JOHANNES KEPLER
UNIVERSITÄT LINZ

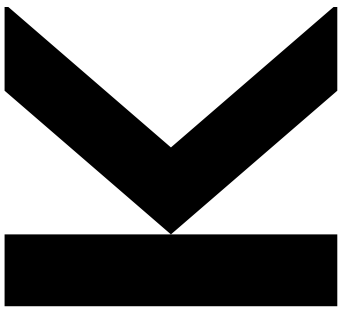
Eingereicht von
Michael Haas

Angefertigt am
Institut für Systemsoftware

Betreuer
Dipl.-Ing. Sebastian Kloibhofer

Mai 2023

Testautomatisierung einer WPF-Applikation mittels Appium



Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

Informatik

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract. Das Testen von Software mit grafischer Benutzeroberfläche spielt bei Applikationen mit einem Kunden als Endnutzer eine wichtige Rolle. Speziell bei schnellen Fehlerbehebungen oder Erweiterungen der Software durch den Entwickler ist es essenziell, dass die Funktionalität weiterhin gewährleistet werden kann. Um dies sicherzustellen, kann die Software nun entweder manuell oder automatisiert getestet werden. In beiden Fällen werden zuerst Testfälle verfasst, welche die auszuführenden Aktionen und die erwünschten Ergebnisse widerspiegeln. Beim manuellen Testen führt eine Person die verschiedenen Aktionen aus und überprüft die Ergebnisse mit den festgehaltenen, erwarteten Resultaten. Da das manuelle Testen üblicherweise viel Zeit in Anspruch nimmt und auch Fehler übersehen werden können, wird das Testen oft automatisiert. Mittels eines Testtreibers wird dazu programmatisch auf die Elemente der Benutzeroberfläche zugegriffen und das jeweilige Verhalten der Applikation überprüft. In dieser Bachelorarbeit entwickelten wir eine Testapplikation für die Software auf den ASFINAG-Verkehrsautomaten, welche die Benutzeroberfläche auf ihre Funktionalität überprüft. Für das Abrufen der Ergebnisse inklusive Testbericht erweiterten wir eine bestehende Webseite. Somit kann die Software mit einem einfachen Starten der Testapplikation zu jeder Zeit automatisch überprüft werden.

Inhaltsverzeichnis

Kapitel 1

Einführung

Das Unternehmen ASFINAG bezieht seine Einnahmen aus dem Verkauf der Lkw-Maut, der Vignette und der Streckenmaut. Seit einigen Jahren gibt es neben der Klebevignette und dem Streckenmautticket auch die Digitale Vignette und Digitale Streckenmaut. Neben Onlineshop und Vertriebspartnern können diese auch an ca. 70 Automaten auf diversen Rastanlagen und Grenzübergängen bargeldlos gekauft werden. Auf diesen Automaten läuft eine GUI-Applikation, welche mit dem WPF-Framework von .NET 6 entwickelt wurde. Da es immer wieder zu Softwareänderungen wie zum Beispiel kleinen Designänderungen oder Fehlerbehebungen kommen kann, muss die Applikation regelmäßig getestet werden. Dies wurde bisher durch ein externes Unternehmen durchgeführt. Hierfür wurden unterschiedliche Testfälle mit den gewünschten Zwischenergebnissen erstellt und die Software manuell getestet. Da dies auf Dauer keine geeignete Lösung ist und vor allem bei wichtigen Fehlerbehebungen ein schnelles Feedback wichtig ist, wurde beschlossen, die Tests zu automatisieren. Für das automatisierte Testen stellte sich das Appium-Framework mit dem WinAppDriver als am besten geeignet heraus. Eine bereits entwickelte Testapplikation wies jedoch sehr viele Codeduplikationen sowie eine niedrige Wiederverwendbarkeit auf und deckte die komplette Softwarefunktionalität noch nicht ab. Deshalb implementierten wir die Testapplikation im Rahmen dieser Bachelorarbeit neu. Ziel war es außerdem, einer im Laufe dieser Bachelorarbeit entwickelten neuen GUI eine gute Basis für das Testen bereitzustellen. Um fehlgeschlagene Tests reproduzieren zu können, bauten wir in die Testapplikation ebenfalls eine Komponente ein, welche die ausgeführten Schritte dokumentiert und bei Fehlern entsprechende Fehlermeldungen speichert. Außerdem erweiterten wir ein bestehendes Frontend und Backend, sodass alle Testresultate übersichtlich dargestellt werden.
In Kapitel 3 wird eine Übersicht über bestehende Softwarekomponenten, wie z. B. die GUI-Applikation, gegeben und ihre Funktionsweise dargestellt. Zuvor wird in Kapitel 2 auf die von den Softwarekomponenten und der im Rahmen dieser Arbeit entwickelten Testapplikation verwendeten Frameworks eingegangen. Die darauffolgenden Kapitel 4, 5 beschäftigen sich dann mit der Testapplikation.

Kapitel 2

Hintergrund

In den folgenden Unterkapiteln werden die von den Softwarekomponenten verwendeten Frameworks beschrieben. Zuerst wird das WPF-Framework erklärt, welches das Entwickeln von Software mit Benutzeroberfläche ermöglicht. ASP.NET Core stellt Werkzeuge zur Verfügung, um Backends und somit Web-APIs zu erstellen. Angular wiederum ist ein Framework zur Erstellung von Frontends, welche mit Backends kommunizieren. Das Entity Framework Core bietet eine Schnittstelle für den Zugriff auf Datenbanken in .NET. Appium und der WinAppDriver werden, wie in Kapitel 1 bereits erwähnt, für die Kommunikation mit der GUI verwendet. Zuletzt wird noch das NUnit-Framework erklärt, welches das Schreiben von Testfällen unterstützt.

2.1 Windows Presentation Framework

Diese Bachelorarbeit beschäftigt sich zwar mit dem Testen einer Applikation mit Benutzeroberfläche, jedoch wurden nur Änderungen vorgenommen, welche die Testbarkeit verbessern. Diese Änderungen hatten keinerlei Auswirkungen auf die grafische Darstellung der Benutzeroberfläche. Deshalb wird der Hintergrund über das Windows Presentation Framework (WPF) sehr kurz gehalten.
Das Windows Presentation Framework ermöglicht die Erstellung grafischer Benutzeroberflächen, welche dann mit einem .NET SDK auf unterschiedlichen Betriebssystemen ausgeführt werden können [4]. Obwohl eine WPF-Applikation komplett in C# erstellt werden kann, ist es Konvention, die Oberfläche mit der deklarative Sprache XAML zu erstellen. Dies hat den Vorteil, dass die Logik von der Oberfläche getrennt wird und die aktuelle Ansicht der Benutzeroberfläche jederzeit in einer Entwicklungsumgebung abgefragt werden kann. Die Elemente einer XAML-Datei repräsentieren .NET-Klassen, während die Attribute der Elemente Properties oder Events einer Klasse sind. Mit dieser XML-ähnlichen Sprache kann somit ein Baum aus .NET-Klassen aufgebaut werden. Wie wir in Abschnitt 2.5 zeigen, erlauben spezielle Attribute das Testen von Applikationen.

2.2 ASP.NET Core

Dieses Framework ermöglicht es, in .NET Webapplikationen oder Schnittstellen zu erstellen [9]. Es basiert auf dem Model-View-Controller-Prinzip, kurz MVC. Der Nutzer kommuniziert hier mit einer View, löst Anfragen an einen Controller aus, welcher wiederum mit einem Model kommuniziert. Da in dieser Bachelorarbeit nur mit Schnittstellen

gearbeitet wurde, werden die anderen Funktionalitäten von ASP.NET Core nicht weiter behandelt.
Bei Schnittstellen werden keine Views bereitgestellt; das V in MVC fällt also weg. Stattdessen wird auf HTTP-Anfragen von Single-Page-Applikationen mit Daten im JSON- oder XML-Format geantwortet. Dieses Konzept wird nun an einer beispielhaften Web-API besser veranschaulicht:
Diese Web-API gibt alle oder nur indizierte Wochentage zurück. Die Klasse `DaysController` stellt einen Controller dar. Außerdem wurde die Klasse mit dem `Route` Attribut und einer Route versehen. In dieser Route befinden sich die zwei Platzhalter `controller` und `action`. In diesem Fall bezeichnet `controller` den Klassenname `Days` und `action` einen Methodennamen. Für alle Methoden in dieser Klasse ist dies die Basisroute. Um auf Anfragen reagieren zu können, muss die Klasse Aktionsmethoden beeinhalten. Die Klasse `DaysController` beinhaltet die Objektmethoden `GetDay` und `GetAllDays`. Je nach Route wird dann die entsprechende Methode aufgerufen. Da bei der Methode `GetAllDays` nichts weiter spezifiziert wurde, ist die Route dieser Methode `Days/GetAllDays`. Die Methode `GetDay` wurde hingegen mit dem Attribut `HttpGet` [8] und einer neuen Route versehen . Dies bedeutet, dass die Methode nur bei einer GET-Anfrage aufgerufen wird. Der Parameter vom Typ `int` wird an den entsprechenden Teil in der Route gebunden. Wird in einer Aktionsmethode ein Parameter angegeben, sucht ASP.NET Core die entsprechenden Daten zuerst im Body der HTTP-Anfrage, danach in der URL und zuletzt in Query-Parametern, welche am Ende einer URL angehängt werden können. Als Rückgabetyp von Aktionsmethoden wird der Datentyp `IActionResult` angegeben, mit dem der Statuscode in der HTTP-Antwort manuell gesetzt werden kann. Das übergebene Objekt wird, wenn in der HTTP-Anfrage nichts anderes angegeben wird, per Default in JSON serialisiert.

```
1     [ApiController]
2     [Route("[controller]/[action]")]
3     public class DaysController : ControllerBase
4     {
5       private static readonly string[] Days = new[]
6       {
7           "Mo", "Tue", "We", "Thu", "Fr", "Sa", "Su"
8       };
9
10      [HttpGet("/[controller]/Day/{id}")]
11      public IActionResult GetDay(int id)
12      {
13        return Ok(Days[id]);
14      }
15
16      public IActionResult GetAllDays()
17      {
18        return Ok(Days);
19      }
20    }
```

Wird nun eine Anfrage an die Route `Days/GetAllDays` gesendet, wird richtigerweise die Methode `GetAllDays` ausgeführt und eine HTTP-Antwort mit dem Statuscode 200 und dem JSON-Array `["Mo","Tue","We","Thu","Fr","Sa","Su"]` zurückgesendet. Bei einer GET-Anfrage mit der Route `Days/Day/2`, wird im Hintergrund vom Array `Days` das zweite Element indiziert und somit der Wochentag `"We"` zurückgegeben.

2.3 Angular

Der Begriff Angular bezeichnet eine Plattform mit der Single-Page-Anwendungen (SPA) erstellt werden können [10]. Bei SPAs wird nur einmal eine HTML-Datei angefordert und die Ansicht danach mit JavaScript geändert. Angular stellt ein eigenes Kommandozeilen-Interface (CLI) und eine Menge an Bibliotheken zur Verfügung. Die CLI stellt dem Benutzer Befehle für das Erstellen, Kompilieren und Ausführen eines Projekts sowie für das Hinzufügen neuer Komponenten und externer Bibliotheken zur Verfügung. Angular verwendet TypeScript, eine Obermenge von JavaScript, welches Typinformationen bereitstellt. Dies erklärt den benötigten Kompilierschritt, welcher TypeScript in JavaScript umwandelt.

2.3.1 Komponenten

Angular teilt seine Funktionalität in Komponenten auf [10]. Komponenten bestehen jeweils aus einer TypeScript-, HTML- und CSS-Datei. In der TypeScript-Datei wird eine Klasse erstellt, welche mit dem `@Component` Attribut versehen wird. In diesem Attribut wird der Pfad zur HTML- und CSS-Datei angegeben, sowie ein Tagname, mit dem die grafische Darstellung dieser Komponente in anderen HTML-Dateien genutzt werden kann. Komponenten können Eingabeparameter entgegennehmen und Ergebnisse mithilfe von Events zur Verfügung stellen. Ihren Konstruktoren können Services übergeben werden, welche Angular mithilfe von Dependency Injection erstellt und der Komponente somit eine spezielle Funktionalität zur Verfügung stellt. Dependency Injection ist ein Mechanismus, bei dem gewünschte Objekte angefordert werden können, anstatt sie selbst erzeugen zu müssen. Während eines Applikationslaufs durchläuft eine Komponente einen Lebenszyklus. Um auf gewisse Zeitpunkte eines Lebenszyklus reagieren zu können, stellt Angular Lifecycle Hooks zur Verfügung. Dies sind Methoden, die zu speziellen Zeitpunkten aufgerufen werden. Ein Beispiel ist die `OnInit` Methode. Während im Konstruktor Inputs einer Komponente nicht verfügbar sind, kann in der `OnInit` Methode davon ausgegangen werden, dass die Komponente komplett initialisiert wurde.

2.3.2 Services

Wie bereits erwähnt, können in Angular Services benutzt werden [10]. Services sind TypeScript-Klassen, welche oft benötigte Logik zur Verfügung stellen. Diese Klassen werden mit dem Attribut `@Injectible` versehen, damit sie im Konstruktor einer Komponente per Dependency Injection erzeugt werden können. Ein wichtiges Service in Angular ist der `HttpClient`. Mit diesem Service können Daten von einer Web-API bezogen werden.

2.3.3 Routing

Um zwischen Komponenten wechseln zu können, bietet Angular eine Routing-Bibliothek an [10]. Hiefür wird ein Array mit Routen und den dazugehörigen Komponenten erstellt. Diese Routen werden dann beim Router registriert. In den HTML-Dateien kann dann das `routerLink` Attribut verwendet werden, um zu einer anderen Komponente zu wechseln.

2.4 Entity Framework Core

Das Entity Framework Core (EF Core) stellt eine Schnittstelle für den Datenaustausch mit relationalen Datenbanken zur Verfügung [9]. Hierbei wird vom Entwickler eine Klasse erstellt, welche die Datenbank repräsentiert. Dieser Klasse wird eine Verbindungszeichenfolge übergeben, welche Servername, Datenbank, Logindaten etc. beinhaltet. Außerdem beinhaltet diese Klasse für jede relationale Tabelle, welche durch sogenannte Plain Old CLR Objects (POJOs) abgebildet, eine Property. POJOs sind .NET Klassen, welche nicht von einer anderen Klasse erben oder Attribute aufweisen. In den folgenden Beispielen wird gezeigt, wie die verschiedenen Beziehungen in relationalen Datenbanken mithilfe von EF Core abgebildet werden.

2.4.1 1:1-Beziehung

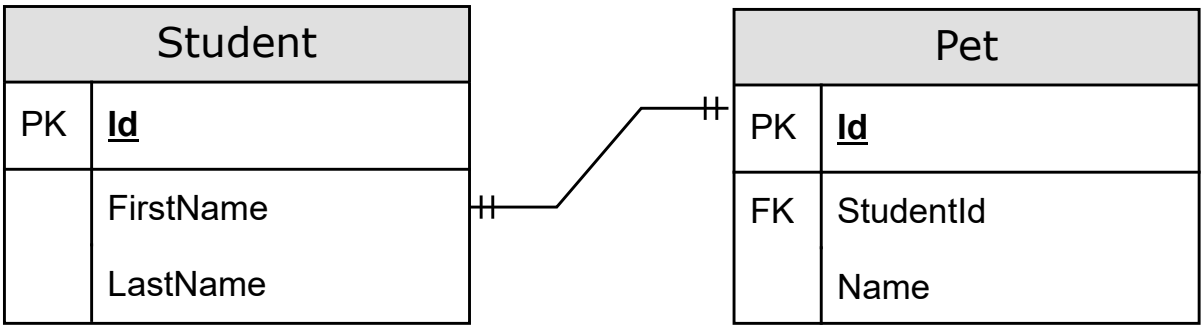


Abbildung 2.1: Beispiel einer 1:1-Beziehung

```
1     public class Student{
2         public int Id {get; set;}
3         public string FirstName {get; set;}
4         public string LastName {get; set;}
5         public Pet Pet {get; set;}
6     }
7
8     public class Pet{
9         public int Id {get; set;}
10        public string Name {get; set;}
11        public Student Student {get; set;}
12        public int StudentId {get; set;}
13    }
```

Abbildung 2.2: Darstellung einer 1:1-Beziehung in EF Core

Das Framework erkennt anhand der gegenseitigen Referenzierungen im Code 2.2, dass es sich hier um eine 1:1-Beziehung handelt. Die Tabellen werden wie im Bild 2.1 in der Datenbank abgespeichert.

2.4.2 1:n-Beziehung

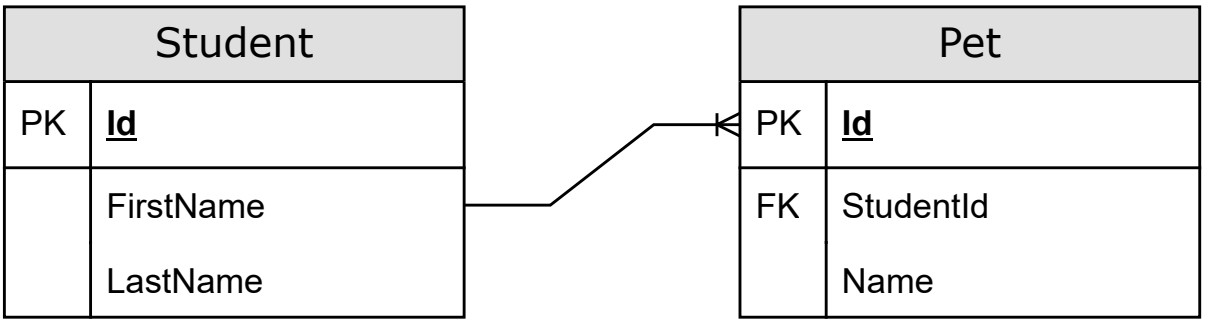


Abbildung 2.3: Beispiel einer 1:n-Beziehung

```
1     public class Student{
2         public int Id {get; set;}
3         public string FirstName {get; set;}
4         public string LastName {get; set;}
5         public List<Pet> Pets {get; set;}
6     }
7
8     public class Pet{
9         public int Id {get; set;}
10        public string Name {get; set;}
11        public Student Student {get; set;}
12        public int StudentId {get; set;}
13    }
```

Abbildung 2.4: Darstellung einer 1:n-Beziehung in EF Core

Das Framework erkennt anhand der Liste in der Klasse `Student` im Code 2.4, dass es sich hier um eine 1:n-Beziehung handelt. Die Tabellen werden wie im Bild 2.3 in der Datenbank abgespeichert.

2.4.3 m:n-Beziehung

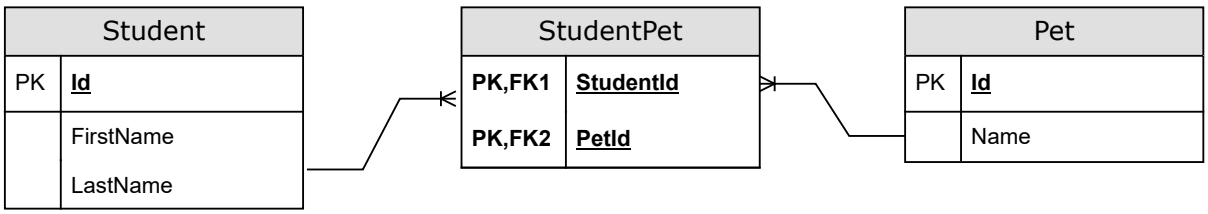


Abbildung 2.5: Beispiel einer m:n-Beziehung

```
1   public class Student{
2       public int Id {get; set;}
3       public string FirstName {get; set;}
4       public string LastName {get; set;}
5       public List<StudentPet> StudentPets {get; set;}
6   }
7
8   public class Pet{
9       public int Id {get; set;}
10      public string Name {get; set;}
11      public List<StudentPet> StudentPets {get; set;}
12  }
13
14  public class StudentPet{
15      public int StudentId {get; set;}
16      public Student Student {get; set;}
17      public int PetId {get; set;}
18      public Pet Pet {get; set;}
19  }
```

Abbildung 2.6: Darstellung einer m:n-Beziehung in EF Core

Das Framework erkennt anhand des Codes 2.6 in diesem Beispiel automatisch, dass es sich um eine m:n-Beziehung handeln muss. Die Tabellen werden wie im Bild 2.5 in der Datenbank abgespeichert.

2.5 Appium und WinAppDriver

Appium ist ein Framework, welches in ein .NET-Projekt eingebunden werden kann [1]. Mithilfe der verfügbaren Schnittstellen können GUI-Aktionen wie z. B. das Klicken oder Eingeben von Text simuliert werden. Die genaue Funktionsweise wird im folgenden Abschnitt erklärt.

2.5.1 Kommunikation zwischen Appium und WinAppDriver

Appium kommuniziert nicht direkt mit der GUI. Diese Aufgabe wird in Windows von dem WinAppDriver übernommen. Dieses Programm ist vergleichbar mit einem Server, welcher Anfragen entgegennimmt und diese an die GUI weiterleitet. Mithilfe der Kommandozeile kann der WinAppDriver gestartet werden. Der Treiber ist danach einsatzbereit und stellt eine REST-API zur Verfügung. Über diese REST-API kann Appium nun Nachrichten und Daten als JSON an den Server senden. In einer Applikation wird dann mit Appium ein Objekt erstellt, welches den Treiber darstellt. Diesem Objekt können mehrere Parameter, wie z. B. die GUI-Applikation, mitgegeben werden. Beim Erstellen des Objekts startet der WinAppDriver dann die zu testende Software. Auf diesem Treiber-Objekt können nun Methoden aufgerufen werden, welche die gewünschten Aktionen auf der GUI simulieren. Im Hintergrund sendet Appium HTTP-Anfragen an den Server.

2.5.2 Zugriff auf die Elemente einer GUI

Um über Appium Zugriff auf die einzelnen Elemente der GUI zu bekommen, ist es wichtig, den Elementen eindeutige Automation-IDs zu geben [7]. Dies kann in XAML mit den Attributen `Name` oder `AutomationProperties.AutomationId` bewerkstelligt werden. Während ersteres nur statische Werte erlaubt, können mit dem Attribut `AutomationProperties.AutomationId` auch dynamisch Werte spezifiziert und die ID erst zur Laufzeit ermittelt werden. Um den Inhalt eines Elements, wie z. B. von einem Button, auslesen zu können, muss entweder das Attribut `Content` oder `AutomationProperties.AutomationName` gesetzt werden. Beide Attribute erlauben dynamische Werte. In der Tabelle 2.1 befinden sich die wichtigsten Methoden für diese Bachelorarbeit.

Methode	Beschreibung
`AppiumDriver<T>.FindElementByAccessibilityID(string selector)`	Gibt das erste Element zurück, welches die im Parameter `selector` angegebene Automation-ID besitzt.
`AppiumDriver<T>.FindElementsByAccessibilityID(string selector)`	Gleich wie bei `FindElementByAccessibilityID`, nur dass alle Elemente zurückgegeben werden.
`RemoteWebElement.Click()`	Führt einen Klick auf ein Element, z. B. einen Button, aus.
`RemoteWebElement.GetAttribute(string attributeName)`	Liefert den Inhalt eines Attributes eines Elements zurück. Hiermit können angezeigte Texte abgefragt werden.
`RemoteWebElement.Enabled()`	Liefert einen Wahrheitswert zurück, der angibt, ob ein Element (z. B. ein Button) aktiviert ist.

Tabelle 2.1: Verwendete Methoden von Appium

2.5.3 Beispiel

Im folgenden Abschnitt wird nun eine beispielhafte Kommunikation zwischen Appium und dem WinAppDriver dargestellt. Ziel des Beispiels ist es, den Windows Texteditor zu starten und einen Text einzugeben. Wir starten also den WinAppDriver und geben in einer Applikation den Code 2.7 ein.

```
1    AppiumOptions appCapabilities = new AppiumOptions();
2    appCapabilities.AddAdditionalCapability("app",
     ↪  @"C:\Windows\System32\notepad.exe");
3    var winAppDriver = new WindowsDriver<WindowsElement>(new
     ↪  Uri("http://127.0.0.1:4723"), appCapabilities);
```

Abbildung 2.7: Kommunikationsaufbau zwischen Appium und WinAppDriver im Code

Es wird also ein neues Treiberobjekt mit der IP-Adresse, dem Treiberport und der

zu testenden Software als Parameter erzeugt. Beim Erzeugen des Objekts wird die Nachricht 2.8 an den Server gesendet.

```
1  POST /session HTTP/1.1
2  Accept: application/json, image/png
3  Connection: Keep-Alive
4  Content-Length: 152
5  Content-Type: application/json;charset=utf-8
6  Host: 127.0.0.1:4723
7  User-Agent: selenium/3.141.0 (.net windows)
8  X-Idempotency-Key: edc9ef0a-5969-4d28-9642-2806354d88c4
9
10 {"desiredCapabilities":{"app":"C:\\Windows\\System32\\notepad.exe","platfo⌋
   ↪  rmName":"Windows"},"capabilities":{"firstMatch":[{"platformName":"Wind⌋
   ↪  ows"}]}}
11 HTTP/1.1 200 OK
12 Content-Length: 141
13 Content-Type: application/json
14
15 {"sessionId":"DD82B987-664F-4BD9-8513-77E1382F1D32","status":0,"value":{"a⌋
   ↪  pp":"C:\\Windows\\System32\\notepad.exe","platformName":"Windows"}}
```

Abbildung 2.8: Kommunikationsaufbau zwischen Appium und WinAppDriver

Der WinAppDriver wird aufgefordert, eine neue Session mit der zu testenden Software zu erzeugen und uns die ID der erzeugten Session zurückzusenden. Diese Session-ID wird dann bei jeder Anfrage in der URL mitangegeben. Nun können wir auch schon Aktionen auf der GUI simulieren. Um Text in den Editor zu schreiben, muss zuerst die Automation-ID dieses Elements ermittelt werden. Hierzu nehmen wir ein von Windows zur Verfügung gestelltes Programm "inspect.exe" zur Hilfe. Bewegen wir nun den Cursor über das Textfeld, wird in der inspect.exe die Automation-ID 15 angezeigt (siehe Abbildung 2.10). Anstatt im Code danach zu suchen, wurden die Automation-IDs der GUI auf den Verkehrsautomaten ebenfalls mit diesem Schema ermittelt. Der Code 2.9 sucht nun nach diesem Element und übergibt ihm den gewünschten Input, welcher schlussendlich im Editor angezeigt wird.

```
1      var textField = winAppDriver.FindElementByAccessibilityId("15");
2      textField.SendKeys("This is an example");
```

Abbildung 2.9: Simulation einer Eingabe auf dem Windows Texteditor

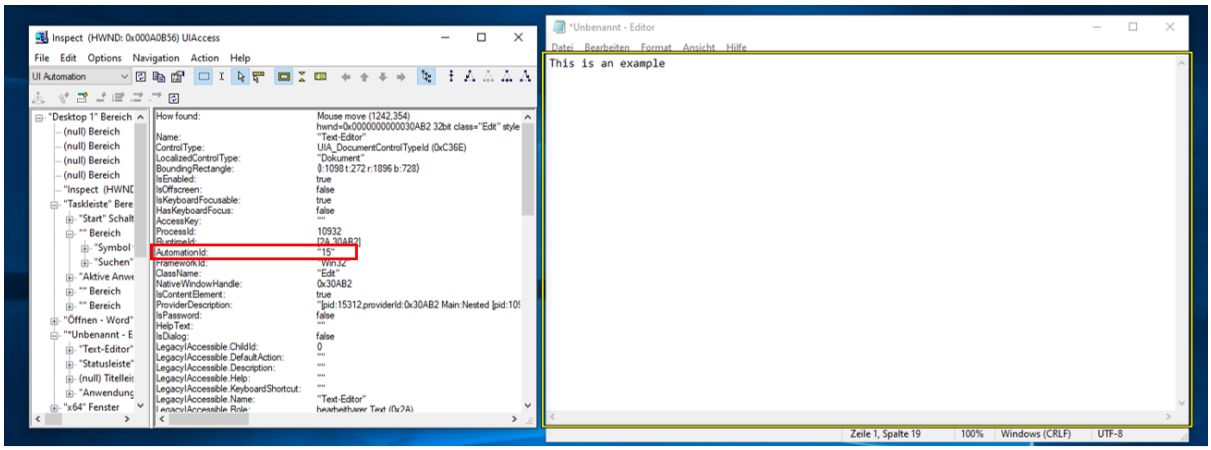


Abbildung 2.10: Auslesen einer Automation-ID mit der inspect.exe und Eingabe von Text in den Windows Texteditor

2.6 NUnit

In einer Applikation können Testfälle mithilfe des NUnit-Frameworks geschrieben werden [2]. Testfälle sind hierbei einfache Methoden, welche attribuiert werden. Welche Attribute benötigt werden und wie die Testfälle ausgeführt werden können, wird in den folgenden Abschnitten erklärt.

2.6.1 Attribute

Zum Erstellen von Testfällen muss eine Klasse mit Objektmethoden geschrieben werden [2]. Die Methoden müssen mit dem Attribut `Test` versehen werden. Soll eine Methode vor jedem Testfall ausgeführt werden, muss diese Methode mit dem Attribut `SetUp` annotiert werden. Für das Ausführen danach kann das Attribut `TearDown` verwendet werden. Um vor dem Ausführen aller Testmethoden eines Namensraums Code auszuführen, kann eine Klasse geschrieben werden, in der sich Objektmethoden mit den Attributen `OneTimeSetup` und/oder `OneTimeTeardown` befinden.

2.6.2 Ausführung

Um die Testmethoden auszuführen, kann Visual Studio und dessen Test Explorer verwendet werden. Eine Testapplikation kann aber auch mit der dotnet CLI gestartet werden. Um nun z. B. die Testapplikation TestApp.exe auszuführen, kann in der Kommandozeile der Befehl `dotnet test TestApp.exe` aufgerufen werden [6]. Mit der Option `filter` und einem Filterausdruck werden nur spezifische Testmethoden ausgeführt. Die genaue Funktionsweise wird im Kapitel 4.8 noch beschrieben.

Kapitel 3

Systemübersicht

In diesem Kapitel geben wir nun einen kurzen Überblick über bereits bestehende Softwarekomponenten, die im Rahmen dieser Bachelorarbeit benutzt oder erweitert wurden. Im Diagramm 3.1 wird zur besseren Veranschaulichung die Kommunikation zwischen diesen Komponenten dargestellt. Die WPF-Applikation wird als WPFClient bezeichnet.

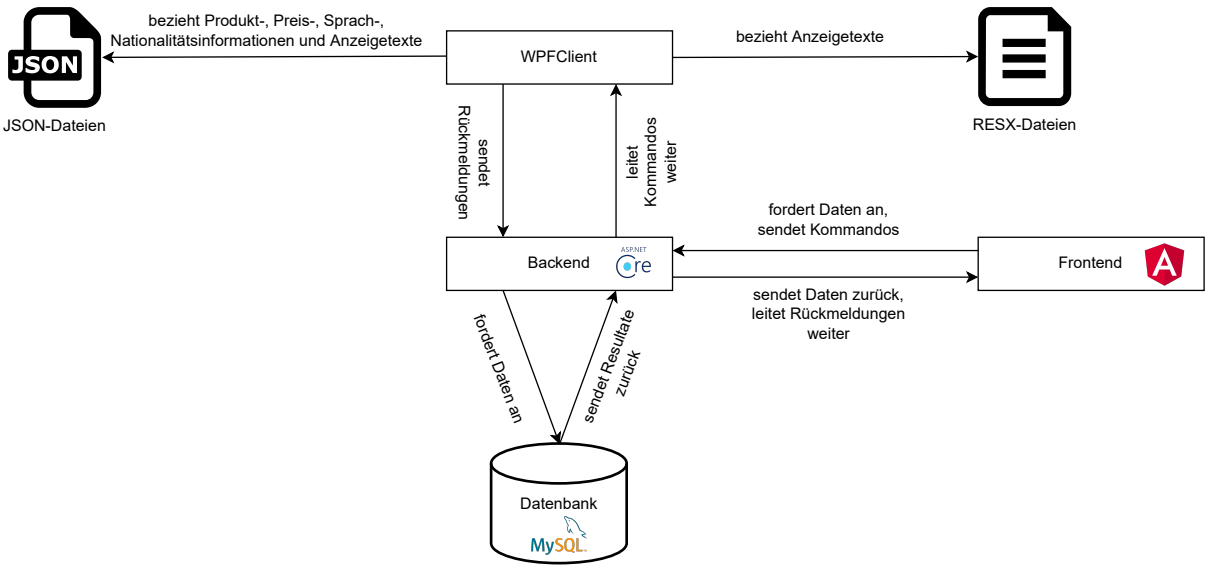


Abbildung 3.1: Systemübersicht

3.1 WPF-Applikation

Da sich diese Bachelorarbeit mit dem automatisierten Testen einer WPF-Anwendung beschäftigt, gehen wir vor der Implementierung in diesem Kapitel auf den Aufbau und die Funktionsweise dieser Software ein. Die Software besteht aus dem Willkommensbildschirm, Produktbildschirm, Kennzeichenbildschirm, Zusammenfassungsbildschirm, Zahlungsbildschirm und Druckbildschirm.

3.1.1 Willkommensbildschirm

Dieser Bildschirm agiert als Bildschirmschoner und erscheint, wenn gerade kein Kauf durchgeführt wird. Er gibt dem Kunden eine grobe Beschreibung, welche Produkte auf

diesem Automaten verkauft werden. Der Bildschirm erscheint automatisch, wenn nach einiger Zeit keine Tätigkeit am Automaten erkannt wird. Die Abbildung 3.2 zeigt den Willkommensbildschirm.

Abbildung 3.2: Willkommensbildschirm

3.1.2 Produktbildschirm

Nach einem Klick auf dem Willkommensbildschirm erscheint der Produktbildschirm. Hierbei erscheinen jedoch nicht gleich die Produkte, sondern ein kleines Fenster, mit dem der Kunde gefragt wird, ob er auch eine slowenische E-Vignette erwerben will. Auf dem Produktbildschirm kann dann zwischen Digitaler Vignette und Streckenmaut entschieden werden. Im oberen Bereich werden Buttons zum Wechseln der Sprache angezeigt. Außerdem wird eine Statusleiste angezeigt, die dem Kunden mitteilt, wie weit er mit dem Kauf schon fortgeschritten ist. Im unteren Bereich werden die Buttons "Weiter", "Zurück" und "Neustart" angezeigt. Falls auf diesem Automaten auch die slowenische E-Vignette angeboten wird, erscheint unten ebenfalls ein Button mit dem navigiert werden kann. Bei der Digitalen Vignette kann zwischen Pkw und Motorrad gewechselt werden. Ausgewählte Produkte der Digitalen Vignette werden bei einem Wechsel gelöscht. Nachdem der Kunde sich für ein Produkt entschieden hat, sieht er die Gesamtanzahl und den Gesamtbetrag in einem Warenkorb. Bei einigen Produkten ist es möglich, die Gültigkeitsdauer zu ändern. Die Abbildung 3.3 zeigt den Produktbildschirm.

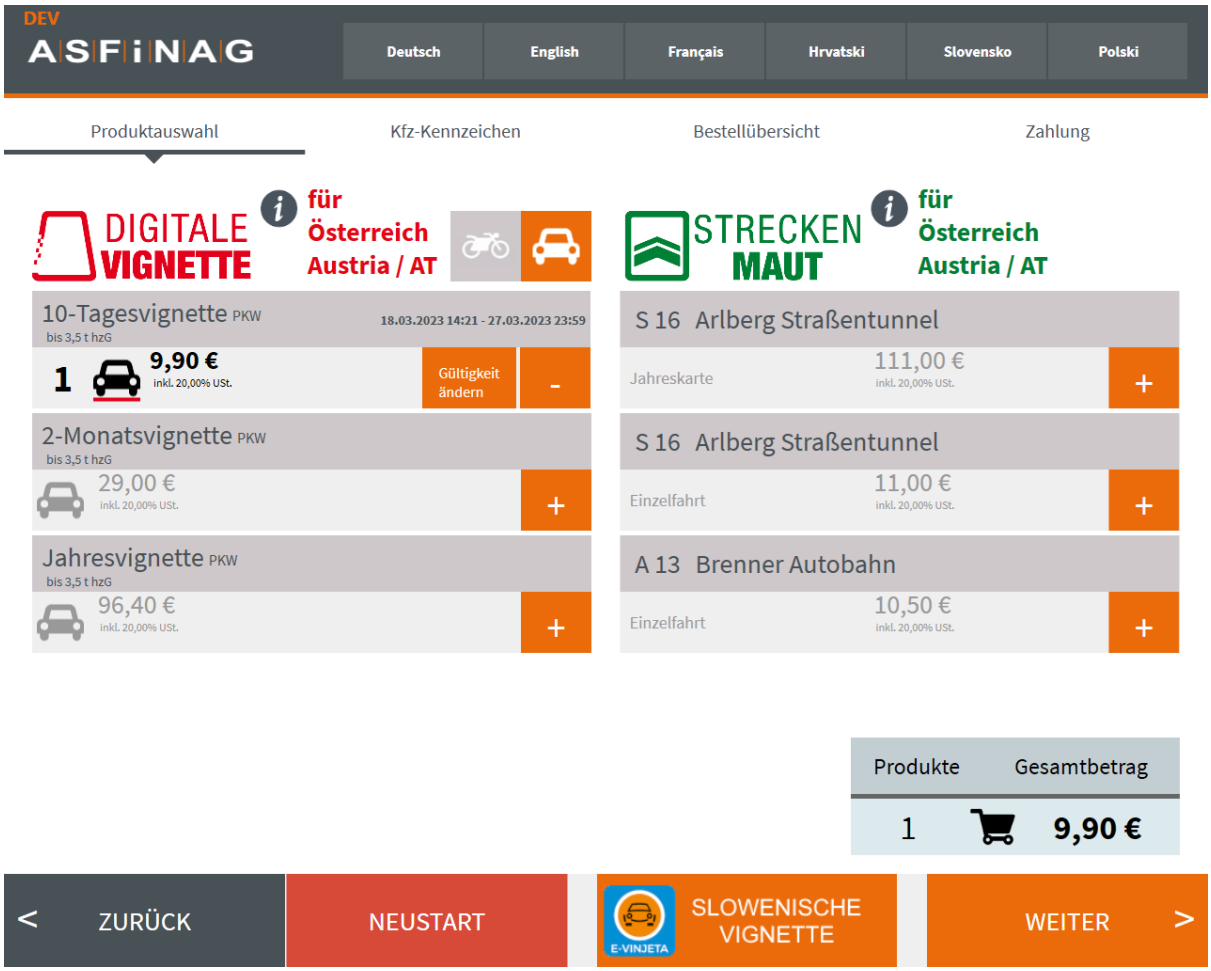


Abbildung 3.3: Produktbildschirm

3.1.3 Kennzeichenbildschirm

Nach einem Klick auf den Weiter-Button am Produktbildschirm erscheint der Kennzeichenbildschirm. Im Gegensatz zum Produktbildschirm kann hier nicht mehr die Sprache geändert werden. Auf diesem Bildschirm kann der Kunde nun die Nationalität seines Kennzeichens auswählen und dieses dann in einem Textfeld eingeben und bestätigen. Für die Eingabe des Kennzeichens erscheint am Bildschirm eine Tastatur. Bei den Nationalitäten werden nur die zuletzt am häufigsten verwendeten Nationalitäten angezeigt. Ist die entsprechende Nationalität nicht vorhanden, kann sich der Kunde über einen Button alle Nationalitäten anzeigen lassen. Wurde am Produktbildschirm eine Streckenmaut für ein Jahr erworben, bekommt der Kunde, falls er eine gültige Jahres-Klebevignette besitzt, eine Gutschrift ausgestellt. Die Abbildung 3.4 zeigt den Kennzeichenbildschirm.

Abbildung 3.4: Kennzeichenbildschirm

3.1.4 Zusammenfassungsbildschirm

Nach dem Produktbildschirm erscheint der Zusammenfassungsbildschirm. Falls am Produktbildschirm keine Streckenmaut ausgewählt wurde, erscheint auf diesem Bildschirm nochmals ein kleines Hilfsfenster mit einer Auswahl. Auf dem Zusammenfassungsbildschirm werden alle Produkte vom Warenkorb sowie das eingegebene Kennzeichen angezeigt. Der Kunde hat die Möglichkeit, Produkte zu löschen. Ein Produkt, welches n-mal ausgewählt wurde, wird am Zusammenfassungsbildschirm auch n-mal untereinander angezeigt. Die Abbildung 3.5 zeigt den Zusammenfassungsbildschirm.

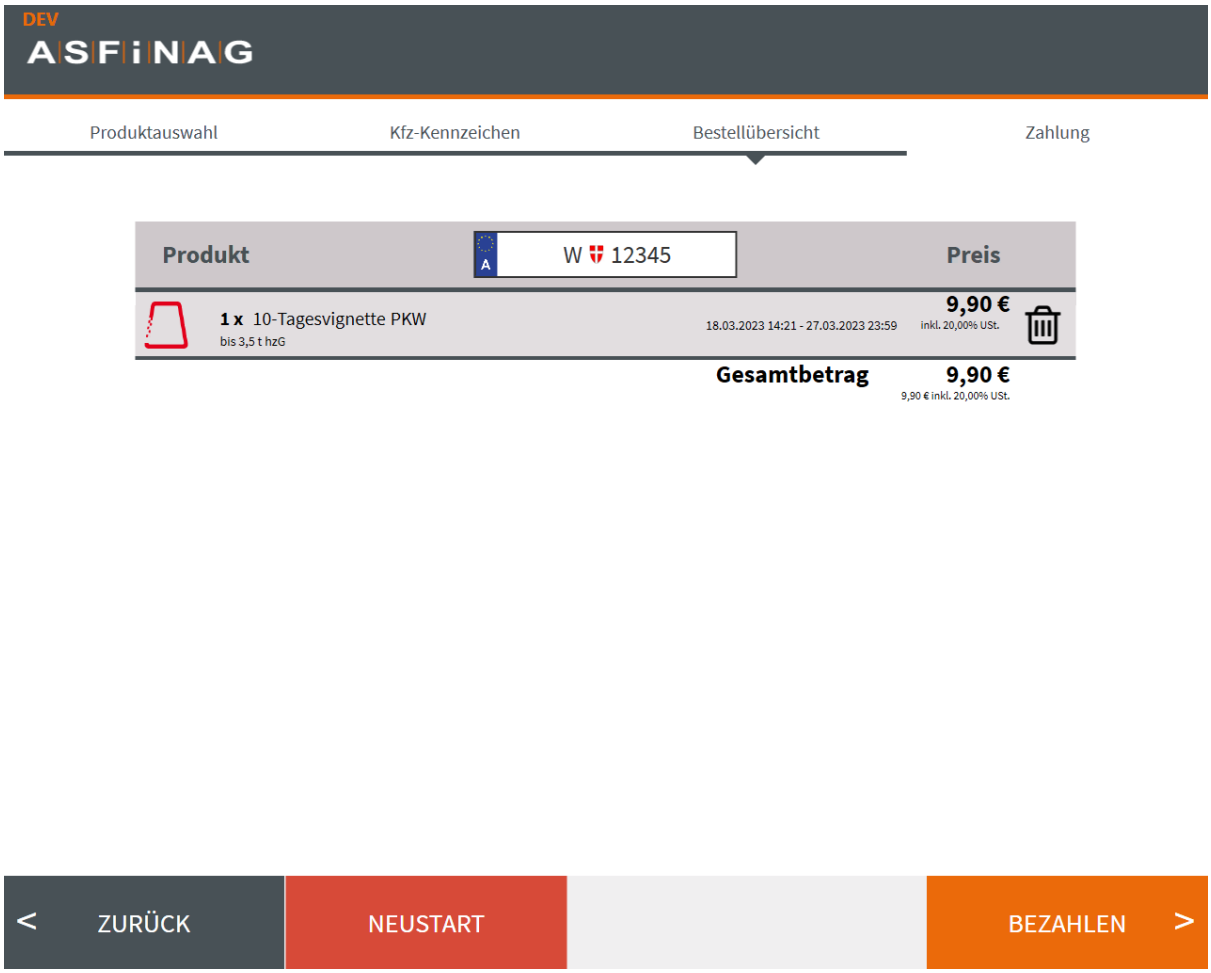


Abbildung 3.5: Zusammenfassungsbildschirm

3.1.5 Zahlungsbildschirm

Fährt der Kunde am Zusammenfassungsbildschirm fort, wird der Zahlungsbildschirm angezeigt. Der Kunde wird aufgefordert seine Zahlungsmethode bekanntzugeben und den angezeigten Betrag zu zahlen. Die Abbildung 3.6 zeigt den Zahlungsbildschirm.

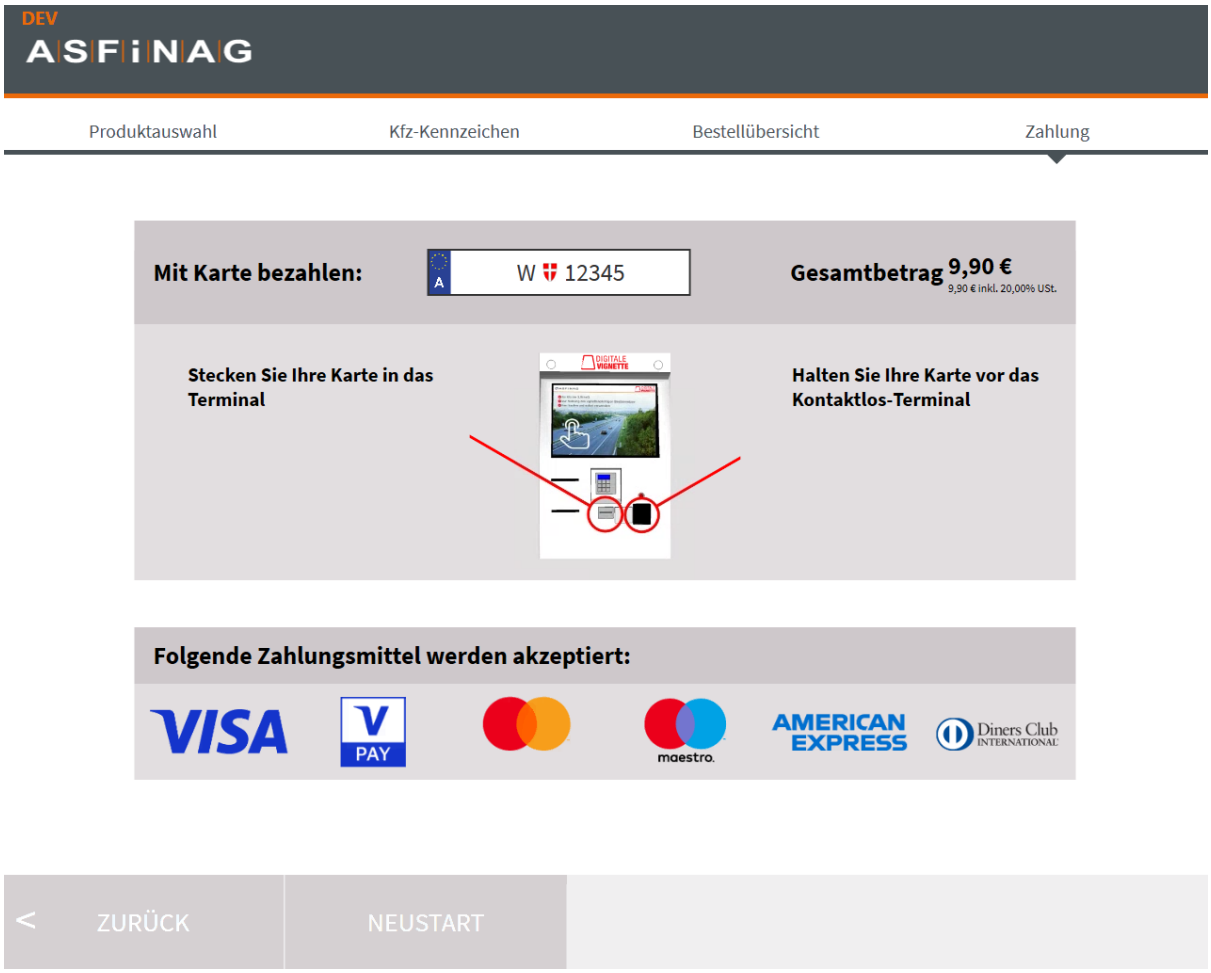


Abbildung 3.6: Zahlungsbildschirm

3.1.6 Druckbildschirm

Dies ist der letzte Bildschirm der GUI und wird angezeigt, wenn die Zahlung erfolgreich durchgeführt wurde. Der Kunde wird darum gebeten, den Zahlungsbeleg aufzubewahren. Mit einem Klick auf den Neustart-Button wird zum Willkommensbildschirm zurücknavigiert. Die Abbildung 3.7 zeigt den Druckbildschirm.

Abbildung 3.7: Druckbildschirm

3.1.7 Ablauf Produkteinkauf

Ein Produkteinkauf kann nun folgendermaßen aussehen:

1. Klick auf den Willkommensbildschirm
2. Auswahl der Sprache Französisch
3. 2-Monats-Vignette zum Warenkorb hinzufügen
4. Änderung des Gültigkeitsstarts der 2-Monats-Vignette
5. Klick auf den Weiter-Button
6. Auswahl der Nationalität Österreich
7. Eingabe des Kennzeichens "W-12345"
8. Bestätigung des Kennzeichens "W-12345"
9. Klick auf den Weiter-Button
10. Kontrolle der ausgewählten Produkte

11. Klick auf den Weiter-Button
12. Auswahl des Zahlungsmittels Visa
13. Zahlung des geforderten Betrags
14. Entnahme der Rechnung

Wie später in der Arbeit noch genauer beschrieben wird, dokumentiert der Testbericht diese einzelnen Schritte mit.

3.1.8 JSON- und RESX-Dateien

Alle auf den Bildschirmen angezeigten Daten bezieht die GUI von JSON- und RESX-Dateien. In der "Product.json" sind beispielsweise alle Produkte enthalten, die von der ASFINAG verkauft werden. Da jedem Produkt Anzeigetexte zugeordnet sind, gibt es diese Datei mehrmals in unterschiedlichen Sprachen. Auf den Verkehrsautomaten wird immer nur der Teil dieser Produkte zum Kauf angeboten, welche in der Datei "Price.json" angegeben werden. Neben dem Preis bezieht die GUI auch die Gültigkeitsdauern aus dieser Datei. Da in dieser Datei keine Anzeigetexte vorkommen, ist sie auch nur einmal vorhanden. Die Datei "Language.json" beinhaltet alle Sprachen, welche am Produktbildschirm ausgewählt werden können. Diese Datei existiert ebenfalls nur einmal. Für die Nationalitäten am Kennzeichenbildschirm existieren mehrere Dateien für die unterschiedlichen Sprachen. Alle anderen Texte, die der Automat in unterschiedlichen Sprachen anzeigen kann, werden in RESX-Dateien gespeichert. Für jede Sprache gibt es auch hier jeweils eine Datei. RESX-Dateien sind XML-Dateien, in denen Schlüssel-Werte-Paare abgespeichert werden. Ein Textelement der Software kann somit mit einem eindeutigen Schlüssel den Anzeigetext in der richtigen Sprache beziehen. Um Änderungen in Zukunft schneller vornehmen zu können, werden diese Dateien bei der neuen GUI in der Datenbank gespeichert.

3.2 Frontend

Um die Zustände der einzelnen Automaten zu verfolgen oder Befehle an die Automaten zu senden, gibt es ein bestehendes Frontend, welches mit Angular (siehe Abschnitt 2.3) erstellt wurde. In diesem Frontend können auch Daten aus der Datenbank dargestellt und bearbeitet werden.
Um eine Tabelle aus der Datenbank formatiert und übersichtlich dazustellen, kann die Komponente `DataTable` verwendet werden, welche Teil einer bestehenden Bibliothek ist. Diese Komponente kann über Parameter konfiguriert werden und stellt die entsprechenden Daten aus der Datenbank grafisch dar.

3.3 Backend

Angular fordert Daten von einem Backend an. Das Backend greift auf die Datenbank zu und sendet die Daten an das Frontend zurück. Das Backend dient aber auch als Bindeglied zwischen Frontend und WPFClient.

3.3.1 API

Um Automaten Befehle zu erteilen oder spezielle Daten aus der Datenbank zu bekommen, gibt es im Backend mehrere, mit dem `ApiController` Attribut versehene, Controller mit unterschiedlichen Methoden. Diese können bei Bedarf auch erweitert werden. Die Klasse `DataModelController` sorgt dafür, dass die Komponente `DataTable` weiß, wie die Daten darzustellen sind. Die darin definierte Aktionsmethode gibt bei ihrem Aufruf ein Objekt vom Typ `DataModel` zurück. In diesem Objekt können die Darstellungsreihenfolge der Spalten einer Datenbanktabelle, die Beziehungen zwischen Tabellen und Änderungen der darzustellenden Spaltennamen eingetragen werden. Die Komponente `DataTable` holt sich diese Metainformationen aus dem Backend und stellt die Tabelle übersichtlich dar (siehe Abschnitt 3.2).

3.3.2 OData

Die Komponente `DataTable` bezieht ihre Daten nicht direkt von den vorhin erwähnten Controllern, sondern verwendet eine in ASP.NET Core konfigurierbare OData-Schnittstelle. OData stellt CRUD (Create, Read, Update, Delete) Operationen zur Verfügung [5, 3]. Um dies in ASP.NET Core zu bewerkstelligen, muss für jede Datenbanktabelle, die zugreifbar sein soll, ein Controller erstellt werden, welcher von der `ODataController` Klasse erbt. In dieser Klasse werden dann die entsprechenden Aktionsmethoden mit den Attributen `HttpGet`, `HttpPost`, `HttpPut`, `HttpDelete` annotiert. Je nach HTTP-Methode wird die dementsprechende Aktionsmethode aufgerufen [8].

Kapitel 4

Testapplikation

In diesem Kapitel wird nun die Testapplikation vorgestellt, welche die GUI testet. In den folgenden Kapiteln wird der Ausdruck "WPFClient" als Synonym für die GUI verwendet.

4.1 Aufbau

In dieser Bachelorarbeit bezeichnen wir mit dem Begriff Testapplikation immer einen Teil eines .NET-Projekts. In diesem Projekt befinden sich auch andere Tests, welche jedoch nicht die Funktionalität der Benutzeroberfläche überprüfen und deshalb nicht Teil dieser Arbeit sind.
Der Aufbau und die grundsätzliche Funktionalität der Testapplikation kann aus der Abbildung 4.1 entnommen werden. Jeder Testfall greift auf eine GUI-Schnittstelle zu. Dies ist ein Objekt, welches einem Testfall alle möglichen Funktionalitäten der GUI zur Verfügung stellt. Die Konfigurationen werden über das Backend von der Datenbank gelesen und in einem Objekt gespeichert, welches diese Werte dann an die GUI-Schnittstelle weitergibt. Die GUI-Schnittstelle kommuniziert direkt mit dem WinAppDriver und simuliert Aktionen auf der GUI. Diese Aktionen speichert sie dann in der Testfalldokumentation. Merkt die GUI-Schnittstelle, dass sich Soll- und Ist-Zustand unterscheiden, wird dies in der Dokumentation als Fehler vermerkt und der Test abgebrochen. Auf Details zur GUI-Schnittstelle gehen wir im Laufe des Kapitels ein.

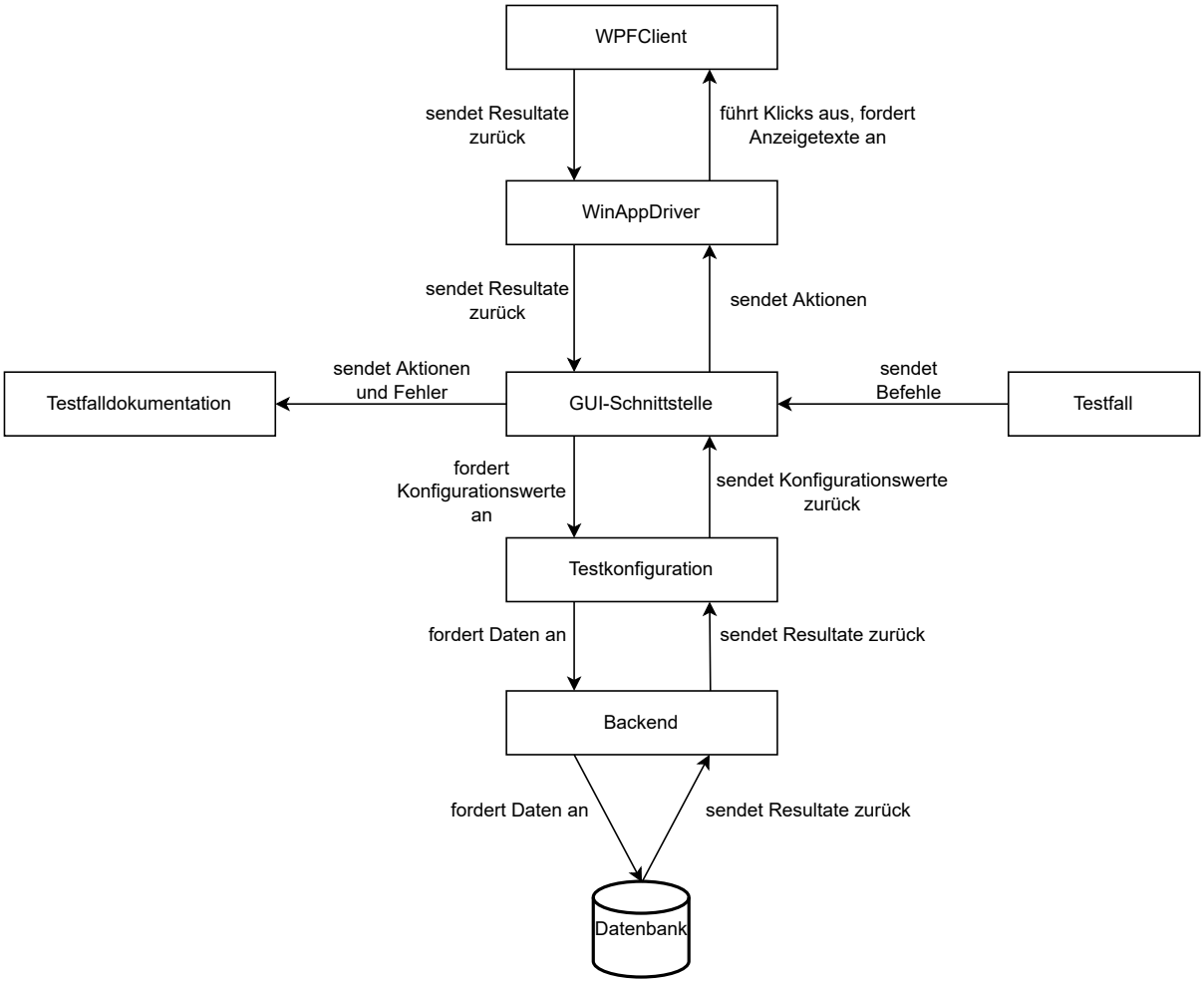


Abbildung 4.1: Aufbau der Testapplikation

4.2 Zugriff auf die Datenbank

Bevor in diesem Abschnitt erklärt wird, wie die Testapplikation auf die Datenbank zugreift, wird zuerst das Datenbankschema vorgestellt. In der Abbildung 4.2 ist ein Teil des Schemas der Datenbank zu sehen, welcher speziell für das Testen eine wichtige Rolle spielt.

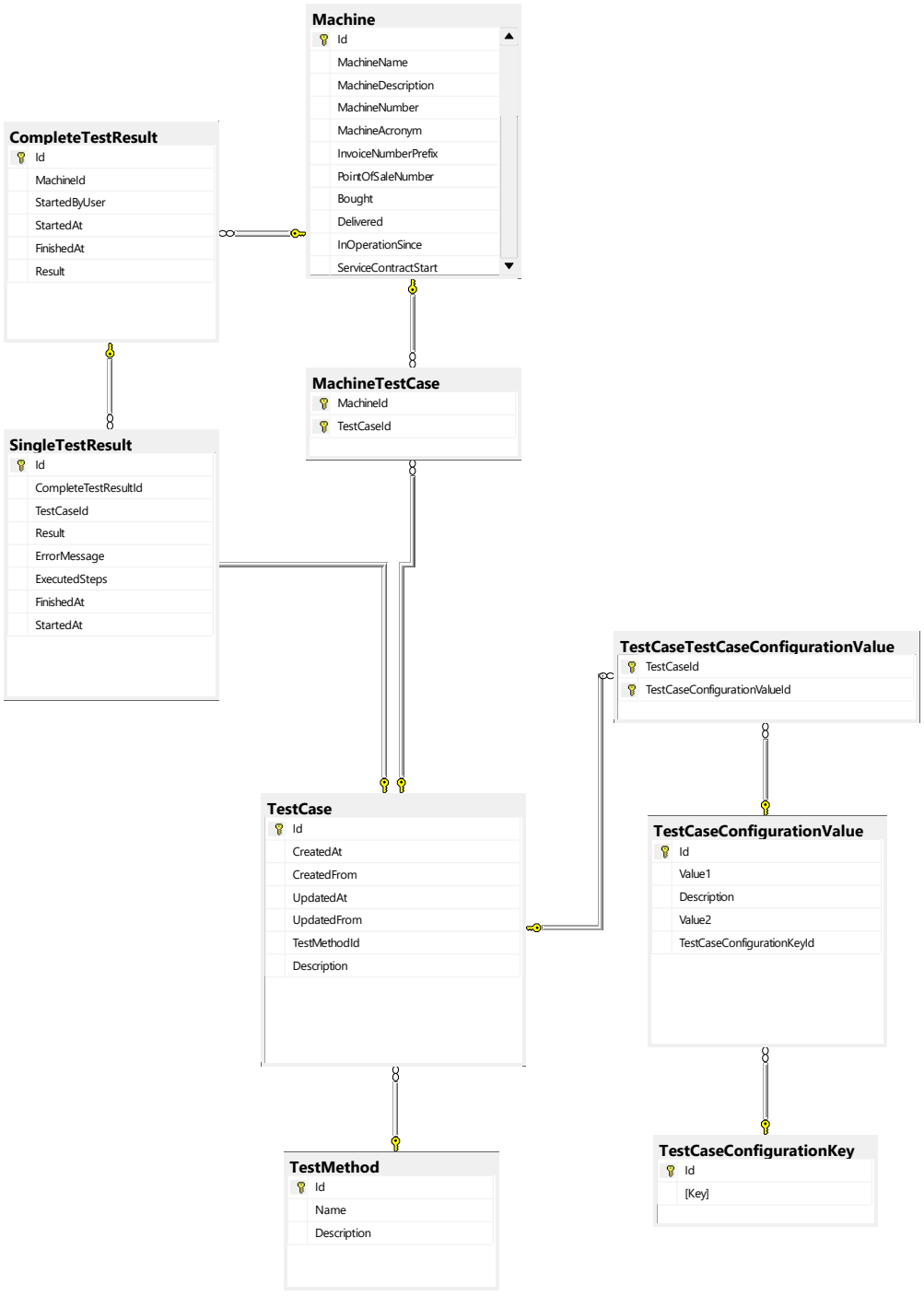


Abbildung 4.2: Ausschnitt aus dem Datenbankschema

Jeder Automat mit einem installierten WPFClient kann in die Tabelle `Machine` eingetragen werden. Für das Testen ist in dieser Tabelle allerdings nur der eindeutige Maschinenname von Bedeutung. Mit einer Funktion, welche den Automatennamen zurückgibt, kann nach diesem Tabelleneintrag gefiltert werden.
Die `TestCase` Tabelle speichert Informationen zu jedem Testfall. Neben Erstelldatum und Ersteller wird auch die ID einer Testmethode gespeichert. Die damit referenzierte Testmethode repräsentiert den tatsächlichen Code, welcher bei diesem Testfall ausgeführt wird. Mehrere Testfälle können die gleiche Testmethode haben, sie unterscheiden sich jedoch in den Konfigurationswerten. Zwischen der Tabelle `Machine` und `TestCase` besteht eine m:n-Beziehung. Jeder Maschine können somit mehrere Testfälle und ein Testfall mehreren Maschinen zugewiesen werden.
Um dem Testfall mitzuteilen, welche Produkte, Sprache, Nationalität, Kennzeichen etc. er am WPFClient auszuwählen hat, werden ihm mehrere Konfigurationswerte zugeteilt. Diese Werte werden in der Tabelle `TestCaseConfigurationValue` gespeichert. In dieser Tabelle besteht jeder Tabelleneintrag aus einem Schlüssel und zwei Werten. Soll zum Beispiel ein Produkt am WPFClient angeklickt werden, werden der entsprechende Konfigurationsschlüssel sowie die Produkt-ID und die Anzahl als Werte angegeben. Mithilfe des EF Core (siehe Abschnitt 2.4) kann auf dieses Schema zugegriffen werden.
Für die Testdokumentation wird in der Tabelle `SingleTestResult` für jeden durchgeführten Testfall ein Tabelleneintrag mit dem Resultat, den durchgeführten Schritten, Start- und Endzeit etc. erstellt. Da jeder durchgeführte Testfall Teil eines kompletten Testlaufs mit zahlreichen Testfällen ist, referenziert jeder Tabelleneintrag in der Tabelle `SingleTestResult` das Ergebnis eines kompletten Testlaufs, welcher in der Tabelle `CompleteTestResult` gespeichert ist. Ein kompletter Testlauf wird wiederum einem Automaten zugeordnet.
Um diese Daten über das Backend verfügbar zu machen, mussten wir die API erweitern. Hierfür erstellten wir eine neue Klasse `TestAutomationController`. In diesem Controller gibt es für jede Tabelle eine Aktionsmethode, welche mittels EF Core auf die Datenbank zugreift und alle Tabelleneinträge als JSON zurückgibt. Für das Speichern von Daten, wie z. B. in der Tabelle `SingleTestResult`, mussten wir Aktionsmethoden für POST-Anfragen erstellen. Die Route aller Aktionsmethoden in dieser Klasse ist `api/[controller]/[action]`. Jeder Methodenname besteht aus dem Plural des Tabellennamens, aus dem die Methode die Daten bezieht. Die Route für die Daten der Tabelle `CompleteTestResult` ist somit `api/TestAutomation/CompleteTestResults`. Daten können nun über GET-Anfragen angefordert und über POST-Anfragen gespeichert werden.
Damit sich die Testapplikation nicht um HTTP-Anfragen kümmern muss, implementierten wir die Klasse `TestAutomationService`. In dieser befindet sich für jede Aktionsmethode in der Klasse `TestAutomationController` eine Objektmethode, welche eine HTTP-Anfrage an das Backend sendet. Da es bereits ähnliche Klassen gibt, musste der Code für das Senden einer HTTP-Anfrage nicht mehr selbst implementiert werden.
Der Testapplikation steht nun also eine Schnittstelle zur Verfügung, mit der sie auf die Datenbank zugreifen kann. In der Testapplikation selbst erstellten wir noch die statische Klasse `DatabaseManager`. Diese Klasse lädt zu Beginn jede Tabelle mithilfe eines Objekts der Klasse `TestAutomationService` und speichert die Daten intern ab. Somit werden die Daten zwischengelagert und müssen nicht jedesmal über das Backend abgefragt werden.

4.3 Konfigurationswerte

In der Tabelle `TestCaseConfigurationValues` werden ein Schlüssel und zwei Werte als Tabelleneinträge gespeichert. Da die beiden Werte in der Datenbank als Texte abgespeichert sind, müssen sie in der Testapplikation zum erwarteten Datentyp konvertiert werden. In der Tabelle 4.3 wird gezeigt, welche Datentypen die Testapplikation bei den einzelnen Schlüsseln erwartet.

Schlüssel	Wert1	Wert2
InputAdditionalProductPopup	int	int
InputDarsEVignettePromotion	bool	
InputExistingVignettePromotion	bool	
InputExistingVignetteBarcode	string	bool
InputLanguageID	string	
InputLicensePlateID	string	int?
InputNationalityID	string	
InputProductID	int	int
InputProductScreenDeleteProductID	int	int
InputSummaryScreenDeleteProductID	int	int
InputValidityDateID	string	

Abbildung 4.3: Konfigurationsmöglichkeiten eines Testfalls

Der Schlüssel `InputAdditionalProductPopup` wird angegeben, wenn im Testfall erwartet wird, dass am Zusammenfassungsbildschirm das Pop-up 4.4 erscheint:

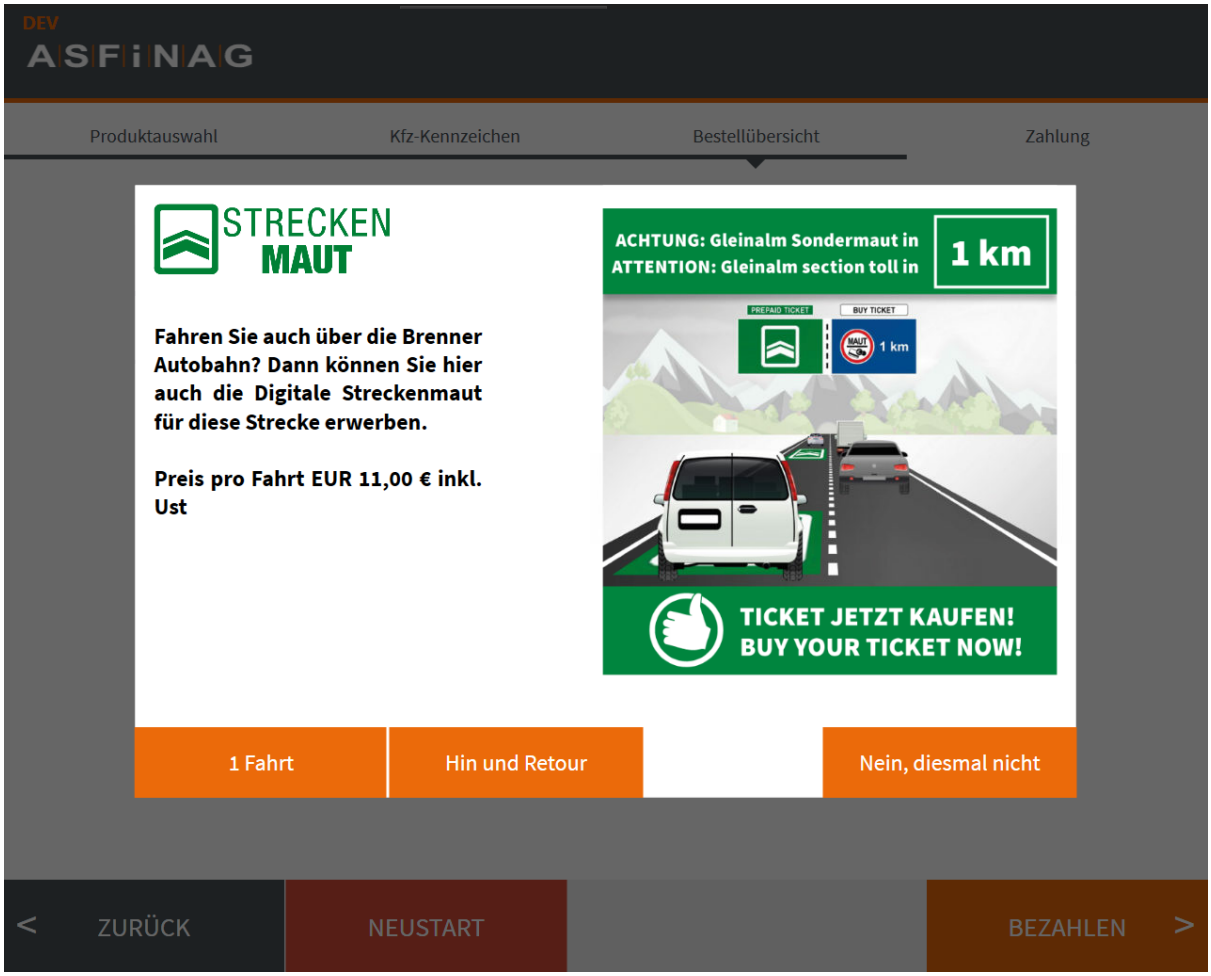


Abbildung 4.4: Erinnerung für den Erwerb einer Streckenmaut

Als Wert1 wird die Produkt-ID und als Wert2 die Anzahl angegeben. Somit wird bei einer Anzahl von 0 der rechte Button, bei einer Anzahl von 1 der linke Button und bei einer Anzahl von 2 der mittlere Button geklickt.

Der Schlüssel `InputDarsEVignettePromotion` muss bei jedem Testfall angegeben werden. Der Wert1 beschreibt einen Wahrheitswert, welcher den Button angibt, der am Pop-up 4.5 geklickt werden soll. Dieses Pop-up erscheint immer.

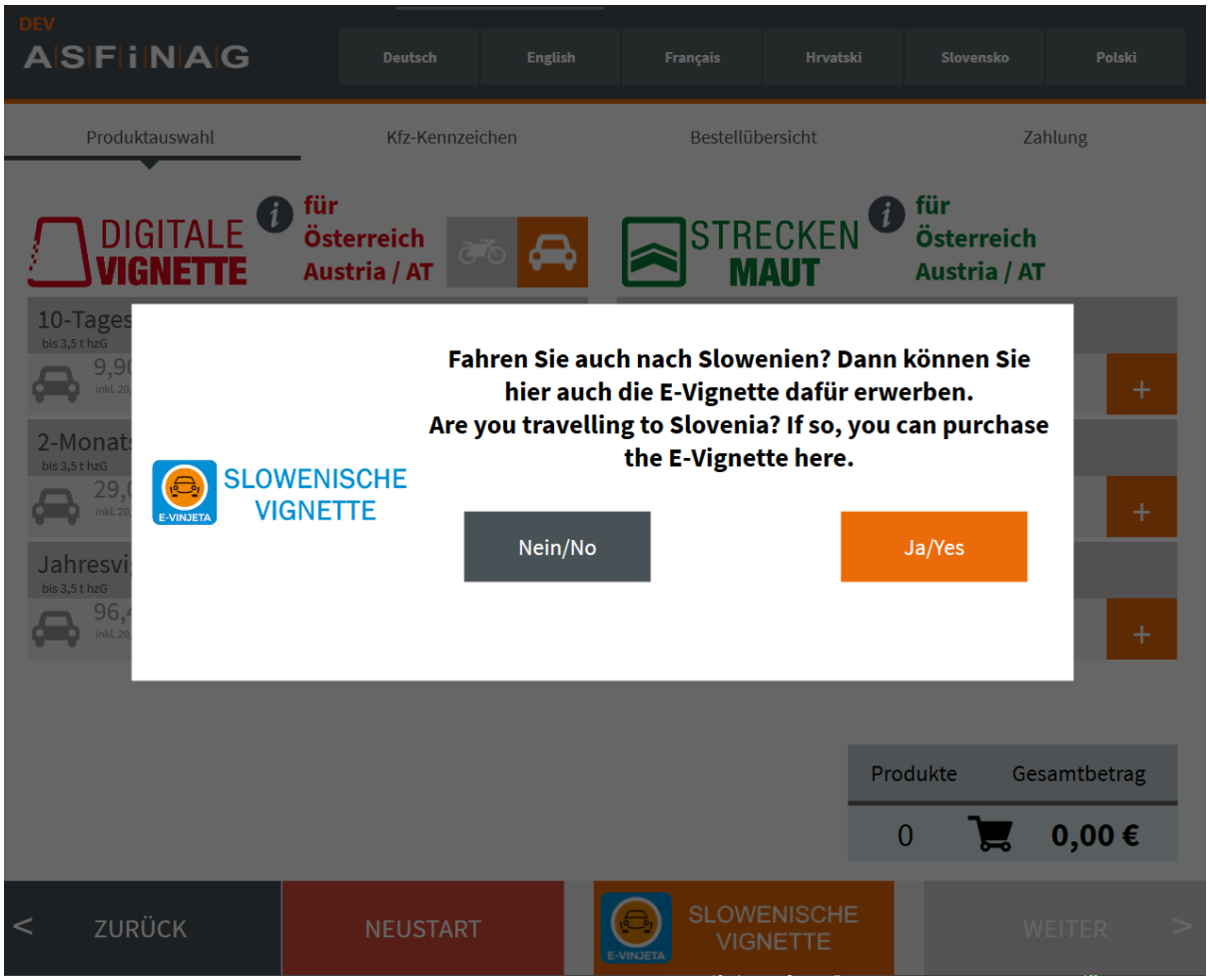


Abbildung 4.5: Abfrage, ob der Kunde eine slowenische E-Vignette erwerben will

Der Schlüssel `InputExistingVignettePromotion` wird angegeben, wenn im Testfall erwartet wird, dass am Kennzeichenbildschirm das Pop-up 4.6 erscheint.

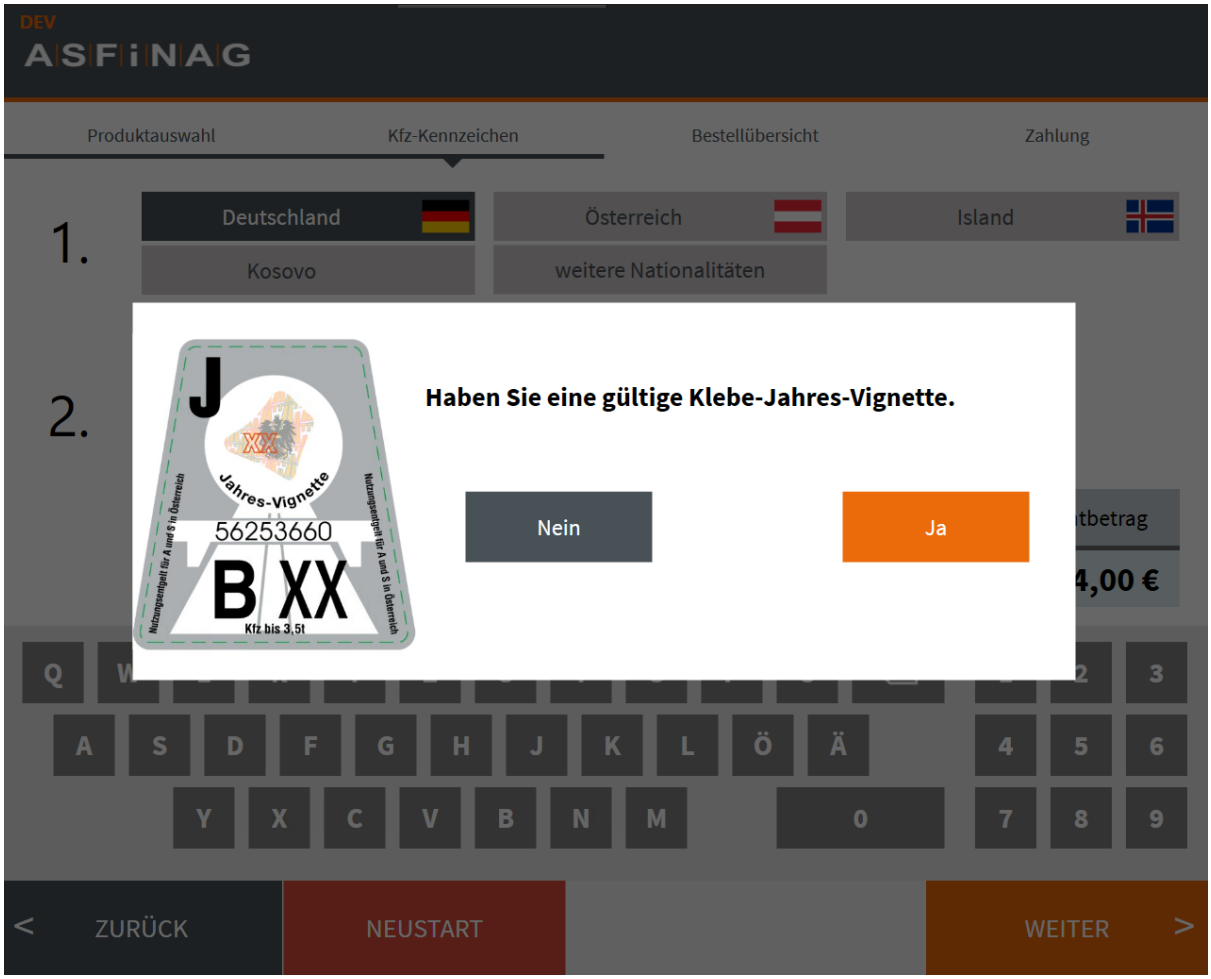


Abbildung 4.6: Abfrage, ob der Kunde eine gültige Klebe-Jahresvignette besitzt

Ist der Wahrheitswert im Wert1 auf wahr gesetzt, muss einem Testfall auch gleichzeitig ein Tabelleneintrag mit dem Schlüssel `InputExistingVignetteBarcode` zugeordnet werden. Bei diesem wird als Wert1 der Barcode angegeben, welcher am Bildschirm 4.7 eingegeben werden soll.

Abbildung 4.7: Eingabemöglichkeit eines Barcodes einer gültigen Klebe-Jahresvignette

Der Wahrheitswert im Wert1 gibt an, ob der Barcode vom WPFClient angenommen oder abgelehnt wird.
Die auszuwählende Nationalität am Kennzeichenbildschirm wird durch den Schlüssel `InputNationalityID` angegeben. Im Wert1 steht die eindeutige Kurzbezeichnung der Nationalität.
Beim Schlüssel `InputLanguageID` wird als Wert1 die Kurzbezeichnung einer Sprache angegeben, welche am Produktbildschirm ausgewählt werden soll.
Der Schlüssel `InputLicensePlateID` hat als Wert1 die ID eines Kennzeichens, welches am Kennzeichenbildschirm eingegeben werden soll. Bei deutschen oder österreichischen Kennzeichen kann es nun vorkommen, dass das Pop-up 4.8 erscheint.

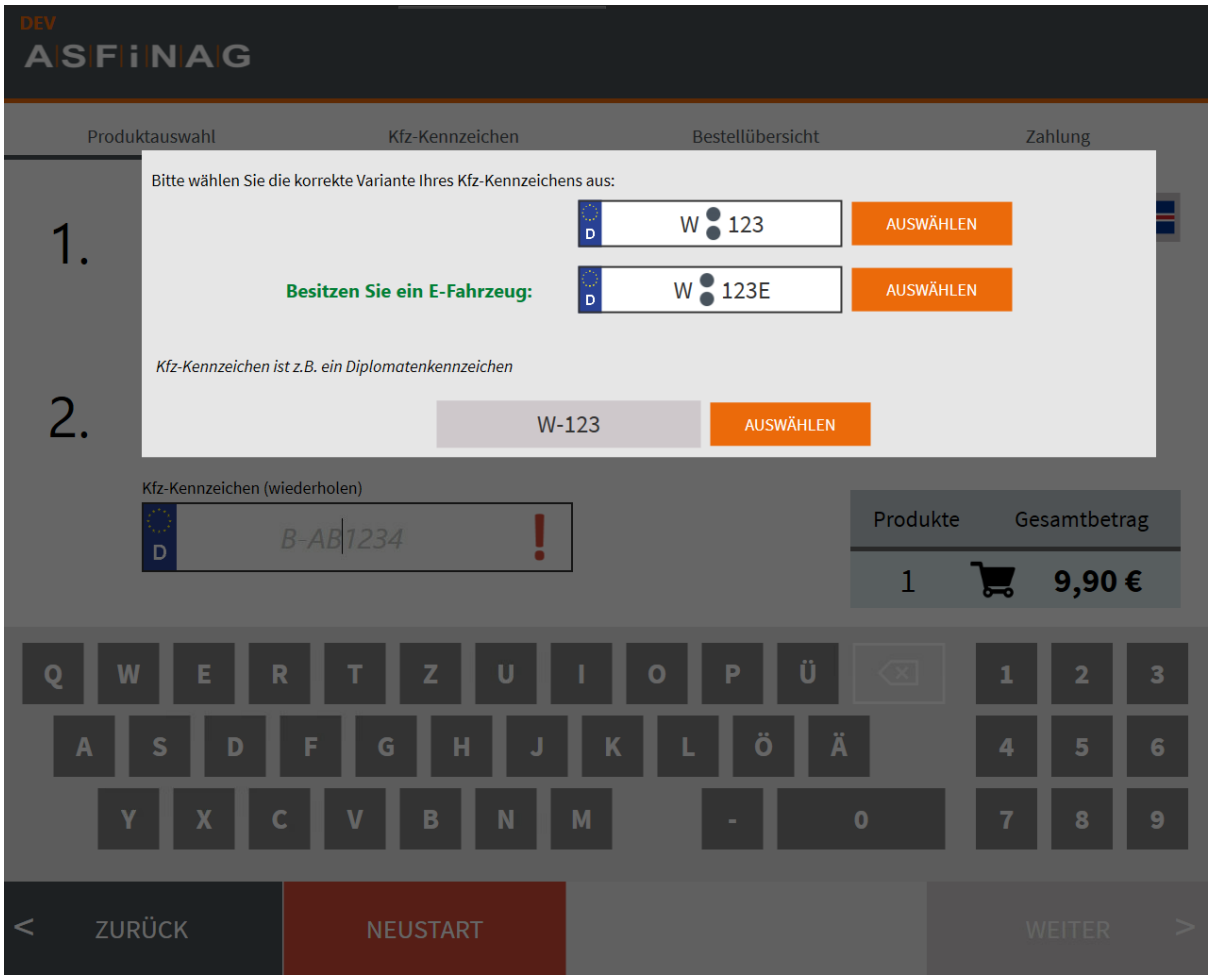


Abbildung 4.8: Auswahlmöglichkeit zwischen mehreren Kennzeichen

Der Tabelleneintrag eines Kennzeichens in der Datenbank gibt bereits an, ob dieses Pop-up erscheint und ob es sich um ein normales oder spezielles Kennzeichen handelt. Das spezielle Kennzeichen ist in diesem Fall die letzte Auswahlmöglichkeit. Da es mehrere Vorschläge für das normale Kennzeichen geben kann, wird im Wert2 der Index beginnend bei 0 eingetragen. Da das Pop-up nicht zwingend erscheint, kann der Wert2 auch leer sein.
Beim Schlüssel `InputProductID` wird als Wert1 die Produkt-ID und als Wert2 die Anzahl des Produkts angegeben. Der Testfall weiß damit, wie oft er ein Produkt am Produktbildschirm anklicken soll. Ähnliches gilt für den Schlüssel `InputProductScreenDeleteProductID`, mit dem der Testfall weiß, wie oft er Produkte am Bildschirm wieder löschen soll. Da am Zusammenfassungsbildschirm auch Produkte gelöscht werden können, gibt es zusätzlich den Schlüssel `InputSummaryScreenDeleteProductID`. Nach Auswahl mancher Produkte ist es am Produktbildschirm möglich, den Gültigkeitsbeginn zu ändern. Beim Schlüssel `InputValidityDateID` wird als Wert1 die Produkt-ID und als Wert2 der neue Gültigkeitsbeginn im Format "dd.MM.yyyy" angegeben.
Für das Einlesen der Konfigurationswerte eines Testfalls entwickelten wir eine Methode, welche diese über das Backend anfordert und sie als Liste zurückgibt. Für die Verarbeitung dieser Liste erstellten wir die Klasse `ConfigurationKeyValuePairs`. Für jeden Schlüssel gibt es in dieser Klasse eine Property. Falls beim Eingeben der Konfigurationswerte ein Fehler passiert ist, wird dies im Testbericht vermerkt (dieser wird

in Abschnitt 4.5 behandelt). Im Testfall muss somit nur noch ein Objekt der Klasse `ConfigurationKeyValuePairs` mit der Liste der Konfigurationswerte als Parameter erstellt werden. Dieses Objekt stellt im Bild 4.1 die Testkonfiguration dar.

4.4 Abbildung der Bildschirme

Um die einzelnen Bildschirme des WPFClient testen zu können, mussten zuerst die Automation-IDs der Elemente ausgelesen werden. Dafür verwendeten wir das Programm "inspect.exe". Da in XAML nicht bei allen Elementen die Automation-ID gesetzt war, mussten wir fehlende Automation-IDs ergänzen. Einige Elemente besaßen aufgrund des statischen Attributs `Name` die gleichen Automation-IDs. Diese wurden durch das Attribut `AutomationProperties.AutomationId` ersetzt und mit eindeutigen IDs versehen (siehe Abschnitt 2.5). Die Automation-IDs am Produktbildschirm sind in der Abbildung 4.9 sichtbar.

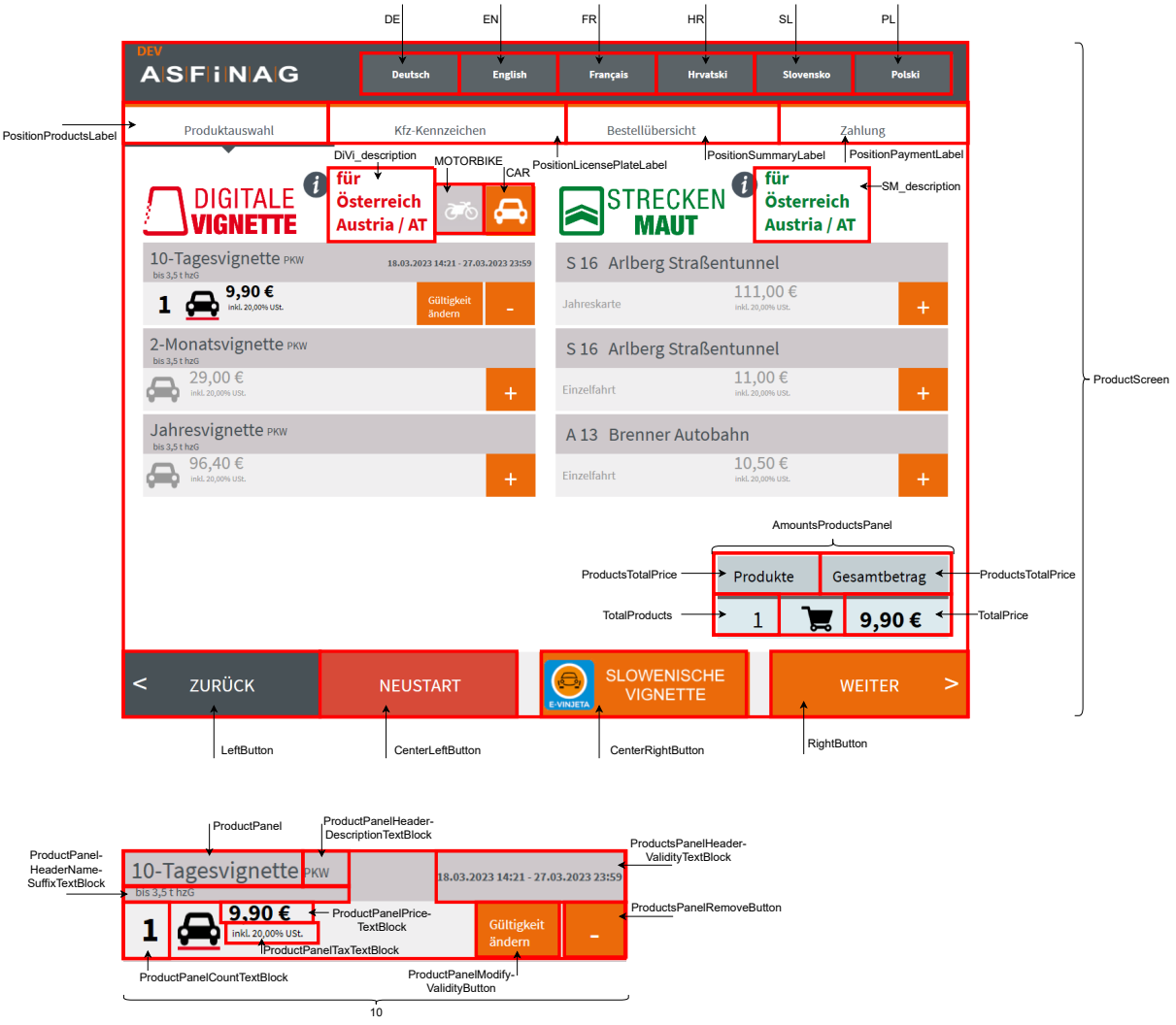


Abbildung 4.9: Automation-IDs am Produktbildschirm

Jeder Bildschirm der GUI, außer dem Willkommensbildschirm, lässt sich in einen oberen, mittleren und unteren Teil gliedern. Im Falle des Produktbildschirms befinden sich

im oberen Teil die Buttons zum Ändern der Sprache sowie die Statusleiste. Im unteren Teil befinden sich vier Buttons für das Navigieren zwischen den Bildschirmen. Der obere und untere Teil ist bei jedem Bildschirm in ähnlicher Ausprägung vorhanden. Im mittleren Teil des Produktbildschirms befindet sich ein Bereich für den Kauf einer Digitalen Vignette sowie ein Bereich für den Kauf einer Streckenmaut. In jedem dieser Bereiche sind die einzelnen Produkte aufgelistet. Bei der Digitalen Vignette kann mithilfe von zwei Buttons zwischen Pkw und Motorrad gewechselt werden. Außerdem wird am unteren Bereich ein Einkaufskorb mit der Gesamtanzahl an Produkten und dem Gesamtpreis angezeigt. Die Funktionalität des Produktbildschirms ist in der von uns entwickelten GUI-Schnittstelle gekapselt. Die GUI-Schnittstelle ist ein Objekt der Klasse `WPFClientApplication`. Die Abbildung 4.10 zeigt einen Teil des Klassendiagramms der GUI-Schnittstelle.

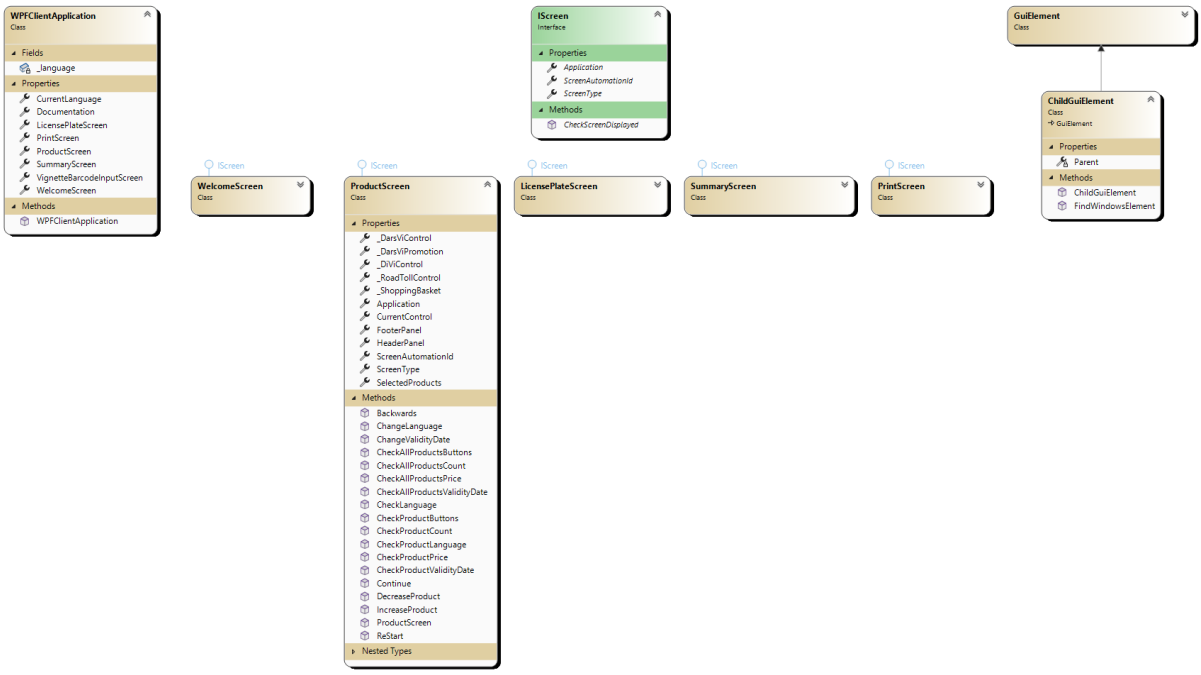


Abbildung 4.10: Ausschnitt aus dem Klassendiagramm der GUI-Schnittstelle

Ein Objekt der Klasse `WPFClientApplication` enthält alle Bildschirme des WPF-Client als Objekte. Während die Klasse `WPFClientApplication` nur Methoden dieser Objekten aufrufen muss, übernehmen die Objekte die eigentliche Kommunikation mit dem WinAppDriver und simulieren Aktionen auf der GUI. Für den Produktbildschirm wurde beispielsweise eine eigene Klasse erstellt. Diese Klasse erbt vom Interface `ISource` eine Methode, welche überprüft, ob am WFPClient der entsprechende Bildschirm angezeigt wird. Die Klasse enthält außerdem als Properties die vorhin erwähnten Bereiche (oben, unten, mittel). Diese Properties haben als Typen wiederum Klassen, welche die Funktionalität des Bereichs kapseln. Der obere Teil des Bildschirms wird z. B. durch die Property `HeaderPanel` dargestellt, der untere Teil durch den `FooterPanel`. In der Grafik 4.11 ist nun ersichtlich, durch welche Klassen die einzelnen Teile des Produktbildschirms abgebildet werden.

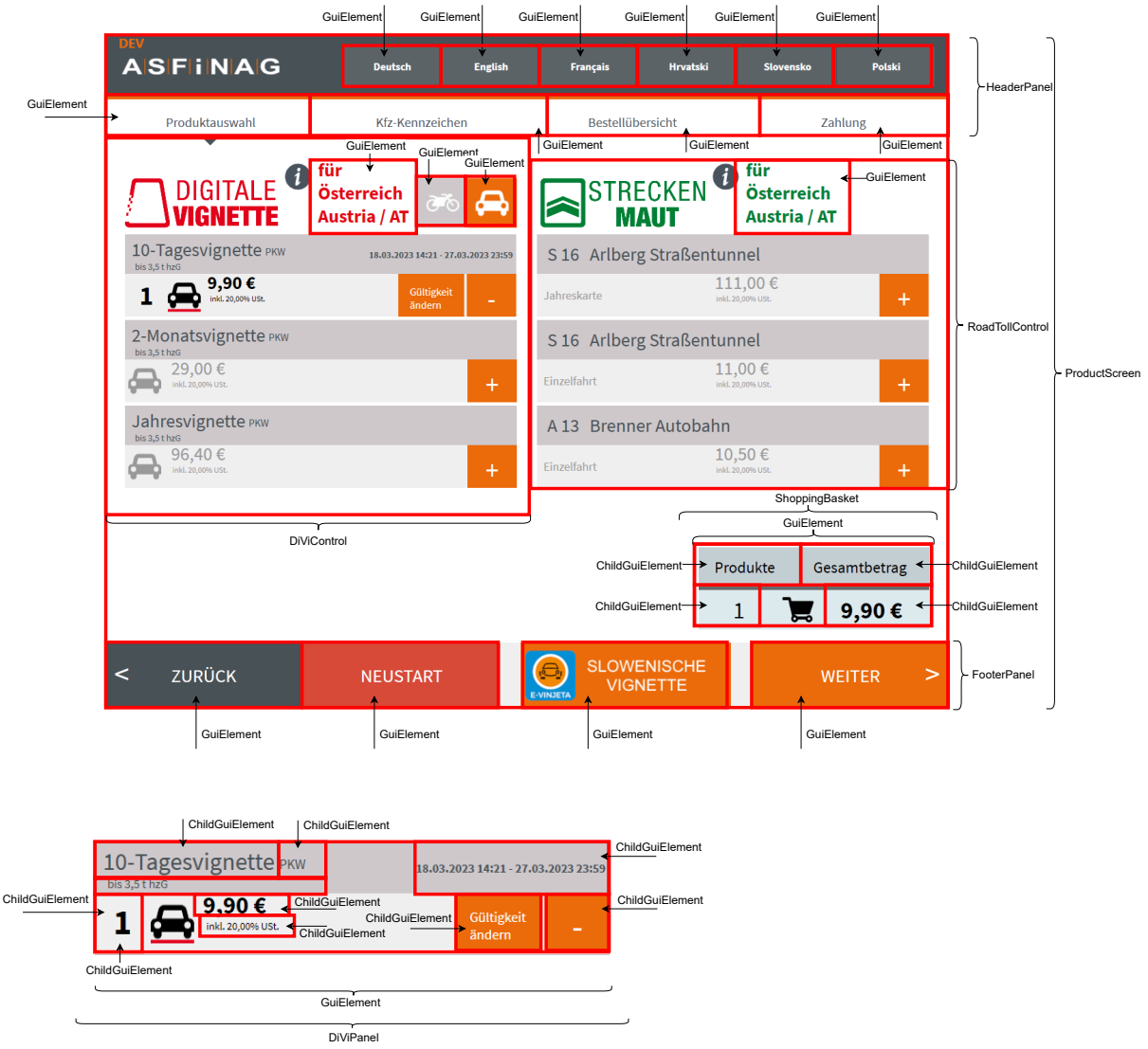


Abbildung 4.11: Darstellung des Produktbildschirms als Klasse

Der `HeaderPanel` und `Footerpanel` beinhalten Felder vom Typen `GuiElement`. Diese Klasse `GuiElement` wurde von uns für Basiselemente wie z. B. Buttons und Textfelder erstellt und stellt Funktionen für das Klicken von Buttons, sowie für das Auslesen von Texten zur Verfügung. Der Bereich für die Digitale Vignette wird durch die Klasse `DiViControl` repräsentiert. Sie beinhaltet neben den Buttons zum Wechseln des Fahrzeugtyps auch noch eine Liste von Produkten. Für diese Produkte entwickelten wir die Klasse `DiViPanel`. Da die Automation-IDs der einzelnen Elemente einer Produktleiste auf dem WPFClient nur innerhalb dieser eindeutig sind, können diese Elemente nicht einfach durch die Klasse `GuiElement` repräsentiert werden. Die Klasse `GuiElement` würde einfach nach dem Element mit der entsprechenden Automation-ID suchen und dieses zurückliefern. In diesem Fall würde immer ein Element aus der Produktleiste der 10-Tages-Vignette zurückgeliefert werden. Deshalb erstellten wir die Klasse `ChildGuiElement`, welche von der Klasse `GuiElement` erbt. Sie überschreibt die Methode zum Finden des Elements auf dem WPFClient. Im Konstruktor wird dieser Klasse ein Elternobjekt vom Typ `GuiElement` übergeben. Somit besteht eine Eltern-Kind-Beziehung. In diesem Elternobjekt ist das Kindobjekt dann eindeutig identifizierbar. Beim Suchen nach dem Element am Bildschirm wird zuerst nach dem Element des Elternobjekts und

mit diesem dann das Element des Kindobjekts gesucht. Die Klasse `GuiElement` verfügt außerdem über eine Methode zum Überprüfen, ob die richtige Sprache angezeigt wird. Damit ein Objekt jedoch weiß, woher es die erwartete Sprache beziehen soll, wird einem `GuiElement` beim Instanziieren ein Delegate mitgegeben. Dieses Delegate gibt bei Aufruf mithilfe der erstellten, statischen Klasse `JsonResxManager` den erwarteten Anzeigetext zurück. Die Klasse `JsonResxManager` liest die JSON- und RESX-Dateien in der aktuellen Sprache aus und stellt die Anzeigetexte über Methoden zur Verfügung. Die GUI-Schnittstelle muss diese Klasse somit informieren, wenn die Sprache am WPFClient geändert wird. Bei der neuen GUI werden die Anzeigetexte dann von der Klasse `DatabaseManager` zur Verfügung gestellt (siehe Abschnitt 3.1.8). Der Zustand des WPFClient wird intern von der GUI-Schnittstelle gespeichert. Somit kann z. B. am Zusammenfassungsbildschirm überprüft werden, ob die richtigen Produkte angezeigt werden. Mit dem gleichen Konzept wurden auch die anderen Bildschirme umgesetzt.

4.5 Testbericht

Startet ein Testfall, holt er sich über eine Methode in der Klasse `TestReport` ein neues Objekt der Klasse `SingleTestDocumentation` (in Abbildung 4.1 als Testdokumentation ersichtlich). Dieses Objekt übergibt der Testfall dann der GUI-Schnittstelle. Mit diesem Objekt kann die GUI-Schnittstelle alle möglichen Aktionen aufzeichnen. In der Klasse `Action` befinden sich dafür alle Aktionstexte als Konstanten. Die durchgeführten Aktionen werden in der Dokumentation in einer Liste gespeichert. Das Notieren von Fehlern wird durch die unterschiedlichen `GuiElement` Objekte vorgenommen (siehe Abschnitt 4.4). Ihnen wird im Konstruktor der entsprechende Fehlertext aus der Klasse `Error` für unterschiedliche Fehlerarten übergeben. Nach Ende eines Tests wird für das Ergebnis, die durchgeführten Schritte, die Fehlernachricht etc. ein Tabelleneintrag in der Tabelle `SingleTestResult` erstellt. Die einzelnen Schritte werden hierbei durch ein Semikolon getrennt als Text gespeichert. Nach Ende eines kompletten Testlaufs wird in der Tabelle `CompleteTestResult` ebenfalls ein Eintrag erzeugt. In diesem ist das Ergebnis positiv, falls kein Testfall fehlgeschlagen ist. Ansonsten ist das Ergebnis negativ. Die Abbildung 4.12 zeigt einen Teil des Klassendiagramms der Testdokumentation.

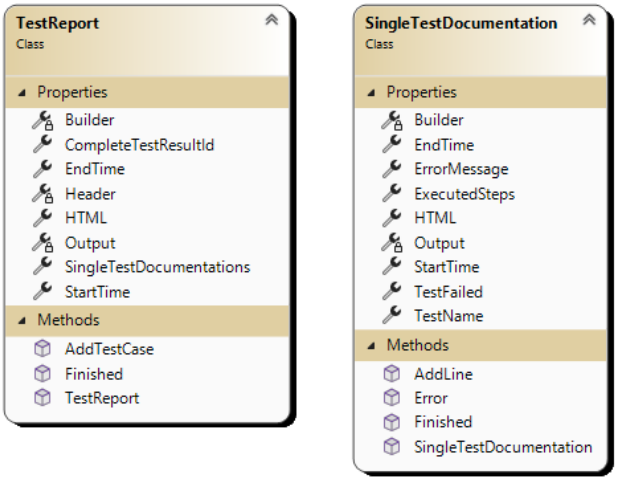


Abbildung 4.12: Ausschnitt aus dem Klassendiagramm der Testdokumentation

4.6 Versenden von E-Mails

Nach jedem Testlauf wird an alle interessierten und mit einer E-Mail-Adresse hinterlegten Personen eine E-Mail mit einer Kurzzusammenfassung der Ergebnisse versendet. Das Senden dieser E-Mails mit dem Testbericht als Inhalt wird durch eine statische Methode ermöglicht, welche eine bestehende Bibliothek verwendet. Beim Senden einer E-Mail wird vom Testbericht eine Kurzzusammenfassung als HTML angefordert. In dieser Kurzzusammenfassung sind die Ergebnisse aller ausgeführten Testfälle dargestellt. Ein Link in der Mail führt zu einer Webseite mit genaueren Details (siehe Abschnitt 5). Bild 4.13 zeigt eine E-Mail für einen Testlauf, bei dem alle Testfälle ohne Fehler beendet wurden, betrachtet werden.

Haas Michael EXTERN

Von:
Gesendet: Donnerstag, 2. März 2023 18:21
An: Haas Michael
Betreff: Testbericht

Der Testlauf startete am 02.03.2023 18:14:47 und endete am 02.03.2023 18:20:57 mit 9 erfolgreichen und 0 fehlgeschlagenen Testfällen.

Eine komplette Übersicht finden Sie unter Ergebnisse Testfälle.

Die folgenden Tests wurden ausgeführt:

Name	Ergebnis
BuyProductsTest	☑
SelectProductsTest	☑
LicensePlateScreenGoingBackwardsTest	☑
ProductScreenGoingBackwardsTest	☑
SummaryScreenGoingBackwardsTest	☑
LicensePlateTest	☑
SummaryScreenTest	☑
AllElementsTranslatedTest	☑
WelcomeScreenNotTranslatedAfterNewStartTest	☑

Abbildung 4.13: Versendete E-Mail nach Beendigung eines fehlerfreien Testlaufs

4.7 Erstellen der Testmethoden

Wie bereits erwähnt wurde, gehört zu jedem Testfall eine Testmethode. Bevor diese Methode ausgeführt wird, muss überprüft werden, ob der WinAppDriver und WPFClient gerade laufen. Falls dies nicht der Fall ist, müssen sie gestartet werden. Eine Methode mit dem Attribut `SetUp` (siehe Abschnitt 2.6) übernimmt diese Funktion und wird automatisch vor jeder Testmethode ausgeführt Nach Ende aller Testmethoden wird eine Methode mit dem Attribut `OneTimeTearDown` ausgeführt, welche die Ergebnisse in der Datenbank speichert. Um Testfälle zu erzeugen, muss zuerst eine Methode mit dem Attribut `Test` erstellt werden. Mithilfe von Reflection werden zur Laufzeit in der Datenbank alle Testfälle gesucht, die zu dieser Testmethode und dem Automaten gehören. Für jeden dieser Testfälle werden danach die Konfigurationswerte ausgelesen und ein Dokumentationsobjekt erzeugt. Danach kann ein Objekt der Klasse `WPFClientApplication` erzeugt werden. Mit diesem Objekt können dann die gewünschten Aktionen auf der GUI simuliert werden. So können z. B. in einer Schleife die in den Konfigurationswerten angegeben Produkte auf der GUI ausgewählt werden. Wichtig ist, dass jeder Testfall innerhalb eines try-catch-Blocks ausgeführt wird, da ein Fehler eine Exception wirft. Abbildung 4.14 zeigt den Ablauf einer Testmethode.

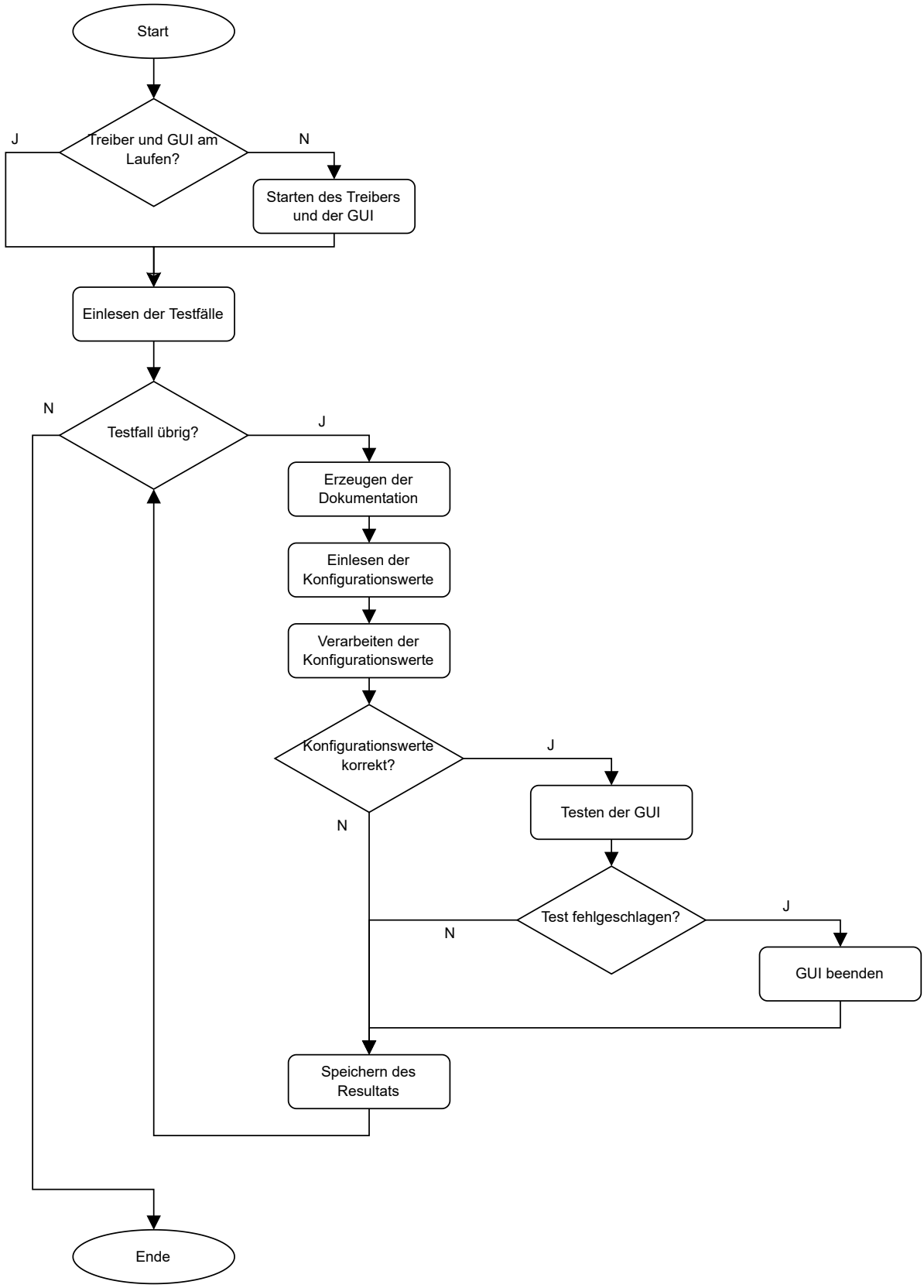


Abbildung 4.14: Ablauf einer Testmethode

Das Auslesen der Konfigurationswerte in einer Testmethode ist im Programmausschnitt 4.15 sichtbar.

```
1         // get the name of the test method
2         string methodName = System.Reflection.MethodBase.GetCurrentMethod().Name;
3         // find all test cases associated with this test method
4         List<TestCase> listOfTestCases = GetTestCasesForCurrentMethod(methodName);
5         // execute each testCase
6         foreach (var testCase in listOfTestCases)
7         {
8             // generate a documentation
9             SingleTestDocumentation singleDoc = TestReport.NewTest(methodName);
10            try
11            {
12                // get all configuration values of the test case
13                var testConfigurationValues = GetTestCaseConfigValues(testCase.Id);
14                // process the configuration values
15                ConfigurationKeyValuePairs testConfigurationObject = new
                  ↪  ConfigurationKeyValuePairs(testConfigurationValues);
```

Abbildung 4.15: Beispielcode einer Testmethode

Die von uns implementierten Testmethoden in der Tabelle 4.16 decken die komplette Funktionalität der GUI ab.

Testmethode	Beschreibung
BuyProductsTest	Überprüfung, ob bei einem Kauf von Produkten alles ordnungsgemäß abläuft
ProductScreenTest	Überprüfung, ob bei der Auswahl von Produkten am Produktbildschirm alles korrekt funktioniert
WelcomeScreenNotTranslatedAfterNewStartTest	Überprüfung, ob der Willkommensbildschirm nach einem Kauf wieder auf Deutsch erscheint
AllElementsTranslatedTest	Überprüfung, ob alle Elemente bei Änderung der Sprache während eines Einkaufs korrekt übersetzt werden
LicensePlateTest	Überprüfung, ob der Kennzeichenbildschirm korrekt funktioniert
ProductScreenGoingBackwardsTest	Überprüfung, ob im Produktbildschirm nach Klick auf den Zurück-Button der Willkommensbildschirm erscheint
LicensePlateScreenGoingBackwardsTest	Überprüfung, ob das Zurückgehen vom Kennzeichenbildschirm bis zum Willkommensbildschirm korrekt funktioniert
SummaryScreenGoingBackwardsTest	Überprüfung, ob das Zurückgehen vom Zusammenfassungsbildschirm bis zum Willkommensbildschirm korrekt funktioniert. Werden am Zusammenfassungsbildschirm alle Produkte gelöscht, muss der WPFClient direkt zum Produktbildschirm zurücknavigieren
SummaryScreenTest	Überprüfung, ob der Zusammenfassungsbildschirm korrekt funktioniert

Abbildung 4.16: Implementierte Testmethoden

4.8 Aufgabenplanung

Die letzte Tätigkeit dieser Bachelorarbeit sah die regelmäßige Ausführung der Testapplikation auf einem Automaten vor. Dafür musste zuerst eine Batch-Datei geschrieben werden. In dieser wird der Befehl

```
dotnet test DiviAutomat-Test.dll --filter "FullyQualifiedName
↪  ~DiviAutomat_Test.WpfClient.GUITest"
```

aufgerufen. Die Option `filter` erwartet einen Ausdruck. Der in diesem Fall angegebene Ausdruck besagt, dass nur jene Tests ausgeführt werden sollen, die sich im Namens-

raum "DiviAutomat_Test.WpfClient.GUITest" befinden. Das sind alle Tests, welche die Funktionalität der GUI überprüfen. Dieses Skript kann nun in die Aufgabenplanung von Windows eingebunden werden. In der Aufgabenplanung stellten wir schlussendlich einen Auslöser ein, welcher das Skript einmal täglich ausführt.

Kapitel 5

Frontend

Damit die Testresultate und die Testberichte im Frontend sichtbar sind, wurde es von uns erweitert. Für das Darstellen der Daten in der Tabelle `CompleteTestResult` konnte die bestehende Komponente `MasterdataList` verwendet werden. Die Komponente erkennt anhand der Route, welche Tabelle sie aus der Datenbank darstellen soll. Die registrierte Route im Router lautet `Masterdata-list/:type`. Der Teil `:type` ist ein Platzhalter. Über einen neu eingefügten Button wird auf die Komponente `Masterdata-List` mit dem :type `CompleteTestResult` navigiert. Da die Komponente `MasterdataList` die Komponente `DataTable` verwendet (siehe Abschnitt 2.3), erweiterten wir im Backend die Klasse `DataModelController` um die Tabelle `CompleteTest-Result`. Das Ergebnis ist in der Abbildung 5.1 sichtbar.

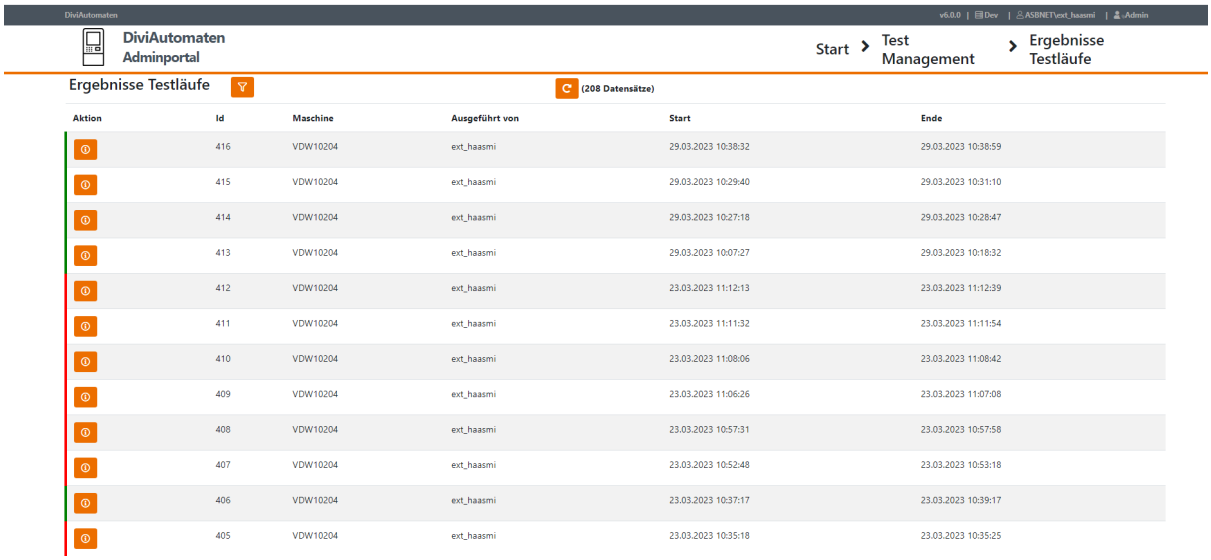


Abbildung 5.1: Darstellung der Testläufe im Frontend

Durch einen Klick auf den Infobutton eines Testlaufs erscheint eine Liste aller Testfälle, welche zum Testlauf gehören. Dies konnte aber nicht mehr mit der Komponente `Data-Table` bewerkstelligt werden, da diese kein Filtern der Einträge zur Verfügung stellt. Deshalb erstellten wir die Komponente `SingleTestResultList`. Diese Komponente bezieht die Information, von welchem Testlauf die Testfälle angezeigt werden sollen, von der Route. Im Backend wurde die Klasse `TestAutomationController` um eine Aktionsmethode erweitert, welche genau diese Daten zurückliefert. Außerdem erstellten wir im

Frontend die Klasse `SingleTestResult`, welche alle Spalten der `Tabelle SingleTestResult` enthält. Über den `HttpClient` (siehe Abschnitt 2.3.2) können die Daten nun angefordert werden. Die zurückgelieferte Liste wird als ein Array mit Elementtyp `SingleTestResult` interpretiert. Diese Daten werden nun durch die Komponente in einer Tabelle dargestellt. Das Resultat ist in Abbildung 5.2 sichtbar.

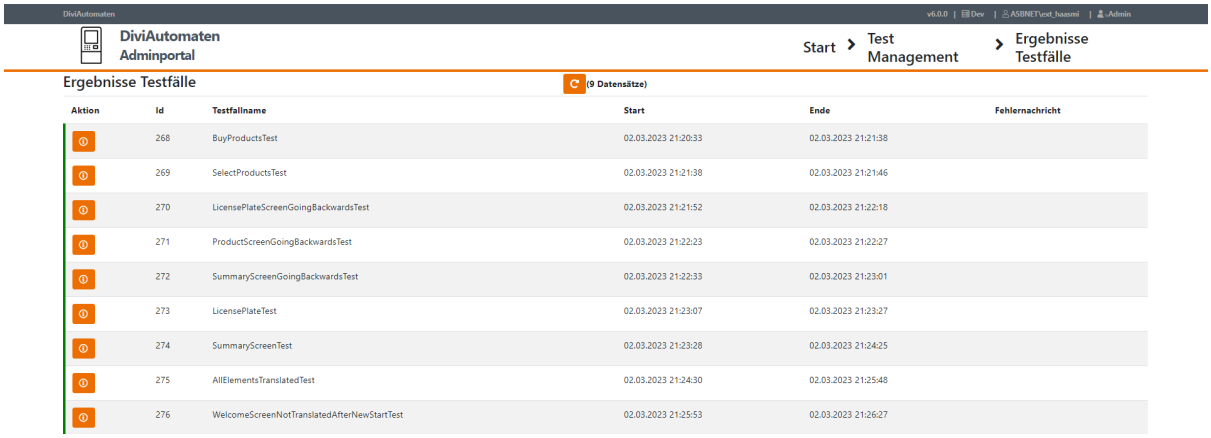


Abbildung 5.2: Darstellung der Testfälle im Frontend

Mit einem Klick auf den Infobutton erscheint wiederum der Testbericht des Testfalls. Die Umsetzung war ähnlich verglichen mit der Komponente `SingleTestResultList`. Es wurde eine eigene Komponente erstellt, welche sich einen bestimmten Tabelleneintrag aus der Tabelle `SingleTestResult` holt. Die durchgeführten Schritte werden dann als Aufzählung dargestellt. Trat im Testfall ein Fehler auf, wird die Fehlernachricht am Ende in roter Schrift dargestellt. Während des Testfalls im Bild 5.3 wurde manuell die Produktanzahl geändert, damit der Testfall fehlschlägt.

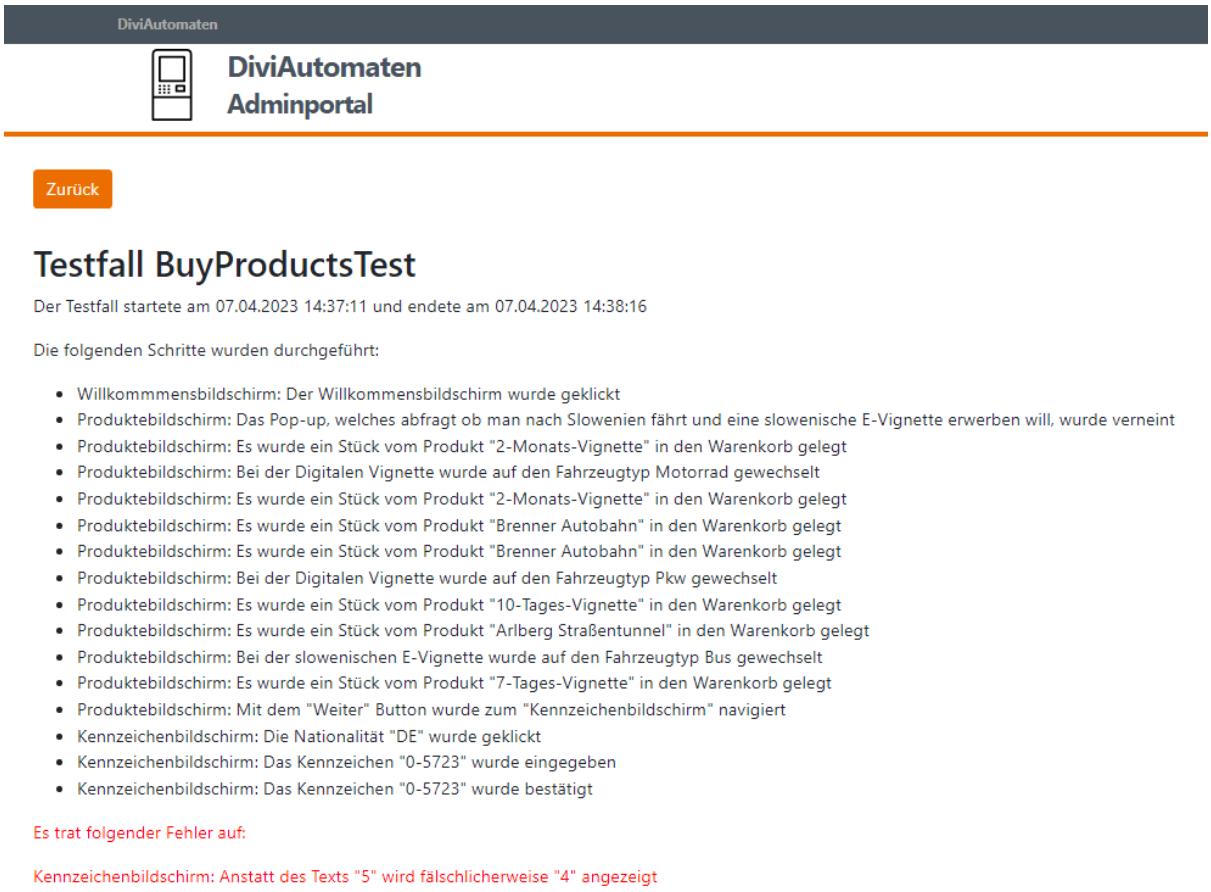


Abbildung 5.3: Ausschnitt eines Testberichts bei einem fehlgeschlagenen Testfall

Der Programmausschnitt 5.4 zeigt die vereinfachte Implementierung der Komponente `SingleTestResult` in TypeScript. Bei der Initialisierung der Komponente wird aus der URL die Testfall-ID ermittelt und der Tabelleneintrag angefordert. Die Daten werden dann in der HTML-Datei verarbeitet und mithilfe der CSS-Datei formatiert dargestellt.

```
1           export class SingleTestResultListComponent implements OnInit {
2
3             public singleTestResult: SingleTestResult
4             public id : string
5
6             constructor(private route: ActivatedRoute, private webserviceClient:
              ↪  AdminClientWebserviceClientService) {
7             }
8
9             ngOnInit(): void {
10                // get the id of the SingleTestResult from the URL
11                this.id = this.route.snapshot.paramMap.get('id');
12                // request the data
13                this.getData()
14            }
15            async getData(){
16                // webserviceClient uses an HttpClient internally
17                this.singleTestResult =  await
                  ↪  this.webserviceClient.getSingleTestResult(this.id);
18            }
```

Abbildung 5.4: Code aus der Komponente `SingleTestResult`

Kapitel 6

Resümee

In dieser Bachelorarbeit implementierten wir für das Testen einer grafischen Benutzeroberfläche eine neue Testapplikation. In der Testapplikation bauten wir zusätzlich einen Testbericht ein, mithilfe dessen Fehler über ein erweitertes Frontend ersichtlich und einfach reproduzierbar sind. Nach jedem Testlauf werden außerdem E-Mails mit einer Kurzzusammenfassung versendet.
Vor Durchführung dieser Bachelorarbeit setzte sich die Testapplikation aus einzelnen Methoden zusammen. Während der Plan in der Anfangsphase der Bachelorarbeit vorsah, eine kleine Adaption durchzuführen, um die Wiederverwendbarkeit zu steigern, verwarfen wir diesen nach genauer Überlegung wieder. Die bisherige Applikation war einfach nicht abstrakt genug, um sie in Zukunft auch für die neu entwickelte GUI wiederverwenden zu können und um einen Testbericht einzubauen. Außerdem wurden intern keine Zustände der GUI gespeichert. Z. B. musste die erwartete Anzahl an ausgewählten Produkten dem Testfall als Parameter mitgegeben werden. Es wurde somit entschieden, die Testapplikation neu zu erstellen und eine eigene Komponente als Schnittstelle zu implementieren, welche alle Funktionalitäten der GUI aufweist und Zustände wie die ausgewählten Produkte intern speichert. Mit Ausnahme einiger Basismethoden (wie dem Starten des WinAppDrivers) verwarfen wir aus diesem Grund große Teile der bisherigen Testapplikation. Da wir zuerst ohne die Klasse `GuiElement` entwickelten und sich herausstellte, dass die Vorteile speziell für den Testbericht beträchtlich waren, musste die Applikation wieder überarbeitet werden.
Das Einlesen in Frameworks und in bestehenden Code stellten für uns speziell zu Beginn eine große Herausforderung dar. Das Abdecken der Funktionalität einer GUI mit ihren Eckfällen stellt ebenfalls einen beträchtlichen Aufwand dar, der nicht zu unterschätzen ist. Wir sind nun aber überzeugt, dass die aktuelle Testapplikation eine gute Basis für das Testen der neu entwickelten GUI bietet.

Abbildungsverzeichnis

Literaturverzeichnis

[1] "Appium Documentation - Appium Documentation," https://appium.io/docs/en/2.0/.

[2] "Attributes — NUnit Docs," https://docs.nunit.org/articles/nunit/writing-tests/attributes.html.

[3] "Using OData with ASP.NET Core Web API - Code Maze," https://code-maze.com/aspnetcore-webapi-using-odata/.

[4] T. C. Huber, *Windows Presentation Foundation: Das umfassende Handbuch zur WPF, aktuell zu .NET Core 3.0, NET 4.8 und Visual Studio 2019*, 5th ed. Bonn: Rheinwerk Computing, Sep. 2019.

[5] J. Martin, *Managing the Data Base Environment*, 1st ed. USA: Prentice Hall PTR, Jan. 1983.

[6] Microsoft, "Dotnet test command - .NET CLI," https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-test, Feb. 2023.

[7] Mikejo5000, "Set a unique automation property - testing UWP controls - Visual Studio (Windows)," https://learn.microsoft.com/en-us/visualstudio/test/set-a-unique-automation-property-for-windows-store-controls-for-testing, Mar. 2023.

[8] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," Internet Engineering Task Force, Request for Comments RFC 2616, Jun. 1999.

[9] A. Troelsen and P. Japikse, *Pro C# 10 With . NET 6: Foundational Principles and Practices in Programming.* Berkeley, CA: Apress L. P, 2022.

[10] J. Wilken, *Angular in Action / Jeremy Wilken.*, 1st ed. Shelter Island, New York: Manning, 2018.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 6. Mai 2023