zühlke
empowering ideas

Car Tuning Configurator

Projektaufgabe für Praktikanten

Von Mademidda und Danijel

zühlke
empowering ideas

Inhaltsverzeichnis

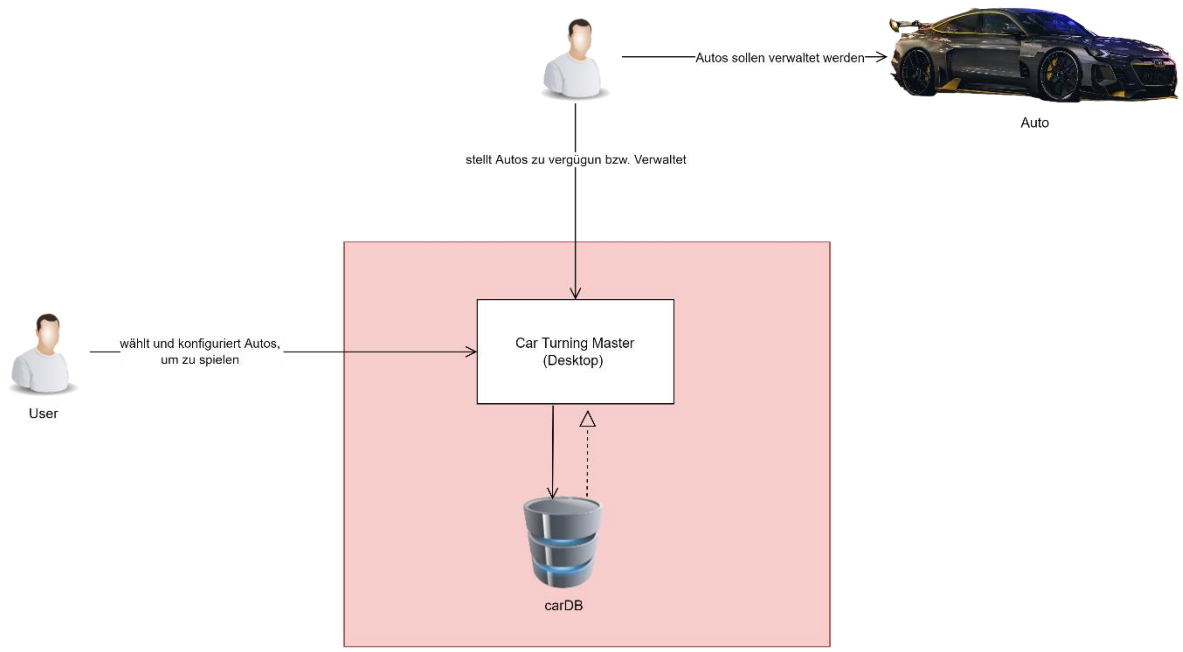
Inhaltsverzeichnis

Analyse

Bevor Wir uns Gedanken machten, wie die Applikation aussehen soll, was sie genau machen soll, und wie wir dies umsetzen sollen, machten wir Schritt für Schritt einen Plan. Angefangen haben wir mit der Analyse.

Kontext

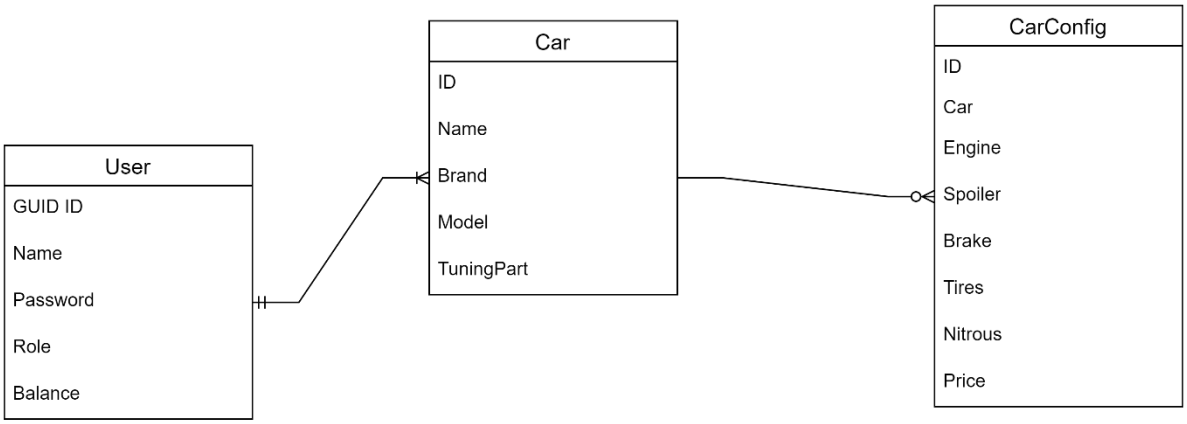
Als erstes haben wir ein Kontextdiagramm erstellt. Hier sehen wir in welchem Kontext die Applikation genau verwendet wird.



Die Applikation und die Datenbank sind in einer sicheren Umgebung. Die Applikation greift auf die Datenbank zu, sie kann Daten einfügen und abrufen. Der User kann in der Applikation Autos für das Spiel konfigurieren. Oben haben wir noch den Admin, welcher die im Spiel vorhandenen Autos verwaltet und neue Autos hinzufügt.

Daten

Für die Analyse der Daten haben wir ein logisches Datenmodell erstellt.



Hier haben wir drei Collections. Die erste ist für die Benutzerdaten. Neben dem Klassischen Namen und dem Password haben wir noch die Attribute «Role» und Balance. Balance ist der Kontostand mit der Ingame-Währung, von welcher man sich die Tuning Parts kaufen kann. Das Attribut «Role» definiert welche Rolle der Benutzer hat, Rolle «admin» oder Rolle «user» als normaler Benutzer.

In der Mitte haben wir die Collection Car für die Autos. Neben den Klassischen Attributen mit den Eigenschaften, haben wir am Schluss noch das Attribut «TuningPart» welches vom «CarConfiguration» erbt.

Im «CarConfiguration» haben wir alle Tuningparts gespeichert, welche zum Auto hinzugefügt werden können. Am Schluss haben wir noch das Attribut Price. Darin sollte der Gesamtwert des Autos mit den hinzugefügten Tuningparts gespeichert werden.

Bei dem Verbinden der Collections war unsere Idee, dass der User ein oder mehrere Autos haben kann. Das Auto hingegen braucht einen Benutzer und kann auch nur einen einzigen haben welcher im zugewiesen wurde. Die Verbindung ist also 1 mandatory zu 1-n.

Ein Tuningpart muss nicht ausgewählt sein, es können aber auch mehrere gleichzeitig hinzugefügt werden. Deswegen ist die Verbindung zwischen «Car» und «CarConfiguration» 1 zu 0-n.

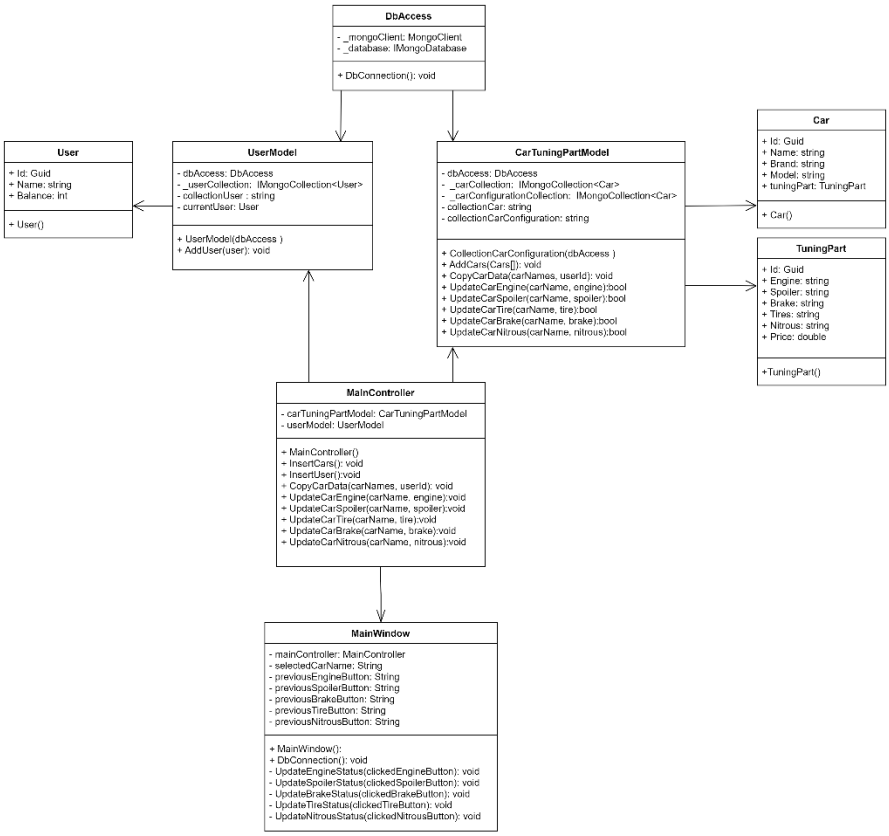
Funktionale Anforderungen

Hier haben wir die Aufgaben und Funktionen der Applikation analysiert. Um Diese mit den Akteuren aufzuzeigen haben wir ein Use Case Diagramm erstellt.

Auf der einen Seite haben wir den normalen Benutzer. Er kann sich seine Autos auswählen und diese mit den Tuning Parts Konfigurieren.

Andererseits haben wir den Admin. Er kann alle CRUD-Befehle ausführen, um die Autos zu verwalten. Später haben wir uns entschieden doch kein interface für den Admin zu implementieren, weil es zu aufwändig wäre. Wir lassen die bisherigen Diagramme aber so sein, falls wir unsere Applikation irgendwann erweitern werden wollen.

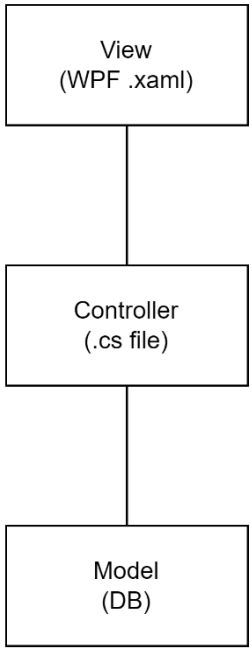
Klassendiagram



Unser Projekt hat acht Klassen, die auf dem MVC-Pattern basieren, zwei Modelle CarTuningPartModel und UserModel und einen MainController, der die beiden Modelle über das MainWindow steuern muss. Eine Klasse DbAccess wird für die Verbindung zu MongoDb verwendet. Diese steuert die beiden Modelle. Der Grund, warum wir uns für zwei Modelle entschieden haben, ist, dass wir nicht wollten, dass der Code unüberschaubar ist.

Architektur

Um die Architektur aufzuzeigen haben wir ein einfaches High-Level Architekturdiagramm erstellt. Es zeigt auf wie die Applikation aufgebaut ist.



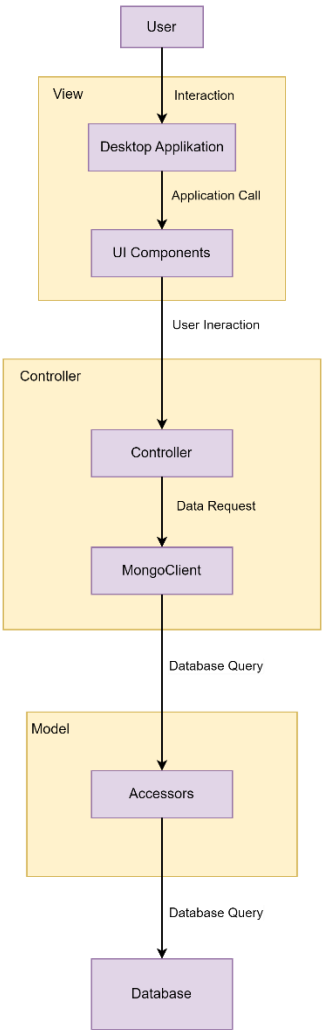
Die Architektur ist nach dem Klassischen MVC-Pattern aufgebaut. Oben haben wir die View, also das GUI. In unsere Applikation ist dies das .xaml WPF-File. Der Controller ist das zum .xaml dazugehörige .cs (C#) File, welches die Benutzerinteraktionen steuert. Zum Schluss haben wir noch das Model, welches die Verbindung zur Datenbank herstellt. Hier sind alle Setter und Getter definiert.

Die Applikation ist also eine Klassische Desktop Applikation, welche in C# nach MVC-Pattern geschrieben wurde. Das GUI wird im WPF-Framework als .xaml designed. Für die Datenbank verwenden wir MongoDB, weil die Konfigurationen in echten Spielen normalerweise auch im JSON gespeichert werden. Diese werden aber in der Regel jedoch nicht über einen DB-Server abgerufen, sondern lokal gespeichert, wenn das Spiel ein Offline-Game ist.

Design

Architektur

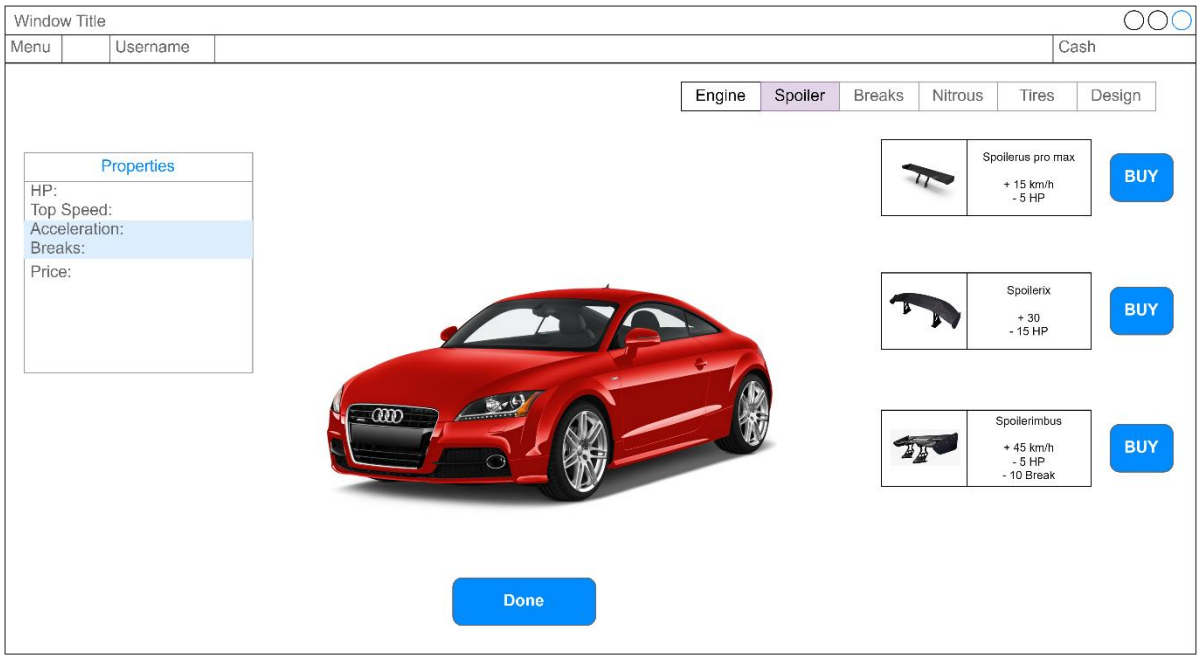
Wir ergänzen nun unser vorheriges High-level Architekturdiagramm zu einem detaillierten Architekturdiagramm, um unsere Applikationsarchitektur langsam zu designen.



Hier haben wir das MVC noch mit Details ergänzt. Ganz oben haben wir den User, welcher mit der View interagiert. Dazu gehört das, was der Benutzer von der Applikation auf dem Bildschirm sieht. Er ruft die einzelnen Komponenten im GUI auf. Diese Interaktion geht dann zum Controller rüber, welcher im Controller ist. Um Daten aufzurufen, ruft der Controller den MongoClient auf, um die Daten aus der Datenbank zu holen. Dieser ruft dann über die Accessors im Model die Datenbank auf und ruft die Daten auf.

UI-Design

Beim UI-Design ging es um das eigentliche Design der Applikation und dessen Aussehen. Um zu zeigen, wie wir uns das Aussehen unserer Applikation vorstellen, haben wir ein grobes Mockup gezeichnet. Zuerst freihändig am Whiteboard Ideen ausgetauscht, danach ein fertiges Mockup im Draw.io erstellt, sobald wir uns auf eine Version geeinigt haben.

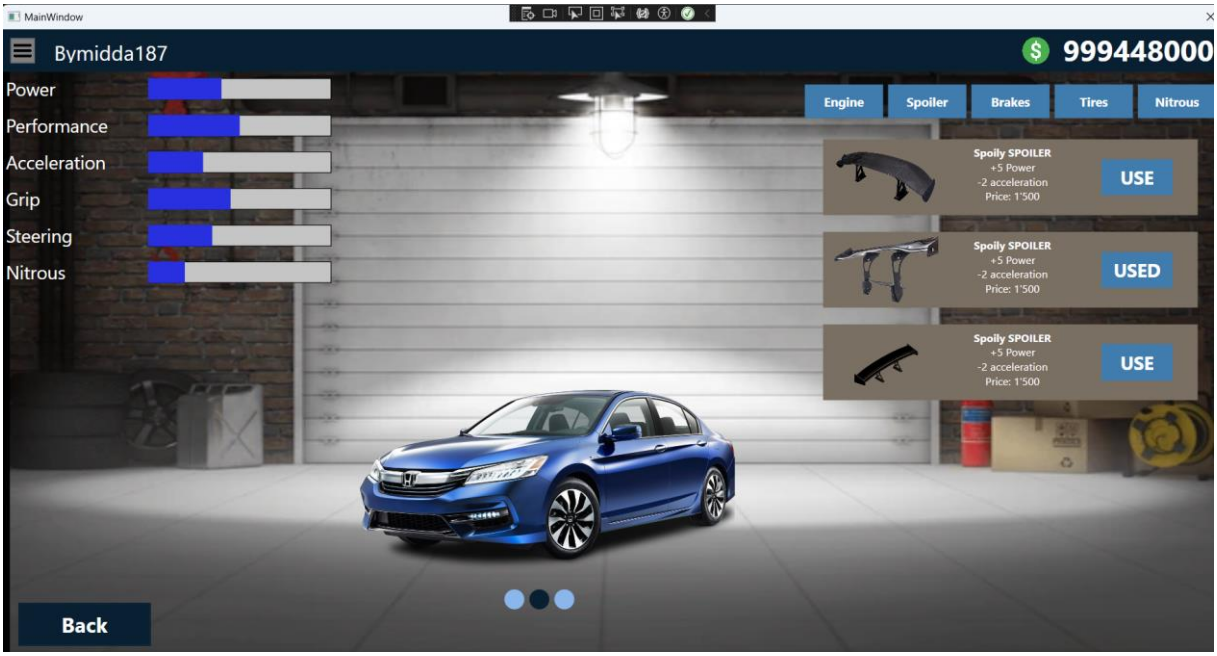


Unsere Idee war es, Das Fenster in Header, Body und Footer zu unterteilen. Im Header sieht man den Benutzernamen von dem Benutzer welcher aktuell eingeloggt ist und dessen Kontostand. Links in der Ecke ist noch ein Menü-Button, welcher die Einstellungen öffnet oder zur Home Seite zurückkehrt.

Der Hauptteil ist im Body. In der Mitte sieht man das Auto welches aktuell ausgewählt ist. Um die Autos mit den Tuning Parts zu konfigurieren haben wir auf der rechten Seite oben eine Menübar um zwischen den Kategorien zu wechseln. Wenn man eine Kategorie anwählt, kommen die einzelnen Teile zum Vorschein, welche man kaufen oder auswählen kann, wenn es bereits gekauft wurde. Die ausgewählten Teile auf der rechten Seite haben Auswirkungen auf die Eigenschaften des Autos, welche man auf der linken Seite des Fensters sehen kann. Unsere Überlegung dahinter war, dass wir die Eigenschaften des Autos mit Punkten von 1-100 beurteilen. Den ganzen Mechanismus, um die Autos einzeln auszuwählen und diese zu einzeln Konfigurieren haben wir uns später überlegt.

Zum Schluss haben wir unten noch den Footer. Dort kommen die Buttons wie «Done» oder «Back», welche zum Hauptmenü zurückkehren. Auch das Auswählen der Autos könnte man hier implementieren.

Nachdem die Zeichnung im Draw.io grob gezeichnet wurde, haben wir einen Prototyp von dem Design im WPF erstellt. Wir haben beide parallel an dem Prototyp gearbeitet und uns ausgetauscht. Jeder hat seine eigene Version gemacht, wie er sich das Aussehen vorstellt. Am Schluss haben wir uns auf das Design von Danijel geeinigt und zusammen etwas überarbeitet, weil es mehr nach einem Spiel aussah.



Das Grundgerüst wurde von dem Mockup aus Draw.io übernommen. Im Hintergrund sehen wir eine Garage und in der Ein rotes Auto. Die Autos sollten auswählbar sein, was wir erst später implementiert haben, nachdem wir mehr oder weniger fertig waren mit der Implementation der Grundfunktionen von dem Konfigurator. Header und Footer wurden in etwa genau wie aus dem draw.io Mockup übernommen. Für die Balance haben wir noch ein Icon statt einem Text verwendet. Im Allgemeinen haben wir darauf geachtet, dass die Farben analog zueinander sind, genauso wie im Webdesign.

Im Body sind die eigentlichen Funktionen der Applikation. Hierfür haben wir ein Grid-Panel genommen und dieses in drei Spalten unterteilt. Rechts haben wir die Tuning Parts und deren Kategorien oben in einer Menübar welche aus normalen Buttons besteht. Das Layout wurde fast eins zu eins aus dem vorherigen Mockup übernommen. Die Mittlere Spalte wurde anfangs leer gelassen. Später haben wir die Autos aus dem Hintergrund in diese Spalte verschoben damit man sie auswählen kann.

Der interessanteste Teil des Konfigurators ist in der linken Spalte. Unsere Idee mit den Punkten von 1 – 100 haben wir hier umgesetzt, jedoch in Form von Progress Bars statt normalen und etwas langweiligeren Ziffern. Den Maximalwert haben wir in jeder Bar auf 100 gesetzt. Wenn jetzt ein Wert gesetzt wird, kommt ein blauer Balken zum Vorschein und zeigt auf welchem Wert die Eigenschaft liegt. So kann man besser visualisieren, wie gut die Leistung des Autos ist. Die einzelnen Attribute haben wir ebenfalls aus dem vorherigen Mockup angepasst.

Implementierung

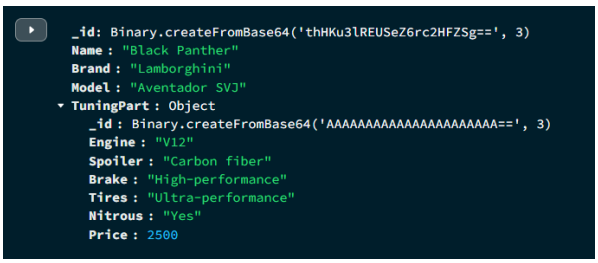
Datenbank

Für die Datenbank verwenden wir MongoDB, weil das Speichern der Daten in JSON, oder hier BSON, für Spiele besser optimiert ist als SQL. Die BSON-Dokumente liegen aber in einer Datenbank. Dies hätte vielleicht Vorteile, um die Sicherheit gegen Hacks zu gewährleisten, wenn unser Spiel ein Richtiges Online Game wäre.

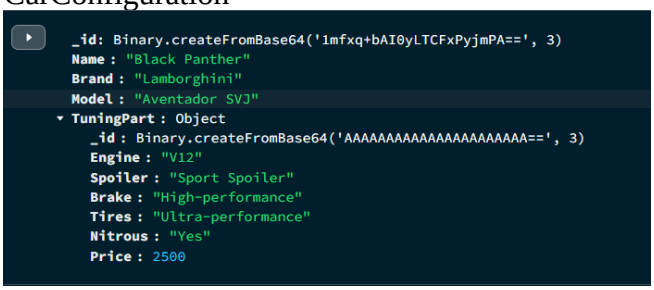
Das Ziel war, dass der User drei Autos auswählen kann und diese aus Collection Car in Collection CarConfiguratio kopiert werden und die Tunin Parts angepasst werden können.

Sobald der User die Applikation startet, werden die Datenbank TunigPartConfigration und die Collection Car und User automatisch in MongoDb angelegt, wenn die Option Auto gewählt wird, wird eine weitere Collection CarConfiguration angelegt und Tuning Part geändert.

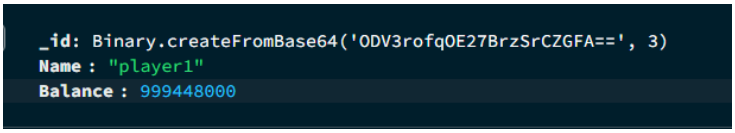
Car



CarConfiguration

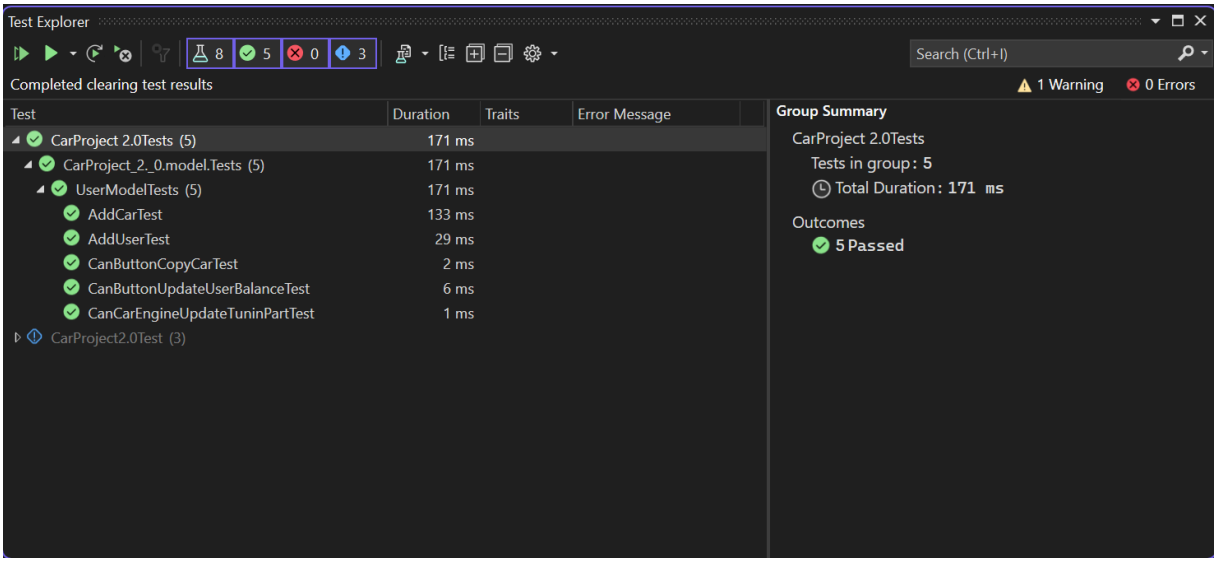


User



Test

Zum Schluss haben wir am Ende noch fünf Unit Tests gemacht. Dies hat uns auch etwas Zeit geraubt, weil wir noch nie solche Tests mit C# in Visual Studio gemacht haben.



Als erstes haben getestet ob die Autos und User überhaupt in die Datenbank eingefügt werden. Danach testen wir mit CanButtonCopyCar ob das Auto aus der Car Collection in die CarConfiguration Collection kopiert wird. Zum Schluss Testen wir noch ob der Kontostand verändert wird, wenn ein Tuning Part gekauft wird und ob das Tuning Part zum Auto in der Collection hinzugefügt wird, das vorherige Tuning Part wird überschrieben. Wie wir sehen liefen alle Tests Erfolgreich.

zühlke
empowering ideas

Reflexion

Das Projekt war eine Herausforderung und wir mussten herausfinden, wie wir es implementieren, planen und umsetzen konnten. Zu Beginn bestand unser Projekt lediglich aus einem Klassendiagramm, einem Use Case und einem Kontextdiagramm. Wir waren uns jedoch nicht über das Design einig, also machte jeder von uns ein Design in WPF, um zu sehen, wer besser war. Wir haben einen Tag lang damit gearbeitet und festgestellt, dass es nicht so weit gehen sollte. Teamarbeit ist sehr wichtig, deshalb haben wir am nächsten Tag immer besprochen, was wir gemacht haben. Wir trafen uns täglich, um zu besprechen, wer was macht, und konnten die Arbeit so aufteilen, dass jeder von uns etwas im Frontend und im Backend zu tun hatte. Wir waren unzufrieden mit der Umsetzung der Nutzer im Projekt, weil es nicht funktioniert, dass jeder User sein Auto nicht wählen kann. Das ist, warum wir User und Login-Seite entfernt haben und diese Applikation zu benutzen, wenn man das Problem beheben will, mussten wir eine Menge Zeit verwenden, um andere nicht zu machen. Unser Ziel wurde nicht erreicht, aber wir sindzufrieden, dass unsere Applikation funktioniert. Wir können den Tuning Part ändern und in der Datenbank speichern.