachträglich hinzugefügt werden können.

### **2.3 Automatisierte Tests**

Die Geschäftslogik wird durch automatisierte Unit-Tests auf Funktionalität geprüft. Um die dahinter liegende Datenzugriffsschicht zu abstrahieren wird ein Mocking-Framework eingesetzt. (Hierfür wird das Framework „moq“ verwendet.) Hierbei werden den zu testenden Logik-Komponenten per Dependency-Injection ein mittels dem Framework gefaktes DAL bzw. DAO Objekt übergeben. Somit erfolgt der Zugriff innerhalb der Logik nicht mehr auf die tatsächliche Datenzugriffsschicht, sondern auf das vorher präparierte Objekt und ermöglicht ein granulares Testen des tatsächlichen Logikcodes ohne Abhängigkeiten hin zur Datenzugriffsschicht.

## 2.4 Architektur

