Hochschule Rhein-Waal
Fakultät Kommunikation und Umwelt
Prof. Dr. Thomas Richter
Dipl.-Inf. Sebastian Jancke

Automatisierte Tests oberflächenbezogener Geschäftslogik mit simulierten Benutzereingaben

Bachelorarbeit

vorgelegt von
Tobias Meyer

Hochschule Rhein-Waal
Fakultät Kommunikation und Umwelt
Prof. Dr. Thomas Richter
Dipl.-Inf. Sebastian Jancke

Automatisierte Tests oberflächenbezogener Geschäftslogik mit simulierten Benutzereingaben

Bachelorarbeit
im Studiengang
Medien- und Kommunikationsinformatik
zur Erlangung des akademischen Grades

Bachelor of Science

vorgelegt von
Tobias Meyer

Matrikelnummer:
23448

Abgabedatum:
08.04.2021

Abstract

A lack of stability and reliability in user interfaces leads to user dissatisfaction. If this dissatisfaction is a repeating occurence users loose trust in the software and the company behind it. In the long run this leads to a migration from existing users to competitors and less new customers. To combat these effects companies have to ensure their user interfaces are stable and reliable. The Schleupen AG provides a browserbased user interface to its customers, which is split into a display layer running in the browser and a logic layer running on the server. In this thesis an automated solution for finding such problems in the logic layer was developed and applied to a selected subset of user interfaces. The solution was able to find multiple issues with most of them representing possible stability problems. This showed that automated testing of user interfaces for stability and reliability is possible and worthwhile.

Keywords: User Experience, Propertybased Testing, Test Automation, Continuous Feedback

Inhaltsverzeichnis

Abbildungsverzeichnis

Die in dieser Arbeit enthaltenen Screenshots entstammen Entwickler- und Testsystemen für die Schleupen.CS Plattform der Schleupen AG und wurden vom Autor persönlich erstellt. Alle in den Testdaten vorhandenen Namen realer Unternehmen wurden zensiert.

Auszüge

1 Einleitung

Die Schleupen AG ist ein mittelständiges Unternehmen, welches mit der Schleupen.CS Plattform und umfassenden Dienstleistungen Lösungen für die Energie- und Wasserwirtschaft bereitstellt. Die Schleupen.CS Plattform setzt sich aus mehreren Geschäftsprozess-Komponenten (GP-Komponenten) zusammen, welche von unterschiedlichen Bereichen entwickelt und betreut werden. Diese GP-Komponenten stellen Benutzeroberflächen bereit, welche dem Endnutzer über eine im Browser laufende Portal UI zugänglich gemacht werden. Als Endnutzer werden in diesem Kontext Sachbearbeiter, zum Beispiel beim Energieversorger, bezeichnet.

Die Portal UI stellt für den Endanwender den Hauptzugangspunkt zur Schleupen.CS Plattform dar. Die Qualität und die Zuverlässigkeit dieser bestimmen zu einem hohen Grad das Vertrauen in und die Wahrnehmung der Software und der Schleupen AG. Deshalb ist es wichtig, dass die UI stabil ist und Fehlerzustände bzw. fehlerhafte Eingaben abgefangen und als Validierungsmeldungen dem Nutzer mitgeteilt werden.

Im Rahmen dieser Bachelorarbeit wird eine prototypische Lösung zur automatisierten Überprüfung der Portal UI auf Fehlerfreiheit und Stabilität entwickelt und im Rahmen einer Fallstudie auf die Oberflächen der GP-Komponente MWM.MML aus dem Bereich Messwertmanagement angewandt.

Zur Überprüfung dieser Anforderungen werden Eigenschaftsbasierte Tests verwendet, welche bestimmte Eigenschaften der Oberflächen untersuchen. Diese verfolgen das Ziel für alle oder eine bestimmte Gruppe von Oberflächen generell gültig zu sein. Für die Umsetzung der Eigenschaften muss eine gemeinsame Basis für die Tests geschaffen und die Tests auf dieser Basis umgesetzt werden. Dabei liegt der Fokus auf der Fachlogik der Oberflächen. Die von dem internen UI Framework bereitgestellte Logik kann an geeigneten Stellen ersetzt werden.

Die zu testenden Oberflächen werden im Produktivbetrieb eingesetzt und sollten deshalb die Anforderungen an Fehlerfreiheit und Stabilität bereits erfüllen. Daraus ergibt sich die Forschungsfrage: **Können durch automatisierte Eigenschaftsbasierte Tests in produktiv eingesetzten Oberflächen Fehler gefunden werden?** Diese Arbeit ist als erfolgreich anzusehen, wenn sie diese Forschungsfrage beantwortet hat.

2 Grundlagen

Zum Verständnis der Lösung zur automatisierten Überprüfung der Portal UI auf Fehlerfreiheit und Stabilität werden in den folgenden Abschnitten die Funktionsweise der Portal UI erklärt, die zwei vorhandenen Testlösungen vorgestellt und der verwendete Ansatz erläutert.

2.1 Funktionsweise der Schleupen.CS Portal UI aus Endnutzersicht

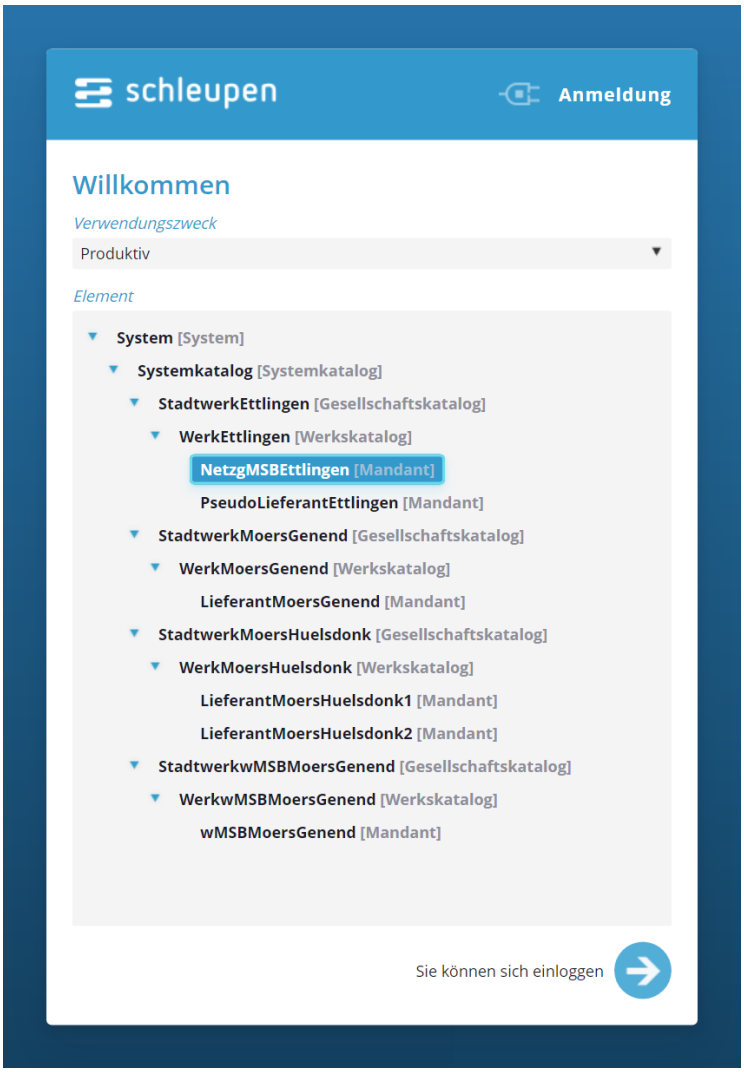


Abbildung 1: Login-Oberfläche der Portal UI

Dem Endnutzer präsentiert sich die Schleupen.CS Portal UI als browserbasierte Webanwendung. Vor Beginn seiner Arbeit muss er sich anmelden. Jedes Schleupen.CS System verfügt über eine hierarchische Systemstruktur nach dem Aufbau *System* → *Systemkataloge* → *Werkskataloge* → *Mandanten*, wobei jeder einzelne Knotenpunkt über eigene Datenbestände verfügt. Über diesen Aufbau können die Unternehmensstrukturen der Kunden und rechtliche Anforderungen am deutschen Energiemarkt zur verpflichtenden Trennung von Marktrollen und Daten abgebildet werden. Der Nutzer muss aus der Liste der Knotenpunkte, wie in Abbildung 1 angezeigt, den für seine Arbeit relevanten Knotenpunkt auswählen. Weil die Abbildung von einem Testsystem stammt, wäre die Anmeldung auf jedem Knotenpunkt möglich. In der Realität hätte der Endnutzer nur die Rechte sich auf einem oder wenigen Knotenpunkten anzumelden.

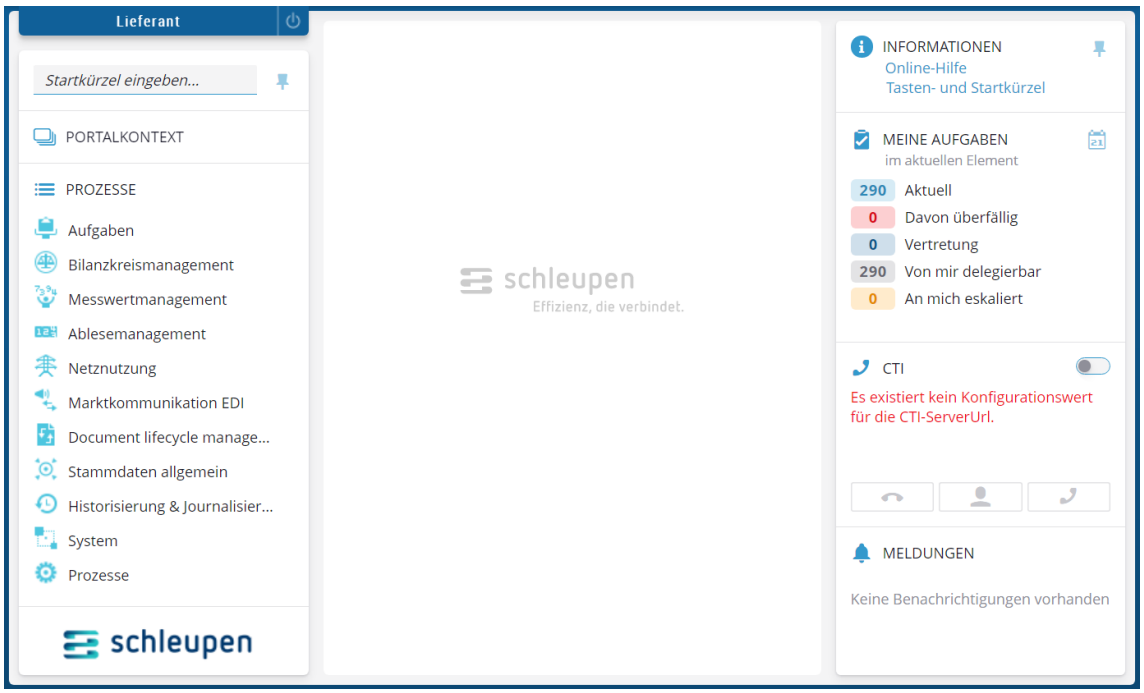


Abbildung 2: Leere Portal UI nach der Anmeldung

Nach der Anmeldung gelangt der Nutzer zur Hauptansicht der Portal UI, welche in drei Spalten aufgeteilt ist. In der linken Leiste befindet sich das Startmenü, in der Mitte eine Freifläche für die Anzeige von Oberflächen und am rechten Rand eine Leiste mit Links zur Dokumentation, der Aufgabenübersicht, der Telefonsystemintegration und Platz für Meldungen.

Innerhalb der Schleupen.CS Plattform sind die meisten Aktivitäten als Prozesse umgesetzt. Der deutsche Energiemarkt ist in unterschiedliche Marktrollen aufgeteilt, welche definierte Aufgabengebiete haben und für gewisse Marktobjekte für die Aufbereitung und Bereitstellung von Daten verantwortlich sind [1]. Die Rolle *Messstellenbetreiber* ist verpflichtet, die für die weiteren Prozesse notwendigen Daten an die relevanten Marktpartner in den Rollen *Lieferant*, *Netzbetreiber* und *Übertragungsnetzbetreiber* zu senden [8]. Wenn diese Marktpartner die Daten nicht fristgericht erhalten oder die erhaltenen Werte von den erwarteten Werten abweichen, haben diese das Recht, die Daten zu reklamieren [2, Abs. 2.8]. Dieser Reklamationsprozess startet mit der Erstellung der Reklamation und soll hier als Beispiel für einen typischen über die Portal UI zu erledigenden Prozess dienen.

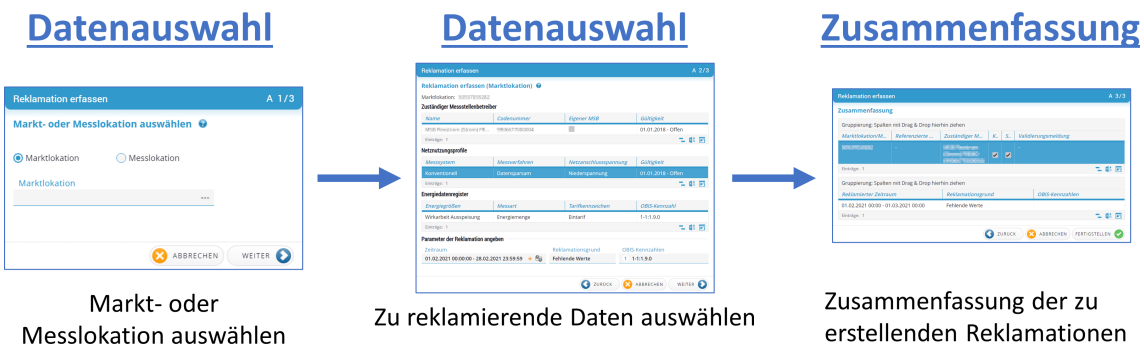


Abbildung 3: Nutzersicht des Prozesses zur Erstellung von Reklamationen

Weil der Reklamationsprozess sich mit Messwerten beschäftigt, ist dieser im hierarchisch angeordneten Startmenü unter *Messwertmanagement* → *Reklamationen* → *Reklamation erstellen* zu finden. Wenn dieser Eintrag im Startmenü ausgewählt wird, öffnet sich eine erste Oberfläche zur Auswahl, ob die Reklamation für eine Marktlokation [1, Abs. 3.2] oder Messlokation [1, Abs. 3.2] erstellt werden soll. Nachdem der Nutzer eine Markt- oder Messlokation ausgewählt hat, drückt dieser auf den *Weiter* Button und gelangt zur nächsten Oberfläche. Diese zeigt an, welcher Messstellenbetreiber für welche Zeiträume zuständig ist und welche Daten reklamiert werden können. In dieser Oberfläche muss der Nutzer auswählen, welche Daten in welchem Zeitraum mit welchem Grund reklamiert werden sollen. Wenn diese Auswahl getroffen wurde, gelangt der Nutzer über einen Klick auf den *Weiter* Button zur nächsten Oberfläche. In dieser wird eine Zusammenfassung der zu erstellenden Reklamationen angezeigt. Wenn eine Reklamation nicht erstellt werden kann, wird der Grund angezeigt. Nach Überprüfung der Angaben, kann der Nutzer mit Klick auf den *Fertigstellen* Button den Prozess abschließen. Die Oberfläche wird geschlossen und im Hintergrund die Reklamationen angelegt, welche anschließend an die entsprechenden Marktpartner versendet werden.

Weil die einzelnen Oberflächen einzelne Schritte in einer Abfolge von Oberflächen sind, werden diese als Dialogschritte und die gesamte Abfolge der Dialogschritte als Dialogablauf bezeichnet. Durch diese Abfolge wird der Nutzer durch die Auswahl von immer spezifischeren Daten geführt, bis diese in einer Aktion wie zum Beispiel der Aktualisierung von Daten oder der Erstellung einer Reklamation enden. Dies erspart das mühsame Zusammensuchen von Daten aus mehreren getrennten Oberflächen und vereinfacht und beschleunigt somit die Abläufe.

2.2 Funktionsweise der Schleupen.CS Portal UI aus Entwicklersicht

Während im vorherigen Abschnitt die Funktionsweise der Portal UI aus Endnutzersicht dargestellt wurde, soll dieser Abschnitt die oben beschriebenen Abläufe technisch beleuchten.

2.2.1 Quellcode-Organisation der Oberfläche

Für den Endnutzer wird die Portal UI als eine Einheit dargestellt. Aus Entwicklersicht ist dies nicht der Fall, sondern die Dialogabläufe und -schritte werden von einer Vielzahl von Geschäftsprozess-Komponenten (GP-Komponenten) bereitgestellt. Da die unterschiedlichen Marktrollen teilweise unterschiedliche Marktprozesse unterstützen müssen, diese teilweise auch rollenspezifische Details haben und nicht jeder Kunde jede Funktionalität lizensiert hat, werden die Geschäftsprozesse in unterschiedlichen Komponenten umgesetzt. Nur die zur Lizenz und zur Marktrolle passenden Komponenten werden installiert.

Aus Entwicklersicht sind Komponenten einzelne Projektmappen im intern verwendeten Code-Editor *Visual Studio*, welche Projekte beinhalten. Alle Dateien für die Portal UI müssen im *ProcessArtifacts* Projekt liegen, neben welchem meist das *ProcessArtifacts.Tests* Projekt zum Testen der Oberflächen liegt.

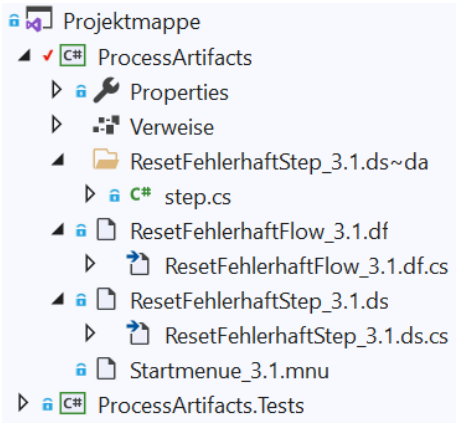


Abbildung 4: Aufbau der Projektmappe einer GP-Komponente

In dem *ProcessArtifacts* Projekt liegen eine *.mnu* Datei, eine *.df* Datei pro Dialogablauf, eine *.ds* Datei pro Dialogschritt und mindestens eine C# Datei pro Dialogablauf und Dialogschritt.

```
<?xml version="1.0" encoding="UTF-8"?>
<DialogStep xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  Namespace="Schleupen.CS.PI.PR.PropertyTesting"
  Name="ButtonThrowsOnClickStep" Version="3.1.0.0"
  Title="Fehlerhafter Button" xmlns="urn://Schleupen.CS.PI.PR.DialogStep_2.1">
  <CodeBehind>
    <SourceFile FileName="step.cs" />
  </CodeBehind>
  <Form>
    <Button Name="ExceptionButton" Size="Wide" Label="Drück mich!">
      <ExecuteMethod Method="ThrowException" />
```

```
    </Button>
  </Form>
</DialogStep>
```

Auszug 1: Beispiel einer *.ds* Datei

In den *.ds* Dateien, nach dem englischen Begriff *dialog step*, sind alle statischen Angaben zu Dialogschritten enthalten. Diese Datei steuert zum Beispiel das Layout, die Felder und ViewModel der Step-Klasse, welche Bibliotheken referenziert und welche weiteren C# Dateien berücksichtigt werden müssen. Aus dieser Datei wird über ein internes Tool eine C# Datei generiert, welche die Step-Klasse beinhaltet. Diese Step-Klasse implementiert alle für die UI benötigten Funktionen und die festgelegten Felder und ViewModel.

Analog sind in *.df* Dateien, nach dem englischen Begriff *dialog flow*, alle statischen Angaben zu Dialogabläufen enthalten. Auch aus dieser Datei wird über ein internes Tool eine C# Datei generiert. Diese beinhaltet die Logik zum Verbinden der einzelnen Dialogschritte.

```
<?xml version="1.0" encoding="utf-8"?>
<StartMenu>
  <Items>
    <MenuGroup>
      <Description>UiPropertyTesting</Description>
      <Name>UiPropertyTesting</Name>
      <Order>1</Order>
      <Items>
        <MenuEntry>
          <Order>1</Order>
          <ExecuteCommand>ResetFehlerhaftFlow_3.1.cmd</ExecuteCommand>
        </MenuEntry>
      </Items>
    </MenuGroup>
  </Items>
  <Commands>
    <Command Id="ResetFehlerhaftFlow_3.1.cmd" Name="Fehlerhafter Reset-Button"
        Description="Reset funktioniert nicht richtig">
      <OpenDialogFlowAction Id="ResetFehlerhaftFlow_3.1.df" />
    </Command>
  </Commands>
</StartMenu>
```

Auszug 2: Beispiel einer *.mnu* Datei

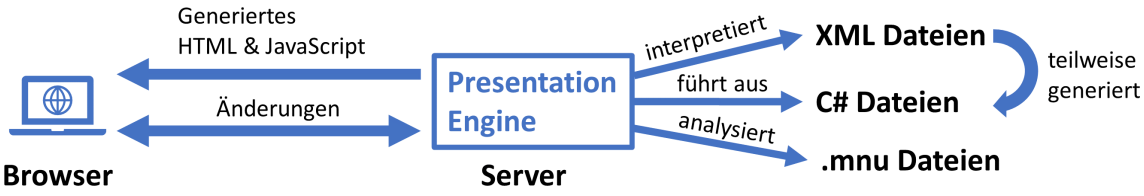
In der *.mnu* Datei wird angegeben, wo die einzelnen Dialogabläufe im Startmenü zu finden sein sollen. Dies geschieht über die Definition von (geschachtelten) Menü-

gruppen und Commands, welche Proxies für Dialogabläufe sind.

Da die oben genannten C# Dateien automatisch generiert werden und bei jedem Generierungsvorgang der alte Inhalt überschrieben wird, muss zusätzlich benötigter Code in weiteren C# Dateien untergebracht werden. Zur besseren Organisation müssen diese in einem passend zu dem Dialogablaufschritt benannten Ordner liegen und in den *.ds* bzw. *.df* Dateien angegeben werden.

2.2.2 Beziehung zwischen Browser und Server

Im vorherigen Abschnitt wurde erklärt wie die Angaben für die Portal UI als Quellcode vorliegen. In diesem Abschnitt wird beschrieben, wie diese in den Browser gelangen.



eigene Abbildung, Icons von https://heroicons.dev/

Abbildung 5: Beziehung zwischen Browser, Server und Quellcode

Die Portal UI ist eine Single-Page-Application (SPA). Im Gegensatz zu den meisten SPAs läuft nur ein sehr geringer Teil, mit genereller Logik, direkt im Browser. Der Großteil der Logik läuft auf dem Server. Zu diesem Zweck läuft im Microsoft Internet Information Services (IIS) Server die intern entwickelte *PresentationEngine*. Diese Anwendung liest alle *.mnu* Dateien aus und erstellt aus diesen das gesamte Startmenü. Wenn der Nutzer zur Hauptansicht gelangt, lädt die Seite das Startmenü als JSON herunter, konvertiert dieses mittels JavaScript zu HTML und zeigt dieses an.

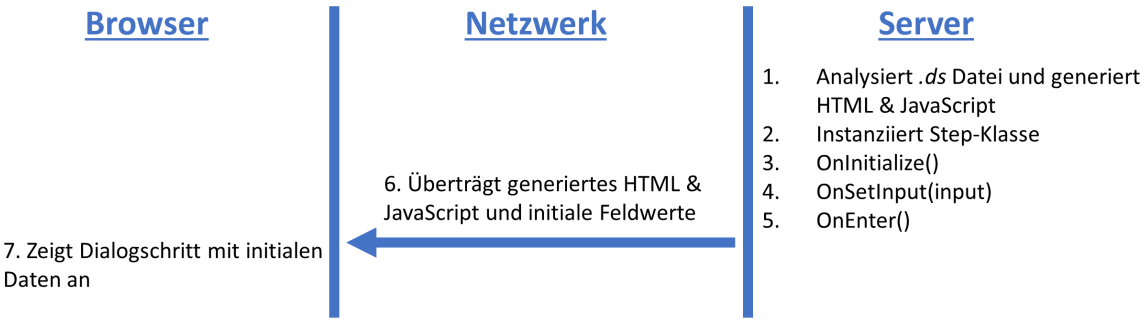


Abbildung 6: Vorgänge zur Öffnung eines Dialogschritts

Wenn der Nutzer anschließend aus dem Startmenü einen Eintrag auswählt, ruft der Browser den Server via Ajax-Request auf und fordert den ersten Dialogschritt

des ausgewählten Dialogablauf an. Die PresentationEngine startet eine Instanz des Dialogablaufs und öffnet den ersten Dialogschritt. Zur Öffnung des Dialogschritts wird die in Abbildung 6 gezeigte Sequenz durchlaufen. Im Browser selbst läuft somit nur die Logik, um die Felder anzuzeigen und zu aktualisieren.

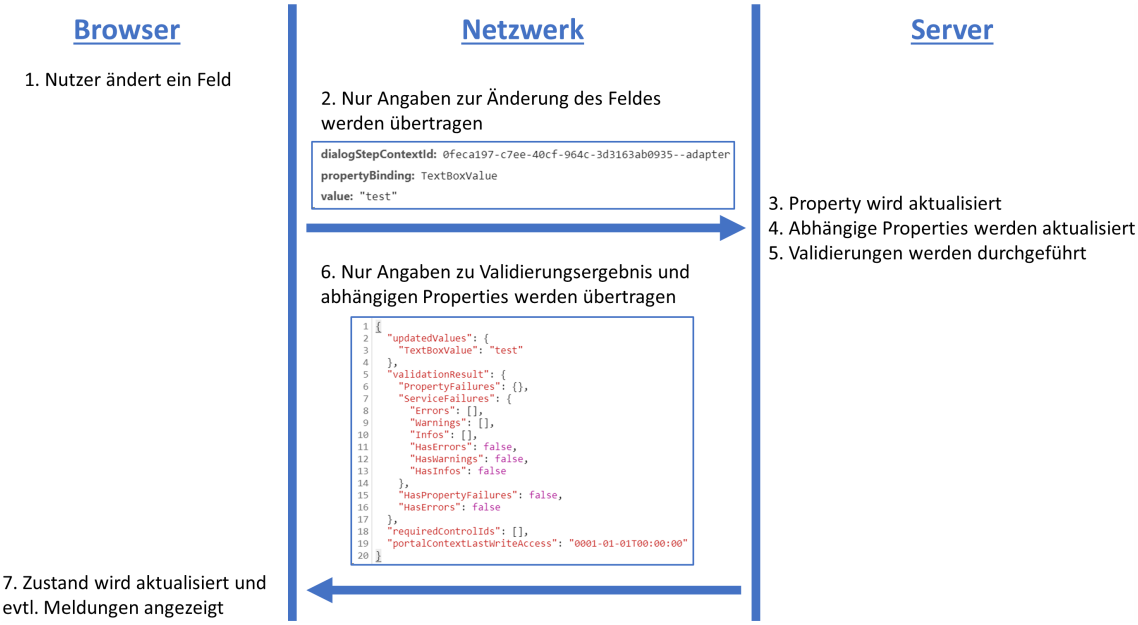


Abbildung 7: Vorgänge zur Aktualisierung eines Feldes

Zu jedem Zeitpunkt während des Lebenszyklus eines Dialogschritts existieren zwei Kopien des Zustands, eine auf dem Server und eine im Browser. Alle Änderungen müssen zwischen diesen synchronisiert werden. In Abbildung 7 wird dargestellt, wie diese Aktualisierung für eine Feldänderung abläuft.

2.2.3 Layout von Dialogschritten

Das Layout von jedem Dialogschritt wird aus unterschiedlichen Elementen zusammengesetzt. Diese Elemente können in die drei Gruppen Container, statische Elemente und interaktive Elemente unterteilt werden. Container werden für die Anordnung von statischen und interaktiven Elementen verwendet. Statische Elemente wie Bilder, Textblöcke und Labels dienen zur Anzeige und Beschriftung von Daten. Interaktive Elemente bieten dem Benutzer Möglichkeiten, Aktionen auszulösen und Daten zu bearbeiten. Für diese Arbeit sind besonders interaktive Elemente von Bedeutung.

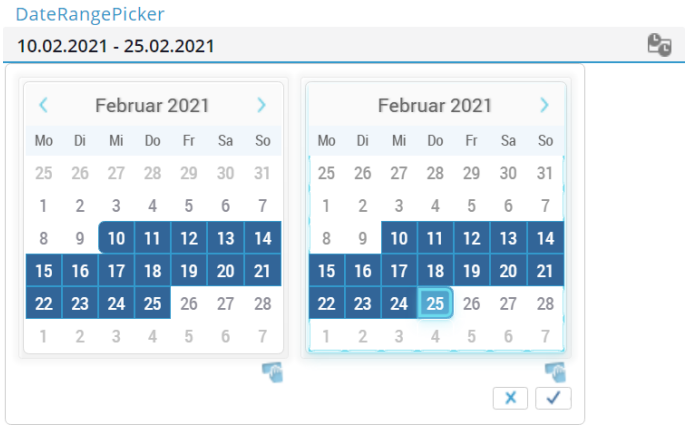


Abbildung 8: DateRangePicker mit Ausfüllhilfe

Diese interaktiven Elemente sind in unterschiedlichen Ausprägungen vorhanden. Durch diese Unterscheidungen können dem Nutzer Hilfen zur Ausfüllung des Elements bereitgestellt werden. In der obigen Abbildung 8 ist ein interaktives Element in der Ausprägung *DateRangePicker* mit geöffneter Ausfüllhilfe dargestellt. Dieses erlaubt die direkte Auswahl des gewünschten Datumsbereichs über eine grafische Oberfläche.

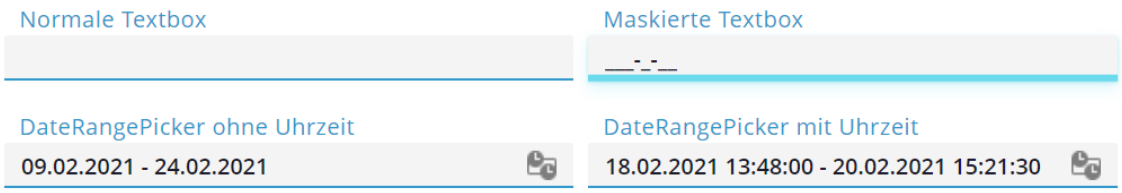


Abbildung 9: Unterschiedliche Textboxen und parametrisierte DateRangePicker

Neben der Anzeige von Ausfüllhilfen ermöglicht die Unterteilung in unterschiedliche Ausprägungen die Angabe von unterschiedlichen Parametern für jeden Typen. Während die normale TextBox keine besonderen Parameter benötigt, benötigt die MaskedTextBox die Angabe einer Maske. Des Weiteren können Parameter zur Anpassung von interaktiven Elementen genutzt werden. Die in der Abbildung 9 gezeigten DateRangePicker haben beide die gleiche Ausprägung, aber für den rechten DateRangePicker wurde die Angabe von Uhrzeiten aktiviert.

```
<TextBox Layout="Row:0,Column:1" Name="TextBox1" TabIndex="0"
  Value="{TextBoxValue}" Label="TextBox" Placeholder="TextBox" />
```

Auszug 3: XML für eine TextBox mit Beschriftung und Platzhalter

Alle Angaben und Parameter der einzelnen Typen werden in der XML-Datei des Dialogschritts gespeichert. Diese Angaben können später analysiert werden, um die Elemente automatisiert zu bedienen.

2.2.4 Validierungsmechanismen für Dialogschritte

Abhängig von dem Kontext können unterschiedliche Vorgaben für Benutzereingaben, wie die Befüllung von Pflichtfeldern, existieren. Zur Mitteilung dieser Anforderungen an den Nutzer wurden Validierungsmechanismen direkt in die Oberfläche integriert.

```
if (instance.DateRangePicker == null)
{
  instance.DateRangePicker = new DateRange();
}
if(instance.DateRangePicker.HasLeftInfiniteEndpoint
  || instance.DateRangePicker.HasRightInfiniteEndpoint)
{
  context.AddFailure(FailureLevel.Error, x => x.DateRangePicker,
    "Bitte Start- und Enddatum angeben.");
}
if(string.IsNullOrWhiteSpace(instance.TextBox))
{
  context.AddFailure(FailureLevel.Error, x => x.TextBox,
    "Bitte angeben.");
}
```

Auszug 4: Validator für FormularModel

Jeder Dialogschritt, jedes ViewModel und jede Property kann über Validierungsregeln verfügen, welche in Validatoren umgesetzt werden. Der hier gezeigte Validator ist für ein im Test-Dialogschritt *Dialogschritt mit Validierung*, welcher in Abschnitt 4.1.5 vorgestellt wird, festgelegtes ViewModel und validiert, dass die Property *TextBox* gefüllt ist und der Wert der Property *DateRange* in beide Richtungen klar definierte Enden hat. Wenn eine oder mehrere dieser Anforderungen nicht erfüllt sind, werden durch den Validator Validierungsmeldungen angelegt. Diese werden an den Feldern, welche an die entsprechenden Properties gebunden sind, angezeigt.

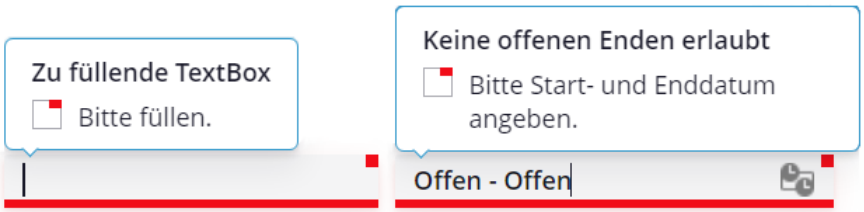


Abbildung 10: Validierungsmeldungen für Felder

Eine weitere Möglichkeit zur Anzeige von Meldungen ist die Anzeige in einer Leiste am unteren Rand des Dialogschritts. Diese wird für Meldungen genutzt, welche sich nicht auf einzelne Felder beziehen.

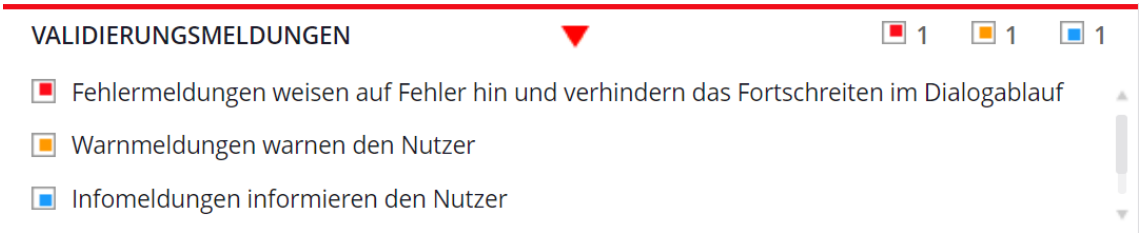


Abbildung 11: Validierungsmeldungen am unteren Rand

Validierungsmeldungen jeder Art können eine von drei Schwerestufen haben. Infomeldungen und Warnungen dienen als Hinweise, haben aber keine sonstigen Auswirkungen. Fehlermeldungen hingegen verhindern das Abschließen des aktuellen Dialogschritts und somit den Wechsel zum nächsten Schritt im Dialogablauf.

2.2.5 Kommunikation mit Services

Die Kommunikation mit Services ist der Hauptweg, um Daten abzurufen und Aktionen durchzuführen.

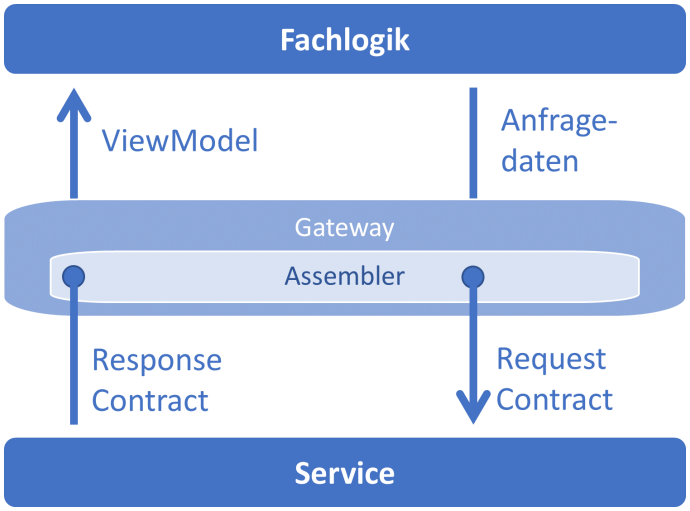


Abbildung 12: Ablauf von Service-Kommunikation

Ausgelöst während der Initialisierung des Dialogschritts oder durch eine Benutzeraktion, wie zum Beispiel die Betätigung eines Such-Buttons, führt die Fachlogik eine Serviceanfrage durch. Zu diesem Zweck ruft sie die für den gewünschten Service zuständige Gateway-Bibliothek auf. Jede Servicemethode verfügt über einen Request und einen Response Contract. Diese werden durch das *Service Interface* definiert und in Abschnitt 2.4 erläutert. In dem Gateway werden die Anfragedaten der Fachlogik in den Request Contract konvertiert. Der Request Contract wird an den Service gesendet, welcher diesen verarbeitet und mit einem Response Contract beantwortet. Anschließend wird der Response Contract in ViewModel konvertiert. Diese Konvertierungen übernehmen sogenannte Assembler, welche Bestandteil des Gateways sind.

Die Verwendung von Gateway-Bibliotheken verfolgt die Ziele Fachlogik und Services zu entkoppeln, die Logik für Serviceaufrufe zu isolieren und deren Wiederverwendung zu ermöglichen. Assembler werden eingesetzt, weil die Contract-Klassen aus dem Service Interface generiert werden. Die Konvertierung in ViewModel vermeidet, im Gegensatz zur direkten Verwendung von Contract-Klassen, die Notwendigkeit von Anpassungen in der Fachlogik aufgrund von Änderungen im Service Interface oder durch Versionswechsel.

2.3 Existierende Testlösungen für die Portal UI

Innerhalb der Schleupen AG existieren zwei Lösungen zum Testen der Portal UI. Sie wurden von zwei unterschiedlichen internen Teams entwickelt und stehen in Konkurrenz zueindander. Durch das Ansetzen an unterschiedlichen Stellen im Aufbau der Portal UI haben die Lösungen unterschiedlichen Vor- und Nachteile abhängig von dem Anwendungskontext der Testfälle, welche in den folgenden Abschnitten vorgestellt werden.

2.3.1 End-to-End-Tests mit Robot Framework

Diese Lösung ist zeitlich zuerst entstanden und ermöglicht die Erstellung von End-to-End-Tests für die Portal UI. Für die Ausführung der Tests werden ein installiertes Schleupen.CS System, ein vollwertiger Browser, das *Robot Framework*, *Selenium* und weitere Abhängigkeiten benötigt. Im Browser wird für jeden Testfall zuerst die Portal UI neu geöffnet und in dieser eine zuvor definierte Liste von Operationen durchgeführt. Die Testfälle werden als Textdateien für das *Robot Framework*, ein generisches Open-Source Framework zur Definition und automatisierten Ausführung von Testfällen, festgehalten. Das Robot Framework wurde durch eine intern entwickelte Erweiterung um spezielle Befehle zur Steuerung der Portal UI erweitert.

Durch das Ansetzen auf der obersten Ebene der Portal UI, wird in den Tests genau der gleiche Code getestet, wie er beim Kunden später installiert wird. Des weiteren können Testfälle für gesamte Dialogabläufe und über Komponentengrenzen hinweg umgesetzt werden. Durch den Overhead durch die Anzeige im Browser, die Kommunikation mit realen Services und das Warten auf Aktualisierungen sind derartige Tests deutlich langsamer als Unit-Tests und instabiler.

2.3.2 Unit-Tests mit PresentationEngineFixture

Das intern entwickelte *PresentationEngineFixture* ermöglicht das Öffnen und die Interaktion mit einzelnen Dialogschritten und Dialogabläufen direkt in C# Code als Unit-Tests umzusetzen. Dabei umgeht das Fixture die gesamte Logik zur Übertragung an den und die Darstellung im Browser, welche durch das interne UI Framework bereitgestellt wird, und greift direkt auf die C# Klassen zu, welche die

oberflächenbezogene Fachlogik implementieren. Zur Ausführung der Tests wird der standardmäßig eingesetzte Testrunner *NUnit* verwendet und die Bibliothek, welche das *PresentationEngineFixture* enthält, wird wie alle anderen Abhängigkeiten der Kompontente von dem Abhängigkeitsmanager *Paket* heruntergeladen. Die Installation aller benötigten Abhängigkeiten ist somit in bestehende Abläufe integriert. Zur Erfüllung von Serviceanfragen werden an Stelle von installierten Services Fake-Services verwendet. Diese müssen die gewünschten Responses in Abhängigkeit von den eingehenden Requests zurückliefern.

Durch den Verzicht auf den Browser, die Kommunikation mit realen Services, und den direkten Zugriff auf den C# Quellcode sind die Tests unabhängig von dem Gesamtsystem, haben eine geringe Durchlaufzeit und geben den Entwicklern direkte Kontrolle über die von Services zurückgelieferten Daten.

2.3.3 Einordnung in die Test-Pipeline

Das Ziel von Tests ist die frühstmögliche Erkennung von Fehlern, weshalb zur Erkennung von Fehlern die schnellste Art von Test verwendet werden sollte [7, S. 132]. In seinem Buch *Succeeding with Agile* führt Mike Cohn das Konzept der Testpyramide (orig. *Test Automation Pyramid*) ein [6, S. 311-313]. Diese ist in mehrere Ebenen, welche jeweils für eine Art von Test stehen, aufgeteilt. In seiner originalen Version unterscheidet er zwischen Unit, Service und UI Tests, wobei Unit-Tests die Basis der Pyramide bilden. Der Aufbau der Pyramide soll die Anzahl der Tests der einzelnen Arten verdeutlichen. Je schneller Tests in der Entwicklung und Ausführung sind, desto tiefer sollen diese in der Testpyramide zu finden sein.

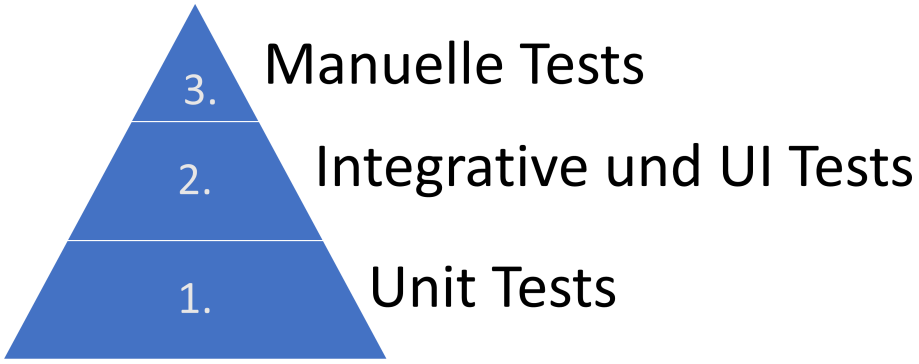


Abbildung 13: Umsetzung der Testpyramide innerhalb der Schleupen AG

Innerhalb der Schleupen AG wurde eine Test-Pipeline erschaffen, welche diese Testpyramide, wie in der Abbildung 13 dargestellt, umsetzt. Die Ebenen stehen für die Ausführungsstufen der Pipeline, beginned mit der untersten Ebene. Die Angaben rechts von der Ebene beschreiben, welche Testarten in der Stufe ausgeführt werden. Die Softwareversionen gelangen nur in die jeweils höhere Ebene, wenn die vorherige Ebene ohne Fehler abgeschlossen wurde. Durch die Ausführung von schnellen Test-

arten vor langsameren Testarten können Fehler mit der schnellsten Testart entdeckt werden. Wenn ein Fehler erst auf einer höheren Ebene gefunden wird, sollte ein Test auf einer niedrigeren Ebene hinzugefügt werden [7, S. 132].

Mit dem *Robot Framework* umgesetzte Tests für die Oberfläche sind integrative Tests und werden im Rahmen der zweiten Teststufe ausgeführt. Zur schnellen Korrektur von Fehlern wird eine schnelle, konstante Feedback-Schleife, wie sie in dem Buch *The DevOps Handbook* beschrieben wird [7, S. 37f.], benötigt. Durch die lange Laufzeit dieser Teststufe werden Entwickler erst mehrere Stunden nach dem Hochladen ihrer Änderungen über eventuelle Fehler informiert. Dadurch vergehen mehrere Stunden bis Tage zwischen Einbau des Fehlers, Erkennung durch fehlgeschlagenen Test, Korrektur und Bestätigung der Korrektur durch erneuten Durchlauf der Test-Pipeline.

Mit dem *PresentationEngineFixture* umgesetzte Unit Tests hingegen werden bereits in der ersten Teststufe durchgeführt. Die Tests dieser Stufe haben eine geringe Durchlaufzeit. Dadurch werden Entwickler innerhalb von maximal einer Stunde über fehlgeschlagene Tests informiert und die Feedback-Schleife beschleunigt. Somit können Fehler früher nach dem Einbau korrigiert werden.

2.4 Defintion von Services über Service Interfaces

Jeder Service wird durch ein versioniertes Service Interface definert. Dieses wird in einem grafischen Tool erstellt und im *Web Services Description Language (WSDL)* [13] XML-Format exportiert. Aus der WSDL-Datei werden über interne Tools die in den entsprechenden Kontexten benötigten C# Dateien generiert.

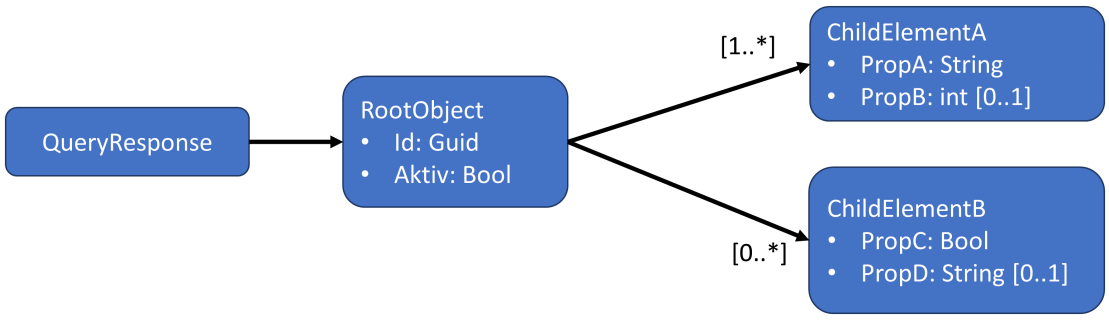


Abbildung 14: Vereinfachtes Service Interface

Die Abbildung 14 zeigt den generellen Aufbau eines stark vereinfachten Service Interface. Jedes Interface ist als eine Baumstruktur aufgebaut. Das oberste Element ist immer ein Interface mit Operationen, welche jeweils ein zugehöriges Request- und ein Response-Objekt haben. Das dargestellte Response-Objekt verfügt über ein direktes Unterobjekt, welches selbst über zwei Unterobjekte verfügt. Jedes Objekt kann Eigenschaften besitzen. Über die Angabe von Kardinalitäten für Eigenschaften und Verbindungen zwischen Objekten wird definiert, ob es sich um einzelne

Elemente oder Listen, und verpflichtende oder optionale Angaben handelt. Diese Struktur mit ihren Angaben wird im Rahmen dieser Bachelorarbeit zur automatisierten Generierung von Response-Objekten verwendet. Dies wird in Abschnitt 4.7 erläutert.

2.5 Reflektion

Für die Analyse der im vorherigen Abschnitt vorgestellten Service Interfaces wird Reflektion genutzt. Die Reflektion ist ein Programmiersprachen-Feature, welches die dynamische Analyse und Verwendung von Objekten und sonstigen Datenstrukturen ermöglicht. In C# ist dieses Feature sehr ausgeprägt und wird im Rahmen dieser Arbeit genutzt, um Vererbungshierarchien, Attribute und Methodensignaturen zu analysieren, Objekte zu erstellen, Methoden aufzurufen und auf Felder sowie Properties lesend und schreibend zuzugreifen. Attribute sind eine Möglichkeit, Zusatzinformationen an Klassen, Methoden, Felder und Properties anzufügen. In den Systembibliotheken werden Attribute, wie zum Beispiel *System.Runtime.Serialization.DataContractAttribute* zur Auszeichnung von DataContracts, definiert und es besteht die Möglichkeit, eigene Attribute zu erstellen.

2.6 Eigenschaftsbasierte Tests

Zur Überprüfung der Portal UI auf Fehlerfreiheit und Stabilität sollen im Rahmen dieser Arbeit Eigenschaftsbasierte Tests (engl. *Property-based Tests*) verwendet werden. In den folgenden Abschnitten wird erläutert, was Eigenschaftsbasierte Tests sind und welche Vorteile sie bieten.

2.6.1 Definition

Die Addition von zwei Zahlen kann vollständig über das Kommutativgesetz, das Assoziativgesetz und die Neutralität der Null definiert werden [14]. Jede Funktion, welche diese Gesetze erfüllt, hat die Addition korrekt implementiert. Während mathematische Gesetze hergeleitet und bewiesen werden können, ist dies für komplexe Computerprogramme nicht oder nur durch unvertretbaren Aufwand möglich. Der beste Ersatz für einen Beweis ist das Aufzeigen der Einhaltung von Gesetzen oder Eigenschaften für den gesamten oder einen bedeutenden Teil des möglichen Eingabenbereich.

Klassische Tests verwenden explizite Beispiele und vergleichen zum Beispiel das Ergebnis der Addition von zwei Werten mit einem erwarteten Wert. Zur Abdeckung eines bedeutenden Teilbereichs des gesamten möglichen Eingabebereichs wird eine in den meisten Fällen nicht manuell implementierbare Menge an expliziten Tests benötigt. Die Bildung von Äquivalenzklassen und Untersuchung von Randbereichen kann helfen die theoretische Abdeckung zu verbessern.

Eigenschaftsbasierte Tests hingegen verwenden keine expliziten Eingabewerte, sondern testen die Einhaltung von Eigenschaften für übergebene oder selbst erzeugte Eingabewerte. Die benötigten Eingabewerte werden meist zufällig generiert. Durch den Verzicht auf bekannte explizite Werte, müssen die Tests allgemeiner formuliert werden. Aus diesem Grund kann ein expliziter Test wie $2+2 = 4$ nicht direkt in einen Eigenschaftsbasierten Test überführt werden, sondern es müssen erst Eigenschaften, wie die oben für die Addition aufgeführten, aufgestellt werden.

2.6.2 Verdeutlichendes Beispiel

Folgendes Beispiel verdeutlicht die Möglichkeiten zur Umsetzung von Eigenschaftsbasierten Tests und deren Vorteile. Als Testsubjekt dient eine Erweiterungsmethode für enumerierbare Objekte mit Namen *ToPartitions*, welche diese in Listen mit einer über einen Parameter anzugebenden Maximallänge portioniert. Im Folgenden werden enumerierbare Objekte von beliebigem Datentypen als Listen bezeichnet.

```
var random = new Random();
for (var i = 0; i < 100; i++)
{
  var partitionSize = random.Next(1, 100);
  var count = random.Next(0, 1000);
  var list = Enumerable.Range(0, count).Select(_ => Guid.NewGuid())
    .ToList();

  var partitions = list.ToPartitions(partitionSize);

  Assert.That(partitions.SelectMany(x => x), Is.EquivalentTo(list));
  Assert.That(partitions.Count,
    Is.LessThanOrEqualTo(count / partitionSize + 1));
  foreach (var partition in partitions)
  {
    Assert.That(partition.Count, Is.LessThanOrEqualTo(partitionSize));
  }
}
```

Auszug 5: Eigenschaftsbasierter Test für *ToPartitions* Erweiterungsmethode

Dieser Eigenschaftsbasierte Test prüft das Ergebnis der Portionierung auf die drei Eigenschaften:

- Die Menge der Elemente nach der Portionierung ist äquivalent zu der originalen Liste.
- Die Liste ist ideal, ohne überflüssige Portionen, aufgeteilt.
- Jede einzelne Portion hält die Maximallänge ein.

Als Eingaben für diesen Test werden zufällige Portionsgrößen größer Null und Listen von zufälliger Länge mit zufällig generierten Elementen verwendet. Durch diesen Grad an Zufälligkeit ist es möglich, einen großen Bereich der möglichen Eingaben abzudecken. Die Beschränkungen der Portionsgröße und der Listenlänge nach oben hin sollen nicht den Eingabenbereich einschränken, sondern die maximale Laufzeit des Tests begrenzen.

```
var input = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var expected = new[] { new[] { 1, 2 }, new[] { 3, 4 }, new[] { 5, 6 },
    new[] { 7, 8 }, new[] { 9 } };

var actual = input.ToPartitions(2);

Assert.That(actual, Is.EqualTo(expected));
```

Auszug 6: Beispielbasierter Test für *ToPartitions* Erweiterungsmethode

In diesem Beispielbasierten Test wird das Ergebnis der Portionierung einer vorgegebenen Liste mit einem vorgegebenen Ergebnis verglichen. Während dieser Test die Portionierung von nur einer Liste mit nur einer Portionsgröße testet, werden neben der eigentlichen Anforderung der Portionierung mehrere implizite Annahmen getestet. In diesem Test wird angenommen, dass das Endresultat eine Liste von Listen mit genau der angegebenen Reihenfolge und genau den angegebenen Elementpaarungen ist. Dies sind jedoch keine Anforderungen an die Portionierung, sondern Implementierungsdetails. Die Paarungen der Elemente, die Reihenfolge der portionierten Listen und welche Listen nicht bis zur Maximallänge befüllt sind, ist vollkommen beliebig, solange die oben genannten Eigenschaften erfüllt werden. Durch diese impliziten Annahmen ist der Test anfällig für Probleme durch Änderungen der Implementierung. Jeder Versuch, diesen Test zu stabilisieren, würde in der Reimplementierung der zu testenden Methode oder in einer Annäherung an den gezeigten Eigenschaftsbasierten Test resultieren.

2.6.3 Vorangegangene Fallstudien und Erfahrungswerte

Für viele Probleme technische Probleme existieren Lösungen in Form von formal verifizierten Algorithmen in wissenschaftlichen Arbeiten. Bei der Übertragung in reale Programme werden Transferleistungen zur Übertragung in und Anpassung an die gewählte Programmiersprache nötig. In wissenschaftlichen Arbeiten werden die Algorithmus meist für idealisierte nicht vollständig definierte Umgebungen entwickelt und verifiziert. Durch die Übertragung in die Realität können Fehler entstehen, da diese von der Idealumgebung abweichen und Entwickler fehler machen können. Zusätzlich ist eine formale Verifizierung von komplexen Systemen meist impraktikabel. Durch Eigenschaftsbasierte Tests können als Ersatz für formale Verifizierung die

formalen Anforderungen für einen signifikanten Teil des Eingabebereichs getestet werden. Dieses Konzept wird in [3] anhand von zwei formal verifizierten Algorithmen zur fehlerresistenten Leader-Auswahl erprobt. Als zu testende Eigenschaften wurden ausgewählt:

1. Nach einer nicht begrenzten Zeit wird ein Leader ausgewählt.
2. Zu jedem Zeitpunkt ist maximal ein Teilnehmer der Leader.

Diese wurden als Tests umgesetzt und gegen eine Implementierung des ersten Algorithmus ausgeführt. Durch diese Tests konnten zwei Verstöße gegen die genannten Eigenschaften gefunden werden. Probleme bei der Beseitigung der Verstöße resultierten in der Implementierung eines zweiten Alogrithmus. Auch in diesem konnten die Autoren durch die Eigenschaftsbasierten Tests mehrere Implementierungsfehler nachweisen, welche jedoch behoben werden konnten. Die in den Implementierungen der beiden Algorithmen gefundenen Probleme traten laut den Autoren nur unter sehr bestimmten Abfolgen von Events statt und wären somit in regulären Tests nicht aufgefallen. Des weiteren waren die gefundenen Fehler nicht in den wissenschaftlichen Arbeiten vorhanden oder lagen ausserhalb des dort vorgesehenen Anwendungsfall. Insgesamt konnten die Autoren zeigen das Eigenschaftsbasierte Tests dabei helfen, zu überprüfen, ob die Theorie auch in der Praxis funktioniert bzw. korrekt übertragen wurde.

Neben dem Testen von Implementierungen von spezifischen Algorithmen, haben die Autoren von [4] positive Erfahrungen im Einsatz von Eigenschaftsbasierten Tests für komplexe Systeme gemacht. In ihrer Fallstudie haben sie ein Programm getestet, das ein spezifisches Protokoll implementiert, und dieses als Blackbox betrachtet. Die gesamte Kommunikation mit dem Programm erfolgte über den Austausch von Nachrichten in dem implementierten Protokoll. Dieses legt fest welche Reihenfolge von Aktionen valide und welche Daten angegeben werden müssen. Mit diesen Angaben stellten sie folgende zu testende Eigenschaften auf:

1. Für jede valide Sequenz antwortet das Programm mit Bestätigungen.
2. Invalide Reihenfolgen resultieren in der Antwort mit Fehlercodes.

Durch die Umsetzung dieser Eigenschaften als Tests konnten mehrere fehlerhafte Annahmen der Entwickler und Unklarheiten in der Spezifikation aufgedeckt werden. Bei Tests gegen ältere Versionen des Programms konnten weitere Fehler identifiziert werden, welche in der aktuellen Version bereits behoben waren. Das Programm hatte bereits mehrere Teststufen durchlaufen und die dabei gefundenen Fehler wurden in einem System dokumentiert. Bei Vergleichen der Autoren ihrer gefundenen Fehlern mit den im System dokumentierten Fehler stellten die Autoren fest, dass ihr System einen bedeutenden Teil der dokumentierten Fehler hätte finden können und nicht

alle ihrer gefundenen Fehler bereits bekannt sind. Insgesamt kamen die Autoren zu dem Schluss, dass der Einsatz ihrer Tests das schnellere automatisierte Aufspüren von Fehlern ermöglicht und bei früherem Einsatz im Entwicklungsprozess einen noch größeren Mehrwert hätte liefern können.

2.7 FsCheck

Als die erste Bibliothek für Eigenschaftsbasierte Tests gilt die von Koen Claessen und John Hughes im Jahre 1999 entwickelte QuickCheck Bibliothek für Haskell [5]. FsCheck ist ein von QuickCheck inspiriertes Framework zur Erstellung von automatisierten Eigenschaftsbasierten Tests für die .NET Sprachen C# und F#.

2.7.1 Eigenschaften

Bevor eine Eigenschaft überprüft werden kann, muss diese definiert werden. Dies geschieht über die Implementierung einer Funktion, welche alle benötigten Inputs entgegennimmt und entweder einen Boolean-Wert zurückgibt oder keinen Rückgabewert besitzt, aber bei Nichterfüllung einen Fehler wirft. Wird ein Boolean-Rückgabewert verwendet, dann muss bei einer erfolgreichen Überprüfung ein wahrer Wert und im Fehlerfall ein unwahrer Wert zurückgegeben werden. Innerhalb der Funktion wird die Einhaltung der Eigenschaft für die Inputs überprüft, wobei FsCheck zur Umsetzung und Komplexität keine Beschränkungen festlegt.

2.7.2 Generatoren

Die Parameter der Eigenschaftsfunktion werden mit generierten Werten befüllt. Für Standarddatentypen wie Integer, Floats, Strings und Booleans, aber auch zum Beispiel für Listen von Integern existieren in FsCheck eingebaute Generatoren. Die möglichen Inputs sind nicht auf diese Datentypen beschränkt, sondern können von beliebigen Typen sein. Zur Generierung von nicht standardmäßig unterstützen Datentypen stellt FsCheck Funktionalitäten zum Anschluss von eigenen Generierungsfunktionen und zur Kombination unterschiedlicher Generatoren bereit.

2.7.3 Shrinking

Der Akt des Shrinkings beziehungsweise die Reduzierung von Inputs auf die unkomplizierteste kleinste Teilmenge des Original-Inputs, welche weiterhin den gleichen Fehler auslöst ist ein wichtiger Beitrag zur Verständlichkeit von Fehlerfällen. Zum Beispiel konnte die Liste von Operationen zur Auslösung eines Fehlers von

1. Press Button "Button_Search"
2. Choose from DropDown "DropDownList1"

3. Press Button "BTN_EinHinzufuegen"
4. Set CheckBox "CheckBox1"to "False"
5. Fill DateRangePicker "DateRangePicker1"with DateRange<29.01.2026 21:54:34, 10.05.2036 03:01:53>

auf

1. Press Button "Button_Search"

reduziert werden. Dies war möglich, weil in diesem Fall die Operation *Press Button "Button_Search"* als erstes ausgeführt wurde, dort bereits ein Fehler aufgetreten ist und die weiteren Operationen somit nicht mehr ausgeführt wurden. Generell können Operationen entfernt werden, wenn diese nicht zur Herstellung des Fehlerzustands benötigt werden. Die neue Liste ist einfacher in der Oberfläche nachstellbar, weil nur ein Buttondruck getätigt werden muss. Zudem wurden die anderen Operationen als mögliche Auslöser für den Fehler ausgeschlossen.

Während der Aufwand für das manuelle Nachstellen der fünf Operationen aus dem Beispiel noch vertretbar ist, wird in [4] ein Fall beschrieben, in dem ein Fehler durch eine Sequenz von über 160 Befehlen ausgelöst wurde. Diese konnte durch Shrinking auf sieben Elemente reduziert werden. Eine Sequenz aus über 160 Elementen für ein komplexes Programm nachzuvollziehen ist nicht direkt möglich, jedoch für die reduzierte Liste. Weiterhin gelang es den Autoren von [9] eine HTML-Datei, die den Firefox Browser zum Absturz bringt, mit fast 900 Zeilen und etwas über 39000 Charakteren auf eine einzige relevante Zeile zu reduzieren. Diese Beispiele zeigen die Bedeutung von Shrinking, besonders für komplexe Programme.

2.7.4 Testausführung

Zur Durchführung von Tests benötigt FsCheck eine zuvor definerte Eigenschaftsfunktion und Generatoren für alle Inputs, welche für die Eigenschaftsfunktion gebraucht werden. Vor der Testausführung wird, sofern nicht angegeben, ein Seed generiert. Für jede der in der Standardkonfiguration 100 Iterationen werden unter Berücksichtigung des Seeds mit Hilfe der Generatoren Inputwerte generiert. Diese Werte werden an die Eigenschaftsfunktion übergeben. Wirft diese einen Fehler oder signalisiert eine nicht eingehaltene Eigenschaft über einen unwahren Boolean-Rückgabewert, wird eine Fehlermeldung ausgegeben und der weitere Testdurchlauf abgebrochen. Ist dies nicht der Fall, wird die nächste Iteration durchgeführt.

Neben dem Seed nehmen Generatoren noch einen Size-Parameter an, welcher die Obergrenze für die Länge, Größe oder Komplexität von generierten Werten festlegt. Standardmäßig steigt dieser Wert mit jeder Iteration, mit der Folge, dass die ersten Tests weniger komplex als spätere Tests sind. Die Mehrzahl der Fehler können

bereits durch einfache Testfälle gefunden werden [10, Abs. 11]. Dies ist eine weitere Maßnahme neben Shrinking, um die Komplexität von Inputs zu minimieren und die Nachvollziehbarkeit von gefundenen Fehlern zu erhöhen. In der Praxis ist es deshalb besser, viele einfache Tests auszuführen als wenige komplexe Tests [10, Abs. 11].

2.7.5 Format von Fehlermeldungen

Wenn durch FsCheck ein Fehler gefunden wird, wird eine Fehlermeldung ausgegeben. In der ersten Zeile der Fehlermeldung befinden sich generelle Angaben zur Anzahl der nötigen Testdurchläufe und Shrinking-Iterationen, sowie der verwendete Seed. Mit diesem Seed kann ein Entwickler die Wiederholung des exakt gleichen Tests für etwaiges Debugging erzwingen. Dies ist möglich dank der im nachfolgenden Abschnitt beschriebenen Replizierbarkeit von Tests. Anschließend folgen die originalen sowie die reduzierten Inputs zur Hervorrufung des Fehlers. Bei dem ersten Wert handelt es sich um den zur Generierung von Responses verwendeten Seed. Dieser Seed wird auch bei Tests angegeben, welche installierte Services verwenden, hat in diesen Fällen jedoch keine Auswirkungen. Der zweite Wert ist die Liste der Operationen, welche an dem Dialogschritt durchgeführt wurden. Den Inputwerten folgt die Exception zum gefundenen Fehler inklusive Stack Trace.

```
Schleupen.CS.PI.PR.PropertyTesting.ResetFehlerhaftStep.V3_1.ResetFehlerhaftStepStepActivity failed

System.Exception : Falsifiable, after 3 tests (2 shrinks) (StdGen (576842773, 296841910)):
Original:
StdGen (479043782, 541550871)
[|Choose from DropDown "DropDownList1"; Fill TextBox "TextBox1" with "";
  Fill DatePicker "DatePicker1" with "19.07.2004 21:47:49"|]
Shrunk:
StdGen (479043782, 541550871)
[|Fill DatePicker "DatePicker1" with "19.07.2004 21:47:49"|]
with exception:
NUnit.Framework.AssertionException:   DatePicker1
  Expected: null
  But was:  2004-07-19 21:47:49.783

   bei NUnit.Framework.Assert.That(Object actual, IResolveConstraint expression, String message, Object[] args)
   bei Schleupen.CS.PI.PR.PropertyTesting.Tests.ResetTest.<.ctor>b__1_1(OperatableStep operatableStep,
     DialogStepActivityCodeBehind step) in C:\dev\pi.pr.PT\PR.PropertyTesting\Tests\ResetTest.cs:Zeile 35.
   bei Schleupen.CS.PI.PR.PropertyTesting.Tests.BaseTest.<>c__DisplayClass19_1.<Execute>b__3(Action`2 hook) in C:\dev
     \pi.pr.PT\PR.PropertyTesting\Tests\BaseTest.cs:Zeile 157.
   bei Castle.Core.Internal.CollectionExtensions.ForEach[T](IEnumerable`1 items, Action`1 action)
   bei Schleupen.CS.PI.PR.PropertyTesting.Tests.BaseTest.<>c__DisplayClass19_0.<Execute>g__DoesNotThrow|0(StdGen
     fsCheckSeed, Operation[] ops) in C:\dev\pi.pr.PT\PR.PropertyTesting\Tests\BaseTest.cs:Zeile 157.
   bei FsCheck.Prop.ForAll@33-7.Invoke(V2 v2)
   bei FsCheck.Testable.evaluate[a,b](FSharpFunc`2 body, a a)

   bei Microsoft.FSharp.Core.PrintfModule.PrintFormatToStringThenFail@1433.Invoke(String message) in F:\workspace\_work
     \1\s\src\fsharp\FSharp.Core\printf.fs:Zeile 1433.
   bei FsCheck.Runner.check[a](Config config, a p)
   bei Schleupen.CS.PI.PR.PropertyTesting.Tests.BaseTest.Execute(OperatableStep operatableStep) in C:\dev\pi.pr.PT
     \PR.PropertyTesting\Tests\BaseTest.cs:Zeile 165.
   bei Schleupen.CS.PI.PR.PropertyTesting.PropertyTestBase.ResetUeberfuehrtInInitialstand(OperatableStep operatableStep)
      in C:\dev\pi.pr.PT\PR.PropertyTesting\PropertyTestBase.cs:Zeile 102.
```

Abbildung 15: Beispiel für eine FsCheck Fehlermeldung

In der Abbildung 15 wird eine Fehlermeldung zur Eigenschaft *Die Betätigung eines eventuell vorhandenen Reset-Buttons überführt die Felder in den Initialzustand.* gezeigt. Für den DatePicker mit Namen DatePicker1 wird der Reset nicht korrekt

durchgeführt, weshalb der durch eine Operation gesetzte Wert nach dem Reset weiterhin bestehen bleibt. Zur Hervorrufung dieses Fehlers wird nur eine Operation zur Befüllung des DatePickers benötigt. Alle weiteren Operationen konnten entfernt werden.

2.7.6 Replizierbarkeit

Nach der Identifikation eines Fehlers muss dieser analysiert und behoben werden. Durch die Verwendung von neuen Zufallswerten für jeden Testdurchlauf testet nicht jeder Durchlauf die gleichen Codepfade. Zur Analyse und zur Sicherstellung der Korrektur ist daher eine einfache wiederholte Ausführung des fehlgeschlagenen Tests nicht zielführend, sondern verleitet zu einer falschen Sicherheit, wenn der Fehler bei mehrfacher erneuter Ausführung nicht mehr auftritt [5, Abs. 5.1.4].

Im Idealfall kann für die Analyse der exakt gleiche Test wie für die Fehleridentifikation ausgeführt werden. Dies ist bei FsCheck durch die Angabe eines Seeds für den Testdurchlauf möglich. Wenn kein Seed angegeben ist, generiert FsCheck einen eigenen und gibt diesen in Fehlermeldungen aus. Zum Debugging kann dieser Seed aus der Fehlermeldung übernommen werden. Bei der Generierung von Testdaten wird dieser an die Generatoren übergeben, welche, bei korrekter Implementierung, für den gleichen Seed immer den gleichen Wert generieren.

3 Zu testende Eigenschaften

Anhand von persönlichen Erfahrungen, Auswertungen von abgeschlossenen Fehlertickets und Gesprächen mit Mitgliedern des internen UI Teams wurde die folgende Selektion an zu testenden Eigenschaften erstellt.

3.1 Die Öffnung des Dialogschritts ist möglich.

Bevor der Nutzer mit einem Dialogschritt interagieren kann, muss dieser geladen und initialisiert werden. Durch Fehler in verwendeten Biblotheken oder dem Initialisierungscode kann das Öffnen von Schritten verhindert werden. Somit stellt diese Eigenschaft die Grundlage für die Bedienbarkeit der Dialogschritte und aller weiteren Eigenschaften dar.

3.2 Die Öffnung des Dialogschritts im Preview-Modus ist möglich.

Während der Entwicklung ist es hilfreich, den Dialogschritt neben der Bearbeitungs-Ansicht im Editor auch in seiner realen Umgebung gefüllt mit Daten anzusehen. Dies

ist unter anderem durch die Navigation zu dem Schritt über den zugehörigen Dialogablauf möglich. Besonders bei Abläufen, welche sich noch in der Entwicklung befinden, und Schritten, welche erst das Ausfüllen von vorherigen Schritten erfordern, beansprucht dies viel Zeit. Zudem müssen für alle zu prüfenden Ansichten passende Datenkonstellationen vorbereitet werden. Aus diesen Gründen existiert für Dialogschritte ein Preview-Modus, welcher als Inputdaten von dem Entwickler spezifizierte Daten verwendet.

Mit dieser Eigenschaft soll sichergestellt werden, dass die Dialogschritte im Preview-Modus geöffnet werden können. Dies ist nicht immer gewährleistet, weil der Preview-Modus vor allem während der Entwicklung des Layouts verwendet wird. Bei Änderungen an der Fachlogik hingegen wird dieser nur selten genutzt. Dies kann in Inkompatibilitäten zu den Inputwerten für den Previw-Modus resultieren. Des Weiteren müssen Service-Aufrufe gezielt angepasst werden, um zu den Previewdaten passende Ergebnisse zurückzuliefern.

3.3 Es existiert keine Folge von validen Benutzereingaben, welche in einem Fehler resultiert.

Nach der Öffnung eines Dialogschritts bieten sich dem Nutzer, mit Ausnahme von Schritten zur reinen Anzeige von Daten, unterschiedliche Möglichkeiten der Interaktion. Zu diesen Möglichkeiten zählen das Befüllen von Feldern, die Betätigung von Buttons, die Auswahl von Elementen in Tabellen und die Verwendung von Kontextmenüs. Nicht in jedem Fall stehen alle der genannten Interaktionsmöglichkeiten dem Nutzer zur Verfügung, jedoch müssen alle dem Nutzer zur Verfügung stehenden Interaktionsmöglichkeiten korrekt umgesetzt sein.

Mit dieser Eigenschaft soll sichergestellt werden, dass alle verfügbaren Möglichkeiten verwendet werden können und zu keinem Fehler führen. Dabei sollen die verwendeten Daten und die Interaktion mit dem Gesamtsystem über reale, lokal installierte Services ablaufen.

3.4 Die Verarbeitung von beliebigen validen Service-Responses ist möglich.

Im Gegensatz zur vorherigen Eigenschaft liegt bei dieser Eigenschaft der Fokus nicht auf der Verarbeitung von Eingabedaten des Nutzers, sondern auf der Verarbeitung der von den Services erhaltenen Responses. Weiterhin werden alle möglichen Interaktionsmöglichkeiten verwendet, um Service-Requests zu erzeugen. Jedoch werden die echten Services durch Fakes ausgetauscht, welche zum Service Interface passende Responses generieren. Durch diesen Ansatz kann die Verarbeitung des gesamten Spektrums an möglichen Responses getestet werden, wobei besonders Randfälle ge-

genüber trivialen Werten interessant sind.

Serviceaufrufe sind der Hauptkommunikationsweg zwischen den Dialogschritten und dem Rest des Systems, welcher sowohl für das Abfragen von Daten sowie die Ausführung von Aktionen dient. Durch diese zentrale Rolle muss die Verarbeitung der von Services erhaltenen Daten äußerst stabil erfolgen. Wenn in der Fachlogik, den Gateways oder den Assemblern falsche Annahmen über die zu erwartenden Daten getroffen werden, führt dies potenziell zu Problemen. Besonders durch die Änderung von Implementierungsdetails der Services, welche keine Änderungen im Service Interface bedingen, können derartige Fehler hervorgerufen werden. Aus diesem Grund müssen alle validen Service Responses verarbeitet werden können.

3.5 Die Betätigung eines eventuell vorhandenen Reset-Buttons überführt die Felder in den Initialzustand.

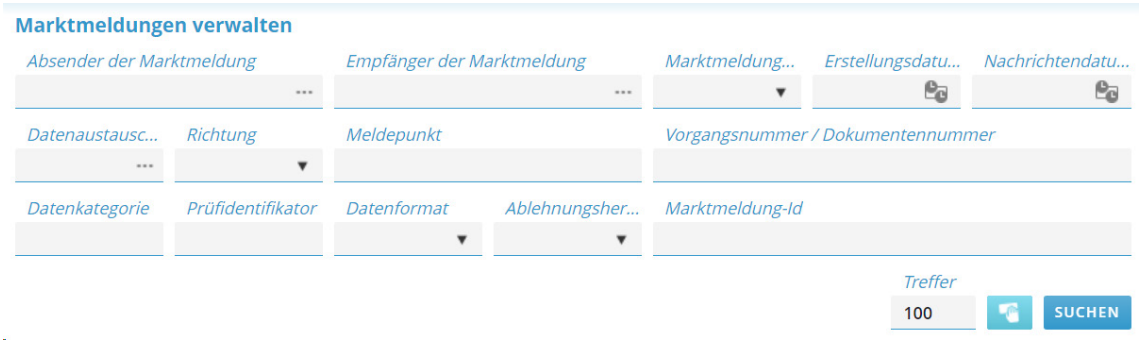


Abbildung 16: Beispiel für ein komplexes Suchformular

Viele Dialogschritte enthalten Suchformulare, welche aufgrund der Komplexität des deutschen Energiemarktes häufig eine ähnliche Komplexität wie das im Beispiel gezeigte Suchformular aufweisen. Besonders bei großen Datenbeständen ist die genaue Eingrenzung der Suchergebnisse unverzichtbar. Sie ist aber auch bei weniger Daten ein wichtiges Feature, weil in vielen Anwendungsfällen die Beantwortung der Frage nach der Existenz von Daten mit bestimmten Eigenschaften ausreichend ist. Als Hilfsmittel zwischen Suchen werden Reset-Buttons für den Nutzer angeboten, welche alle Felder in den Initialzustand versetzen. Der Initialzustand muss dabei kein leeres Feld darstellen. Beispielsweise werden DateRangePicker häufig initial mit einem von dem Kontext abhängigen sinnvollen Zeitraum, wie dem vergangenen Monat, gefüllt.

Der Reset-Button ist an eine Methode gebunden, welche den Reset durchführt und manuell umgesetzt werden muss. Dabei kann besonders der Anschluss später hinzugefügter Felder vergessen werden, weil diese auch durch eventuell vorhandene Tests für den Reset nicht abgedeckt werden.

Mit dieser Eigenschaft soll für alle Suchformulare mit Reset-Button überprüft werden, ob alle Felder nach einem Reset einen äquivalenten Wert zum Initialzustand

nach dem Öffnen des Dialogschritts haben. Äquivalenz wird in diesem Kontext definiert als die Anzeige des gleichen Resultats im Browser. Zum Beispiel werden bei einer TextBox der Null-Wert und ein leerer String beide als leeres Feld angezeigt und sind somit äquivalent, obwohl die Werte unterschiedlich sind.

3.6 Die Änderung eines Feldes in einer Suchmaske resultiert in der Änderung des Such-Requests.

Über die Angaben in der Suchmaske kann die Ergebnismenge eingegrenzt werden. Dabei ist die Erwartung, dass die gefundenen Ergebnisse auch alle über das Suchformular vorgenommenen Einschränkungen erfüllen. Für die Implementierung bedeutet dies, dass jede Änderung eines Feldes sich im Such-Request widerspiegeln muss.

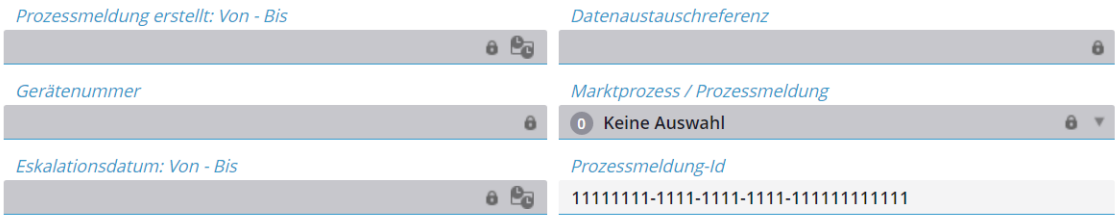


Abbildung 17: Nicht relevante Felder sind im ReadOnly-Modus

Wenn ein oder mehrere Felder für eine Suche nicht relevant sind, weil wie im Beispiel direkt über eine ID gesucht wird, dann müssen diese Felder auch nicht im Such-Request berücksichtigt werden. Dieser Umstand muss dem Benutzer über das Versetzen der entsprechenden Felder in den ReadOnly-Modus oder durch deren Verstecken signalisiert werden.

Mit dieser Eigenschaft soll deshalb geprüft werden, ob alle nicht versteckten oder sich im ReadOnly-Modus befindlichen Felder eine Auswirkung auf den Such-Request haben.

4 Implementierung

Auf Basis der zuvor vorgestellten Grundlagen und zu testenden Eigenschaften wurde eine prototypische Lösung zur automatisierten Überprüfung dieser Eigenschaften entwickelt. Die zum Testen entwickelten Dialogschritte, der den Eigenschaften gemeinsame Testablauf, die Implementierung der einzelnen Eigenschaften und weitere Entwicklungsarbeiten werden in den folgenden Abschnitten vorgestellt.

4.1 Test-Dialogschritte

Zur Demonstration unterschiedlicher Fehlerarten und als Referenzen während der Implementierung wurden mehrere Test-Dialogschritte erstellt. Diese befassen sich jeweils mit einem konkreten Problem, welches im Folgenden für die einzelnen Schritte erläutert wird.

4.1.1 Fehlerhafter Button

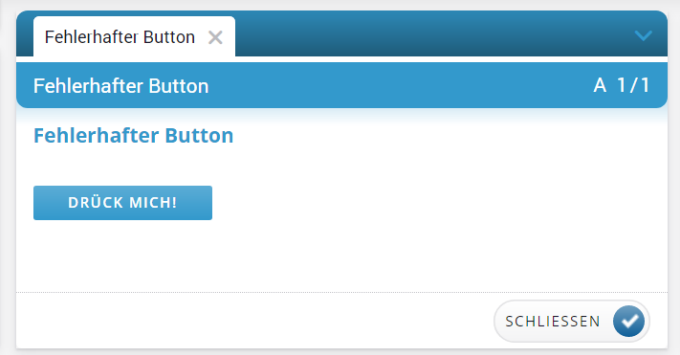


Abbildung 18: Dialogschritt mit fehlerhaftem Button

Dieser Schritt besteht aus einem einzelnen Button. Wird dieser Button gedrückt, wird eine Methode aufgerufen, welche einen Fehler wirft. Dieser wird anschließend als Fehlermeldung in der Benutzeroberfläche angezeigt.

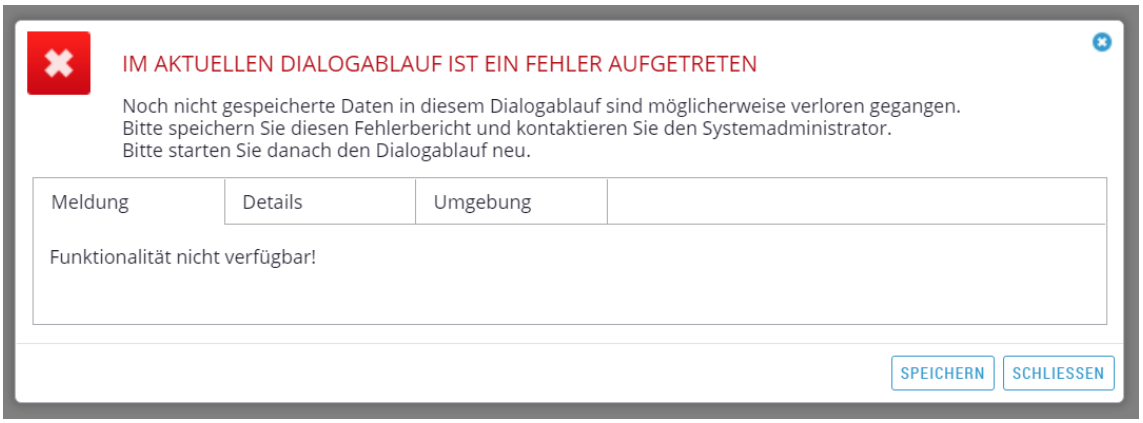


Abbildung 19: Fehlermeldung nach Betätigung des Button

Die Betätigung eines aktiven Buttons ist eine valide Aktion und sollte somit nicht in einem Fehler resultieren. Durch die Eigenschaft *Es existiert keine Folge von validen Benutzereingaben, welche in einem Fehler resultiert.* sollen derartige Fehler erkannt werden.

4.1.2 Fehlerhafter Reset

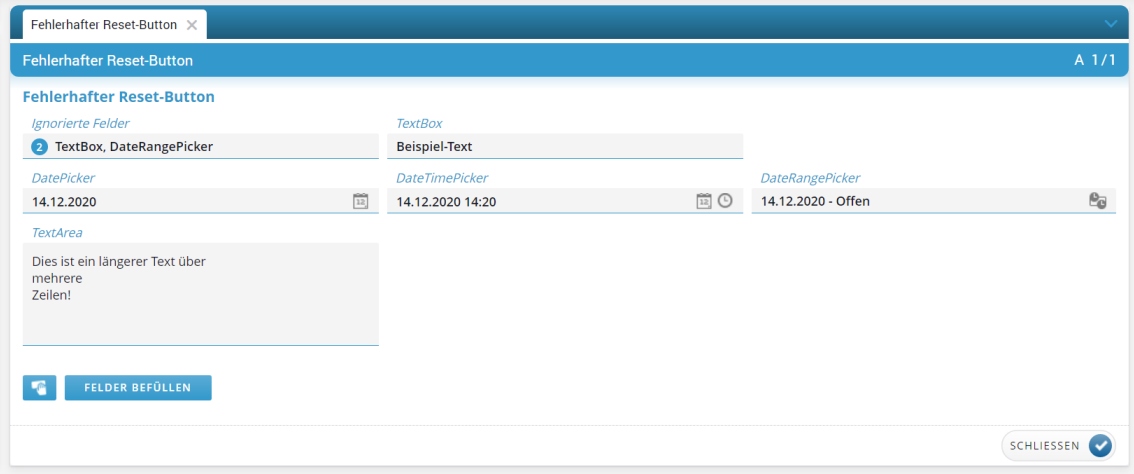


Abbildung 20: Dialogschritt mit fehlerhaftem Reset

In diesem Dialogschritt wurde ein Formular aus den unterschiedlichen Inputtypen und zwei Buttons zusammengebaut. Der linke Button ist ein Reset-Button. Dies ist an dem genutzten Icon erkennbar, welches das Standardicon für alle Reset-Buttons ist. Bei dem rechten Button handelt es sich um eine Hilfe zu Demonstrationszwecken. Dieser Button befüllt alle Felder mit Werten und erspart somit ein manuelles Ausfüllen. Der erste Input, ein DropDown mit der Möglichkeit zur Auswahl von mehreren Elementen, enthält Einträge für sich selbst und alle weiteren Inputs. Alle in diesem DropDown ausgewählten Inputs werden beim Auslösen des Reset-Buttons nicht zurückgesetzt und führen somit zu einem fehlerhaften Reset-Vorgang.

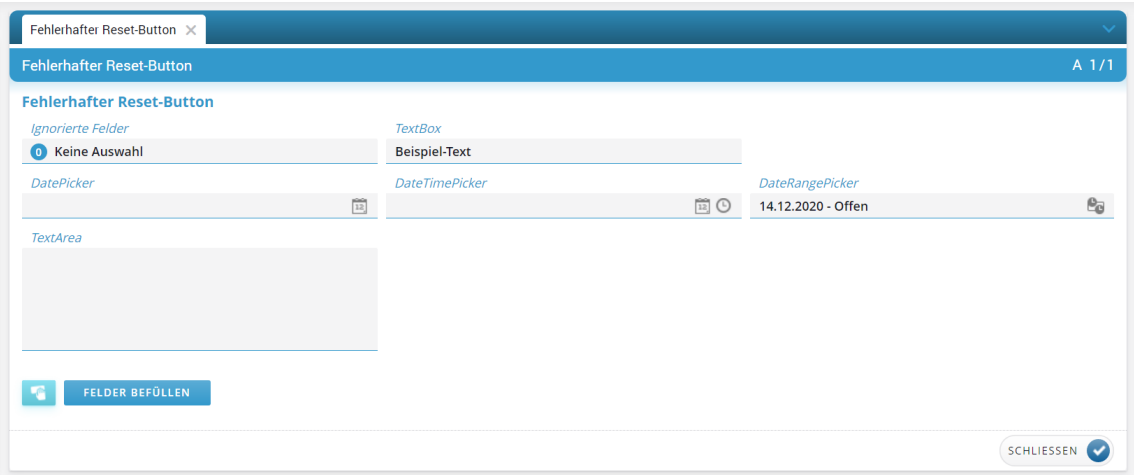


Abbildung 21: Fehlerhaft durchgeführter Reset

Während dieser Schritt für alle validen Operationen keine Exception wirft, sollen derartige Fehler beim Reset von Formularen durch die Eigenschaft *Die Betätigung eines eventuell vorhandenen Reset-Buttons überführt die Felder in den Initialzustand.* identifiziert werden.

4.1.3 Beim Suchen ignoriertes Feld

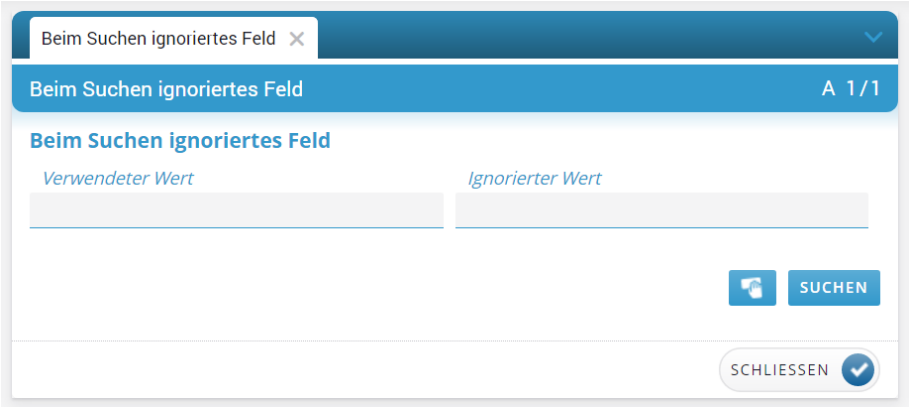


Abbildung 22: Dialogschritt mit beim Suchen ignoriertem Feld

Von den zwei im Formular vorhandenen Feldern wird beim Suchen über einen Service-Aufruf nur das linke Feld verwendet. Alle in einem Suchformular vorhandenen Felder sollten verwendet und ansonsten entfernt oder deaktiviert werden. Das Vorhandensein eines nicht verwendeten Feldes soll durch die Eigenschaft *Die Änderung eines Feldes in einer Suchmaske resultiert in der Änderung des Such-Requests.* erkannt werden.

4.1.4 Dialogschritt mit Webservice-Referenz

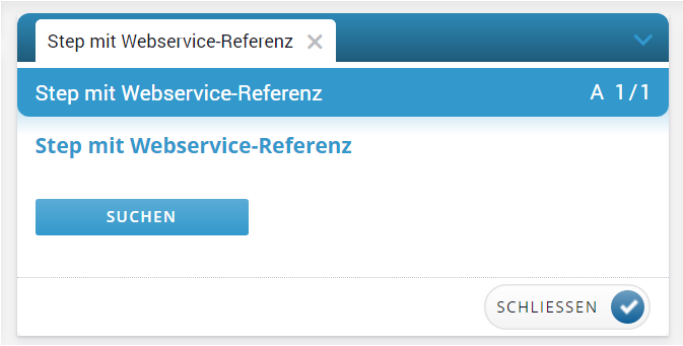


Abbildung 23: Dialogschritt mit Webservice-Referenz

In diesem Dialogschritt sind keine Fehler eingebaut, da es sich hierbei um eine Implementierungshilfe handelt. Durch Betätigung des Buttons wird eine Anfrage an einen Service ausgelöst. Damit dies funktioniert, muss im *PresentationEngineFixture* für diesen Service ein Ersatz registriert sein.

```
System.Exception: Exception aufgetreten nach Operation: Press Button "Button1" --->
  System.Reflection.TargetInvocationException: Ein Aufrufziel hat einen Ausnahmefehler verursacht. --->
  System.InvalidOperationException: Could not find a registered WebserviceSubstitute for
  'Schleupen.CS.PI.BPE.Tasks.TaskQueryService_3.7|
  Schleupen.CS.PI.PR.PropertyTesting.Gateways.TaskQueryServiceGateway.V3_1.TaskQueryService|
  Schleupen.CS.PI.PR.PropertyTesting.ProcessArtifacts, Version=3.26.0.0, Culture=neutral, PublicKeyToken=null'.
  Make sure to register necessary Webserice-Calls with PresentationEngineFixture.RegisterWebserviceSubstitute.
  See http://wi/doku.php?id=sb_ui_unittest_api#aufruf_von_webservices
   bei Schleupen.CS.PI.PR.Testing.Fakes.EmbeddedWebserviceProxyBuilderFake.CreateProxyFor(String serviceId,
     String csharpNamespace, Assembly assembly)
```

Abbildung 24: Fehler durch nicht richtig registrierten Service-Ersatz

Wenn der Testaufbau fehlerhaft ist, dann führt dies beim Drücken des Buttons zu einer Fehlermeldung und fällt durch die Eigenschaft *Es existiert keine Folge von validen Benutzereingaben, welche in einem Fehler resultiert.* auf.

4.1.5 Dialogschritt mit Validierung

Für die TextBox und den DateRangePicker wurden Validierungsregeln festlegt, welche bei Nichterfüllung zu Hinweisen führen, wie sie in der Abbildung 25 gezeigt werden. Nur wenn diese Validierungsregeln erfüllt werden, ist der Aktions-Button aktivierbar.

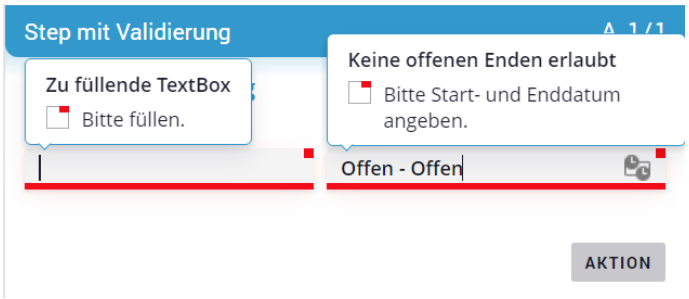


Abbildung 25: Dialogschritt mit Validierung

Anhand dieses Schrittes wurde erprobt, wie die Operationen mit Validierungsmeldungen umgehen müssen. Wenn der Aktions-Button trotz vorhandener Validierungsmeldungen gedrückt wird, führt dies zu einer Fehlermeldung und würde in den Testergebnissen auftauchen.

4.2 Wahl der Testlösung

Für die Umsetzung von Eigenschaftsbasierten Tests muss die gewählte Testlösung eine Vielzahl an Tests in kurzer Zeit ausführen. Zusätzlich sollten diese Tests für den effektiven Einsatz von Shrinking und zur vereinfachten Fehleranalyse unabhängig von äußeren Umständen und replizerbar sein.

Das *Robot Framework* bietet durch die von der internen Erweiterung bereitgestellten Befehle bereits alle Mechanismen zur Interaktion mit der UI an. Die in dieser Arbeit zu entwickelnde Lösung müsste diese Befehle nur automatisiert aufrufen. Das

PresentationEngineFixture hingegen stellt keine direkten Befehle für die Interaktion mit der UI bereit, sondern ermöglicht den direkten Zugriff auf Instanzen der Step-Klassen. Bei Wahl des *PresentationEngineFixture* müsste die Lösung somit die Bedienung der einzelnen interaktiven Elemente selbst implementieren.

Zur Durchführung einer Vielzahl von Tests in kurzer Zeit wird eine geringe Laufzeit pro Test benötigt. End-to-End-Tests mit dem *Robot Framework* benötigen durch den Umweg über den Browser und die Verwendung der installierten Services deutlich länger als mit dem *PresentationEngineFixture* entwickelte Unit Tests, welche direkt auf die C# Klassen zugreifen. Des Weiteren wird durch die Verwendung von installierten Services durch Tests mit dem *Robot Framework* der Datenbestand potenziell verändert. Aus diesem Grund müssten weitere Vorkehrungen getroffen werden, um den Datenbestand vor jedem Test in einen bekannten Zustand zu versetzen, wenn die vollständige Replizierbarkeit von Tests erreicht werden soll. Dies schränkt weiterhin die Effektivität von Shrinking ein, weil sich der Datenbestand zwischen Shrinking-Iterationen ändern kann und somit die gleichen Operationen nicht garantiert zu dem gleichen Ergebnis führen, und erschwert spätere Debuggingmaßnahmen. In Kombination mit dem *PresentationEngineFixture* verwendete Fake-Services können von dem Entwickler festlegbare Responses zurückgeben und somit die Replizierbarkeit von Tests einfacher ermöglichen.

Wie in Abschnitt 2.3.3 zur Einordnung der Testlösungen in die Test-Pipeline beschrieben, werden Unit Tests bereits in der ersten Stufe, im Gegensatz zu End-to-End-Tests in der zweiten Stufe, ausgeführt. Durch die Wahl des *PresentationEngineFixture* als Testlösung würden Entwickler daher schneller über Fehler informiert werden und die Bereitschaft die Tests lokal auszuführen, ist für schnelle Unit-Tests höher als für langsame Tests. Des Weiteren müssen für die Ausführung der beschriebenen Unit Tests weniger Abhängigkeiten installiert werden.

Auf Basis der angeführten Punkte, besonders in Bezug auf Geschwindigkeit und Replizierbarkeit, wird für die Implementierung der prototypischen Lösung in dieser Arbeit das *PresentationEngineFixture* verwendet.

4.3 Operationen zur Bedienung von interaktiven Elementen

Durch die Wahl des PresentationEngineFixture muss, wie beschrieben, die Bedienung der interaktiven Elemente selbst implementiert werden. Für die Bedienung der interaktiven Elemente wurden sogenannte Operationen erstellt. Alle Ausprägungen der interaktiven Elemente verfügen jeweils über eine eigene zugehörige Operation.

```
public interface IOperation {
  void Perform(object step);
}
```

Auszug 7: *IOperation* Interface

Diese Operationen müssen das *IOperation* Interface erfüllen und über eine Perform-Methode verfügen, welche den aktuellen Dialogschritt entgegennimmt und auf diesem die Operation durchführt. Die Angaben zum interaktiven Element und den zu verwendenen Werten werden über den jeweiligen Konstruktor während der Operationengenerierung übergeben. Im Folgenden werden die Details der Umsetzung von ausgewählten Operationen beschrieben.

4.3.1 FillTextBoxOperation

Die einfache TextBox gehört zu den am häufigsten vorkommenden interaktiven Elementen. Über den Konstruktor der Operation werden Metadaten zu der TextBox und ein String, welcher als Feldwert genutzt werden soll, übergeben und gespeichert. Von den Metadaten sind für die Durchführung der Befüllung nur die zwei Angaben zur Bindung des Feldwertes an eine Eigenschaft und den Aktivierungsstatus des ReadOnly-Modus von Bedeutung. Nur wenn der ReadOnly-Modus deaktiviert ist, wird der Feldwert aktualisiert. Dadurch wird das Verhalten des Endnutzers emuliert, welcher ein sich im ReadOnly-Modus befindliches Feld nicht über die Oberfläche aktualisieren kann.

Das beschriebene Verhalten kann als ein Muster interpretiert werden. Die Operation erhält alle benötigten Daten über den Konstruktor. In der Perform-Methode muss nur noch der Feld-Wert angepasst werden, wenn das Feld von dem Benutzer veränderbar ist. Diesem Muster folgen alle Operationen, welche unabhängig von dem aktuellen Feldwert durchgeführt werden können.

4.3.2 ChooseFromDropDownOperation

Zum Auswählen von Einträgen aus einem DropDown ist die aktuelle Auswahl nicht von Bedeutung. Für die Durchführung müssen jedoch die verfügbaren Auswahlmöglichkeiten bekannt sein. Diese können dynamisch während der Verwendung des Dialogschritts geändert werden und stehen somit nicht bereits während der Operationengenerierung zur Verfügung. Als Ersatz für die direkte Angabe der auszuwählenden Einträge wird eine Liste von Integern entgegengenommen. In der Perform-Methode werden die Auswahlmöglichkeiten bestimmt und gespeichert. Die Einträge der Integer-Liste werden als Indizes interpretiert und die korrespondierenden Einträge ausgewählt. Dabei wird beachtet, ob das DropDown Einfach- oder Mehrfachauswahl unterstützt.

4.4 Gemeinsamer Testablauf

Durch eine Vorabanalyse aller für die einzelnen Eigenschaften notwendigen Schritte wurde ein gemeinsamer Testablauf spezifiziert, welcher zur Umsetzung der einzelnen Eigenschaften erweitert werden kann.

Abbildung 26: Prozessübersicht des Testablaufs

Bevor die Dialogschritte getestet werden können, müssen diese zuerst erkannt und alle verfügbaren interaktiven Elemente ausgelesen werden. Anschließend wird die Test-Schleife durchgeführt und das Ergebnis angzeigt. Auf die Details der Implementierung wird im Folgenden eingegangen.

4.4.1 Erkennung von Dialogschritten

Zur Erkennung von Dialogschritten wird der Ordner des ProcessArtifacts-Projekt, dessen Pfad im Rahmen des Anschluss in zu testenden Komponenten festgelegt wird, auf zu Schritten gehörende XML Dateien untersucht. Dies geschieht über die Dateiendung, welche immer *.ds* lauten muss. Innerhalb der XML Dateien sind als Attribute des Wurzelelements Name und Namespace der Schritte angegeben, welche zur eindeutigen Identifikation der zugehörigen C# Klassen dienen. Alle exportierten Klassen der ProcessArtifacts-Assembly, auf welche ein Verweis während der Anbindung übergeben wird, werden auf Übereinstimmung mit dem Identifikator überprüft. Dabei stimmt die Klasse überein, welche sich im richtigen Namespace befindet und deren Name mit dem Namen des Dialogschritts, gefolgt von *StepActivity*, endet.

4.4.2 Auslesen von interaktiven Elementen

Nach der Identifikation der XML Dateien sowie der zugehörigen C# Klassen werden alle interaktiven Elemente extrahiert. Jede XML Datei verfügt über eine Sektion, welche das Layout der Oberfläche festlegt. Innerhalb dieser Sektion befinden sich einzelne Elemente und Container, welche weitere Elemente beinhalten können. Zur Extraktion der interaktiven Elemente wird diese Baumstruktur in eine eindimensionale Liste überführt. Aus dieser werden alle interaktiven Elemente, erkennbar an der Bezeichnung des Elements, entnommen und in einem eigens für diesen Zweck erstelltem Objekt mit Listen für die einzelnen Elementtypen abgelegt.

4.4.3 Ausführung der Test-Schleife

```
void DoesNotThrow(Random.StdGen fsCheckSeed, IOperation[] ops)
{
  // Test-Logik
}

var fsCheckSeedArb = <Generator für FsCheck Seeds>;
var operationsArb = <Generator für Liste von Operationen>;
Prop.ForAll(fsCheckSeedArb, operationsArb, DoesNotThrow)
  .Check(fsCheckConfiguration);
```

Auszug 8: Aufbau der Test-Schleife

Die Test-Schleife kann in zwei Ebenen unterteilt werden. Auf der obersten Ebene findet die Testdurchführung durch FsCheck statt. Für die Testdaten bekommt FsCheck zwei Generatoren, einen für Seeds zur weiteren Datengenerierung und einen für Listen von Operationen, übergeben. Die Überprüfung der zu testenden Eigenschaft findet in der *DoesNotThrow* Methode statt. Diese Methode stellt die zweite Ebene dar und wird in der Standardkonfiguration pro gefundenem Schritt 100 mal, mit jeweils unterschiedlichen Parametern, ausgeführt. Sie trägt den Namen *DoesNotThrow*, weil alle zu testenden Eigenschaften bei Nichteinhaltung einen Fehler werfen.

```
void DoesNotThrow(Random.StdGen fsCheckSeed, IOperation[] ops)
{
  using (var fixture = new PresentationEngineFixture())
  {
    var step = fixture.CreateAndOpenStep<TStepType>();

    foreach (var op in ops)
    {
      try
      {
        fixture.ProcessValueChanges(step, s => op.Perform(s));
```

```
      }
      catch (Exception e)
      {
        throw new Exception($"Exception nach Operation: {op}", e);
      }

      foreach (var hook in postOperationExecutionHooks)
      {
        fixture.ProcessValueChanges(step, s => hook(operatableStep, s));
      }
    }
  }
}
```

Auszug 9: Implementierung von *DoesNotThrow* Methode

In dieser Methode wird eine neue Instanz des PresentationEngineFixture erstellt und mit diesem der zu testende Dialogschritt geöffnet. Zum Öffnen wird die *PresentationEngineFixture.CreateAndOpenStep* Methode verwendet. Diese erzeugt eine neue Instanz der Step-Klasse, ruft die Methoden *OnInitialize()*, *SetInput(input)*, wenn ein Inputwert festgelegt wurde, und anschließend *OnEnter()* der Step-Klasse auf. Die genannten während der Initialisierung aufgerufenen Methoden können frei wählbare Fachlogik implementieren. Typischerweise wird *OnInitialize()* zum Laden und Vorbereiten von allgemeinen Daten genutzt, während *SetInput(input)* zur Verarbeitung von Inputwerten und dem Laden von spezifischen Daten genutzt wird. Mit dem Öffnen ist die Initialisierung der Dialogschritt-Instanz abgeschlossen. In einer Schleife wird über alle Operationen iteriert. Jede Operation wird einzeln durchgeführt, wobei die Ausführung von der *PresentationEngineFixture.ProcessValueChanges* Methode umschlossen ist. Innerhalb des Schritts können Felder von anderen Feldern abhängig sein. Wenn Feld A von Feld B abhängig ist, dann muss der neue Wert von Feld A mittels einer zuvor festgelegten Methode aktualisiert werden, wenn Feld B geändert wurde. Die *ProcessValueChanges* vergleicht die Felder der Dialogschritt-Instanz vor und nach der Durchführung der Operation und aktualisiert alle abhängigen Felder. Nach der Ausführung jeder Operation werden alle registrierten Erweiterungsmethoden, für die Ausführung nach jeder Operation, aufgerufen. Weil diese Erweiterungsmethoden potenziell auch die Felder der Dialogschritt-Instanz ändern könnten, sind auch diese Aufrufe durch die *PresentationEngineFixture.ProcessValueChanges* Methode umschlossen.

Zum einfacheren Verständnis wurde die gezeigte Implementierung der *DoesNotThrow* Methode auf das Wesentliche reduziert, und das dynamische Aufrufen von Methoden via Reflektion und die Ausführung der restlichen Erweiterungsmethoden weggelassen.

4.4.4 Anzeige der Ergebnisse

Wenn kein Fehler während der Test-Schleife identifiziert wurde, gilt der Testfall als erfolgreich und in der Konsole wird die Meldung *Ok, passed 100 tests.* angezeigt, wobei 100 durch die Anzahl der Iterationen der Test-Schleife zu ersetzen ist. Im Fehlerfall gilt der Testfall als fehlgeschlagenen und die in Abschnitt 2.7.5 beschriebene Fehlermeldung wird angezeigt.

4.4.5 Eigenschaftsspezifische Erweiterungen des Testablaufs

Die in Abbildung 27 eingezeichneten Ankerpunkte sind Erweiterungspunkte, welche Eingriffe in den Testablauf ermöglichen und zur Umsetzung von eigenschaftsspezifischer Logik dienen.

Abbildung 27: Prozessübersicht des Testablauf mit Ankerpunkten

Der erste Ankerpunkt nach der Erkennung der Dialogschritte und dem Auslesen der interaktiven Elemente, ermöglicht das Filtern von irrelevanten Schritten, wie zum Beispiel Schritte ohne interaktive Elemente.

Die Generierung von bestimmten Arten von Operationen kann über den zweiten Ankerpunkt aktiviert bzw. deaktiviert werden. Mit dem dritten Ankerpunkt kann die Liste von bereits generierten Operationen verändert werden, zum Beispiel um ungewünschte Operationen zu entfernen oder weitere Operationen hinzuzufügen.

Der vierte Ankerpunkt nach der Öffnung des Dialogschritts erlaubt die Änderung des Dialog-Zustands vor der Durchführung der Operationen, sowie das Abspeichern des initialen Zustands.

Innerhalb der Operationenausführung befindet sich der fünfte Ankerpunkt, welcher

nach jeder ausgeführten Operation aufgerufen wird. Dieser kann zum Beispiel genutzt werden, um direkt auf den veränderten Schritt zugreifen zu können und den aktuellen Stand zu speichern.

Der sechste und letzte Ankerpunkt befindet sich nach der Ausführung der Operationen-Liste. Zu diesem Zeitpunkt ist noch eine Referenz auf die während der Iteration genutzte Dialogschritt-Instanz vorhanden. Wurden während der Durchführung der Operationen keine Fehler aufgedeckt, kann dieser Erweiterungspunkt genutzt werden, um weitere Anforderungen, wie einen bestimmten gewünschten Endzustand der Dialogschritt-Instanz zu überprüfen.

Die genaue Verwendung der Ankerpunkte wird in den folgenden Abschnitten zur Implementierung der einzelnen Eigenschaften weiter erläutert.

4.5 Implementierung der Eigenschaften

In den folgenden Abschnitten wird die Implementierung der in Abschnitt 3 vorgestellten Eigenschaften erläutert. Als Grundlage für die Implementierungen dient der im vorherigen Abschnitt definierte Testablauf.

4.5.1 Die Öffnung des Dialogschritts ist möglich.

Zur Beantwortung der Frage, ob ein Dialogschritt geöffnet werden kann, ist nur ein einzelner Testdurchlauf erforderlich, weil es sich um eine binäre Antwort handelt. Deshalb wurde die Anzahl der Iterationen der Test-Schleife auf einen einzelnen Durchgang reduziert. Des Weiteren zielt diese Eigenschaft nicht auf die Untersuchung der Interaktionsmöglichkeiten ab. Um die Interaktion mit den Schritten zu unterbinden, wurde eine eigene NoOp-Operation geschaffen, welche keine Aktion durchführt, aber die Anforderung des Relevanzfilter erfüllt, dass mindestens eine Operationenart generiert werden können muss. Diese NoOp-Operation ist standardmäßig deaktiviert, wird aber für diese Eigenschaft aktiviert und alle anderen Operationen deaktiviert. Für die Beantwortung von Service-Anfragen während der Initialisierung werden Fake-Services mit generierten Responses verwendet.

4.5.2 Die Öffnung des Dialogschritts im Preview-Modus ist möglich.

Für diese Eigenschaft ist der Testaufbau identisch zur vorangegangenen Eigenschaft. Der einzige Unterschied ist die Art und Weise, wie die Schritte geöffnet werden. Während bei der vorherigen und allen folgenden Eigenschaften die Dialogschritte mit einem optional übergebenen Inputwert geöffnet werden, werden für diese Eigenschaft die Schritte im Preview-Modus geöffnet.

Zur Realisierung dieser Eigenschaft wurde der allgemeine Testablauf durch ein Attribut erweitert, welches festlegt, ob der Preview-Modus verwendet werden soll. Wenn

dieses aktivert ist, dann wird zur Öffnung von Schritten die Methode *CreateAndOpenStepWithSampleInput* des PresentationEngineFixture an Stelle von *CreateAndOpenStep* verwendet.

4.5.3 Es existiert keine Folge von validen Benutzereingaben, welche in einem Fehler resultiert.

Während bei den vorhergehenden Eigenschaften nur eine Iteration durchgeführt und die Auswahl an Operationen auf nur eine Operation ohne Auswirkungen beschränkt wird, ist bei dieser Eigenschaft gerade die Ausführung von vielen Tests unter Berücksichtigung aller möglichen Operationen interessant. Um diese Anforderungen zu erfüllen, kann der grundlegende Testablauf ohne Veränderung benutzt werden.

Für die Abgrenzung zur nächsten Eigenschaft werden installierte Services verwendet, deren Anbindung in Abschnitt 4.6.3 beschrieben wird.

4.5.4 Die Verarbeitung von beliebigen validen Service-Responses ist möglich.

Der Fokus liegt, im Gegensatz zur vorangegangenen Eigenschaft, nicht auf der Verarbeitung von Benutzereingaben, sondern auf der Verarbeitung der von den Services zurückgelieferten Daten. Während weiterhin der grundlegende Testablauf ohne Veränderung verwendet wird, werden Fake-Services eingesetzt, welche valide zum Service Interface passende Responses zurückliefern. Durch diesen Aufbau wird die gesamte Aufrufhierarchie von der auslösenden Aktion bis zur Verarbeitung der erhaltenen Daten, wie in Abbildung 12 dargestellt, getestet.

4.5.5 Die Betätigung eines eventuell vorhandenen Reset-Buttons überführt die Felder in den Initialzustand.

Für diese Eigenschaft sind nur Dialogschritte relevant, welche über einen Reset-Button verfügen. Der Reset-Button kann über die Verwendung des Standardicons für Reset-Buttons von anderen Buttons unterschieden werden.

Zur Überprüfung dieser Eigenschaft muss der Initalzustand der Felder mit deren Endzustand nach dem Reset auf Diskrepanzen überprüft werden. Der Initialzustand kann am Ankerpunkt direkt nach der Öffnung des Schritts abgefragt werden. Um zu vermeiden, dass der Reset-Button bereits durch eine generierte Operation betätigt wird, ist die Operation zur Betätigung von Buttons deaktiviert. Am Ankerpunkt nach der Ausführung der Operationen-Liste muss deshalb der Reset-Button manuell betätigt werden. Nach der Ausführung des Resets wird der Endzustand der Felder abgefragt. Mit den Informationen über den Initial- und Endzustand kann ein Vergleich durchgeführt werden. Die Erwartung ist, dass die beiden Zustände äquivalent zueindander sind.

4.5.6 Die Änderung eines Feldes in einer Suchmaske resultiert in der Änderung des Such-Requests.

Zur Überprüfung, ob eine Suchfeld-Änderung zu einer Veränderung des Such-Requests geführt hat, muss überprüft werden, ob sich ein oder mehrere Felder geändert haben und ob sich der Such-Request verändert hat.

Nach dem Öffnen der Dialogschritt-Instanz wird der Initialzustand der Felder gespeichert und dient als Datenbasis für den ersten Vergleich. Nach jeder ausgeführten Operation wird der aktuelle Stand mit dem vorherigen Stand verglichen. Alle geänderten Felder werden in einer lokalen Variable gespeichert, an eine globale Liste angehängt und der vorherige Stand mit dem aktuellen Stand überschrieben. Anschließend werden alle Buttons, mit Ausnahme von Reset-Buttons, betätigt, um Such-Requests auszulösen. Ohne tiefgreifende Code-Analyse ist es nicht möglich dynamisch herauszufinden, welcher Button oder welche Methode zur Auslösung von Such-Requests führt. Die Betätigung von allen verfügbaren Buttons hat sich in der Praxis als guter Ersatz erwiesen, stellt jedoch eine Optimierungsmöglichkeit dar. Reset-Buttons werden ausgeschlossen, um die gleichzeitge Veränderung aller Felder und die damit einhergehende Fehlidentifikation der zur Änderung führenden Felder zu vermeiden. Die getätigten Requests werden gesammelt und mit dem letzten vorangegangenen Request des gleichen Typen verglichen. Wenn es zu einer Veränderung des Requests gekommen ist, werden alle Namen der geänderten Felder in einer globalen Liste von requeständernden Feldern gespeichert.

Nach Durchführung aller Operationen werden die Listen der geänderten und der requeständernden Felder verglichen. Wenn Elemente nur in der Liste der geänderten Felder vorhanden sind, dann sind diese fehlerhaft. Sie haben keine Auswirkungen auf den Such-Request und verstoßen daher gegen die zu testende Eigenschaft. Wenn fehlerhafte Felder gefunden wurden, wird dies als Fehlermeldung ausgegeben.

Dieser beschriebene Ablauf musste zur Vermeidung von fehlerhaft identifizierten Feldern um weitere Einschränkungen erweitert werden. Zu diesen kam es, wenn kein Request, zum Beispiel durch greifende Validierungen, ausgelöst wurde. In diesen Fällen wurden alle Felder als fehlerhaft markiert, obwohl dies ohne Requests zum Vergleichen nicht möglich ist. Weitere Fehlidentifikationen traten auf, wenn ein Request-Typ zum ersten Mal registriert wurde. Ohne einen vorherigen Request des gleichen Typs kann keine Aussage über eine Veränderung getroffen werden. Aus diesem Grund werden die geänderten Felder nur in die globale Liste eingetragen, wenn Requests ausgelöst wurden und diese alle vergleichbar sind.

4.6 Unterschiedliche Servicearten

Zur Umsetzung der Eigenschaften wurden drei unterschiedliche Arten von Services benötigt, welche auf einer geteilten abstrakten Basisklasse basieren. Die Arten und der Aufbau der einzelnen Services werden in den folgenden Abschnitten erläutert.

4.6.1 Abstrakte Basisklasse *ServiceSubstitute<TService>*

```
public PresentationEngineFixture
  RegisterWebserviceSubstitute<TIService>(TIService serviceInterface)
  where TIService : class
```

Auszug 10: Methode zur Registrierung von Services am PresentationEngineFixture

Das PresentationEngineFixture stellt die *RegisterWebserviceSubstitute<TIService>* Methode zur Registrierung von Services bereit, wobei *TIService* das Service Interface darstellt. Der einfachste Weg, um dynamisch ein Objekt zu erstellen, welches das Service Interface erfüllt, ist die Verwendung einer Mocking-Bibliothek. Als Mocking-Bibliothek wurde *moq* ausgewählt, weil diese innerhalb von Schleupen bereits zu genau diesem Zweck weitverbreitet eingesetzt wird.

```
var mock = new Moq.Mock<ITaskQueryService>(MockBehavior.Strict);
```

Auszug 11: Erstellung einer strikten *Moq.Mock* Instanz

Während eine *Moq.Mock<TIService>* Instanz das reine Service Interface implementiert, würde bei einem Aufruf der Servicemethoden ein Fehler geworfen. Dies wird durch die Angabe des Parameter *MockBehavior.Strict* aktiviert. Bei Weglassung des Parameter oder der Angabe von *MockBehavior.Loose* wird keine Fehlermeldung geworfen, sondern der Defaultwert für den Rückgabeparameter, bei Objekten der Null-Wert, zurückgegeben. Die Rückgabe des Defaultwert für Serviceanfragen ist ein ungewünschtes Verhalten und kann bei korrekter Implementierung der Services nicht vorkommen, weshalb der strikte Modus gewählt wurde.

```
mock.Setup(x => x.Query(It.IsAny<QueryRequest>()))
.Returns(new QueryResponse{
  Tasks = new List<TaskContract>{
    new TaskContract{
      Id = Guid.NewGuid(),
      Description = "Tolle Beschreibung"
    }
  }
});
```

Auszug 12: Setup der *ITaskQueryService.Query* Servicemethode

Für jede Servicemethode muss ein Setup erfolgen, welches die zurückzugebende Antwort festlegt. Die Logik zur Verarbeitung von Requests und der Generierung von Antworten ist die einzige Stelle, welche sich von Serviceart zu Serviceart unterscheidet. Somit kann die restliche Logik in einer zwischen den Services geteilten Basisklasse, der *ServiceSubstitute<TService>* Klasse, implementiert werden.

```
public Type ServiceType;
public TService Substitute;
public void SetSeed(Seed newSeed);
```

Auszug 13: Öffentliches Interface der *ServiceSubstitute<TService>* Klasse

Die *ServiceSubstitute<TService>* Klasse verfügt nur über ein sehr schmales öffentliches Interface, welches einem den Zugriff auf den Typen des Service Interface, ein Objekt, welches das Service Interface implementiert, und eine Methode zum Aktualisieren des Seeds für Servicearten, welche Daten generieren, ermöglicht.

```
typeof(TService)
    .GetMethods(BindingFlags.Instance | BindingFlags.Public)
```

Auszug 14: Ermittlung der Servicemethoden

Intern werden die Mock-Instanz und der Seed initialisiert sowie eine Methode aufgerufen, welche Setups für alle Servicemethoden durchführt. Als Servicemethoden werden alle öffentlichen Funktionen des Service Interface angesehen. Für jede dieser Servicemethoden wird eine Methode ermittelt, welche das entsprechende Request-Objekt als Parameter entgegennimmt und das Response-Objekt zurückliefert. Diese werden an der Mock-Instanz zur Ermittlung der Rückgabewerte für die Servicemethoden hinterlegt.

```
protected abstract Func<TRequest, TResponse>
  GetResponseFunc<TRequest, TResponse>()
  where TResponse : class where TRequest : class;
```

Auszug 15: Signatur der abstrakten *GetResponseFunc*

Zur Ermittlung der Methode, welche das Request-Objekt als Parameter entgegennimmt und das Response-Objekt zurückliefert, wird die *GetResponseFunc* Methode mit dem Request und Response Typen als Typparameter aufgerufen. Da die Logik zur Verarbeitung von Requests und der Generierung von Antworten, wie erwähnt, für jede Serviceart unterschiedlich ist, muss diese Methode durch die erbenden Klassen implementiert werden. Dies wird in C# durch die Markierung der Methode und

Klasse als *abstract* erzwungen. Wie die *GetResponseFunc* von den einzelnen Servicearten implementiert wird, wird in den folgenden Abschnitten erklärt.

4.6.2 Services mit generierten Antworten

Zur Bereitstellung von Services ohne Abhängigkeiten zum lokalen System werden Services mit generierten Antworten verwendet.

```
[System.ServiceModel.ServiceContractAttribute(Namespace="urn://Namespace.
    TaskQueryService_1.0", ConfigurationName="TaskQueryService.V1_0.
    ITaskQueryService")]
public interface ITaskQueryService
{
  [System.ServiceModel.OperationContractAttribute(Action="urn://Namespace.
      TaskQueryService_1.0/QueryById", ReplyAction="urn://Namespace.
      TaskQueryService_1.0/QueryByIdResponse")]
  TaskQueryService.V1_0.QueryByIdResponse QueryById(TaskQueryService.V1_0.
      QueryByIdRequest request);
}
```

Auszug 16: Vereinfachtes Beispiel eines generierten Service Interface

Als Grundlage für die Generierung der Antworten dienen in den Gateway-Bibliotheken enthaltene Service Interfaces. Deren Aufbau und Angaben zu optionalen und verpflichtenden Fehldern werden analysiert und Response-Objekte erzeugt. Diese Funktionalität wird durch die *ResponseGenerator* Klasse bereitgestellt, deren genaue Funktionsweise in Abschnitt 4.7 erklärt wird.

```
public class GeneratorService<TService> : ServiceSubstitute<TService>
  where TService : class
{
  public void OverrideResponseGenerator<TRequest, TResponse>(
    ResponseGenerator<TRequest, TResponse> generator)
    where TRequest : class
    where TResponse : class {
    SetupServiceMethod<TRequest, TResponse>(
      req => generator.Generate(Seed, req));
  }

  protected override Func<TRequest, TResponse>
    GetResponseFunc<TRequest, TResponse>() {
    var generator = new ResponseGenerator<TRequest, TResponse>();
    return req => generator.Generate(Seed, req);
  }
}
```

Auszug 17: *GeneratorService* Klasse

Die *GeneratorService* Klasse setzt diese Art von Service um. Für die Generierung der Antworten wird standardmäßig eine Instanz der *ResponseGenerator* Klasse ohne Filterfunktion verwendet. Wenn eine Filterfunktion verwendet werden soll, zum Beispiel aus einem der in Abschnitt 4.7.3 vorgestellten Gründe, kann der Standardgenerator mittels der *OverrideResponseGenerator* Methode überschrieben werden.

4.6.3 Installierte Services

Installierte Services ermöglichen den Zugriff zu auf dem lokalen System installierte echte Services. Während theoretisch alle möglichen validen Service-Antworten durch Services mit generierten Antworten abgedeckt werden können, kann es von Vorteil sein gegen echte Services zu testen. Besonders durch bereits bestehende Infrastruktur zum Anlegen von Testdaten im Real-System können einfacher komplexe Testfälle aufgebaut werden. Durch die Verwendung von echten Services muss nicht jede Antwort der Services explizit festgelegt werden.

```
public class InstalledService<TService> : ServiceSubstitute<TService>
  where TService : class
{
  private readonly IWebserviceProxy proxy;
  private readonly string sessionToken;

  public InstalledService(string sessionToken) {
    this.sessionToken = sessionToken;
    proxy = BuildWebserviceProxy();
  }

  protected override Func<TRequest, TResponse>
    GetResponseFunc<TRequest, TResponse>() {
    var serviceMethod = MustGetServiceMethod<TRequest, TResponse>();

    return req =>
    {
      ((dynamic) req).SessionToken = sessionToken;
      return (TResponse) proxy.CallMethod(serviceMethod.Name,
        new[] { typeof(TRequest) }, new object[] { req });
    };
  }

  private IWebserviceProxy BuildWebserviceProxy() {
    // Technische Details der Schleupen.CS Plattform
  }
}
```

Auszug 18: *InstalledService* Klasse

Der Zugriff auf installierte Services ist in der *InstalledService* Klasse als direkter Zugriff über ein Proxy-Objekt umgesetzt. Das Proxy-Objekt wird über den Konstruktor spziell für einen Service erstellt und ermöglicht über den Aufruf der *CallMethod* Methode mit dem Namen der Servicemethode und dem Request-Objekt als Parameter den Zugriff auf diese Servicemethode. Zur genauen Identifikation des für den Request zu verwendenen Knotenpunkt der Systemstruktur enthält jeder Request einen SessionToken. Im Konstruktor wird das übergebene SessionToken abgespeichert und das Proxy-Objekt erzeugt. In der *GetResponseFunc* Methode wird die zu den Typparametern passende Servicemethode herausgesucht und die Methode zur Ermittlung des Response zurückgegeben. Diese Methode setzt im Request das SessionToken, ruft über das Proxy-Objekt die Servicemethode mit dem übergebenen Request auf und gibt die erhaltene Antwort zurück.

4.6.4 Abhörbare Services

Abhörbare Services können nicht selber Antworten generieren, sondern sind lediglich Wrapper um andere Servicearten. Sie existieren, um die an Services gestellten Requests abzufangen, ohne die originale Serviceart um derartige Logik erweitern zu müssen.

```
public class InterceptedService<TService> : ServiceSubstitute<TService>
  where TService : class
{
  private readonly IReadOnlyList<Action<object>> interceptors;
  private readonly TService substitute;

  public InterceptedService(ServiceSubstitute<TService> serviceSubstitute,
      IEnumerable<Action<object>> interceptors) {
    this.substitute = serviceSubstitute.Substitute;
    this.interceptors = interceptors.ToList();
  }

  protected override Func<TRequest, TResponse> GetResponseFunc<TRequest,
      TResponse>() {
    var serviceMethod = MustGetServiceMethod<TRequest, TResponse>();

    return req => {
      interceptors.ForEach(x => x(req));

      return (TResponse)serviceMethod.Invoke(substitute, new object[] { req });
    };
  }
}
```

Auszug 19: *InterceptedService* Klasse

Im Konstruktor wird eine Liste von Methoden übergeben, welche ein Objekt, den Request, entgegennehmen. Der Datentyp *object* muss hier eingesetzt werden, weil die Interzeptoren nicht Request-spezifisch sind und die Request-Objekte keine geteilte Oberklasse haben. In der *GetResponseFunc* Methode wird das Request-Objekt an jeden der Interzeptoren übergeben, bevor der gewrappte Service zur Ermittlung des Response aufgerufen wird.

4.7 Generierung von Response-Objekten

Zur Umsetzung der Services mit generierten Antworten muss für jeden Request dynamisch ein Response-Objekt generiert werden, welches das Service Interface und über eine Filterfunktion definierbare weitere Eigenschaften erfüllt. Zusätzlich muss für die Replizierbarkeit des Gesamttests diese Generierung replizerbar erfolgen. Die Umsetzung dieser Anforderungen wird in den folgenden Abschnitten erläutert.

4.7.1 Vergleich von AutoFixture und FsCheck Generatoren

AutoFixture ist eine Bibliothek zur automatischen Generierung von Testdaten und wird bereits in mehreren Komponenten für diesen Zweck verwendet. Über die Funktionalität zur Generierung von Objekten ist es möglich, das Response-Objekt zu erzeugen. Zur Erreichung der zuvor festgelegten Anforderungen fehlt AutoFixture die Möglichkeit der replizerbaren Generierung. Über die Erweiterungsbibliothek *AutoFixture.SeedExtensions* können Seeds angegeben werden, welche jedoch nur als Präfixe verwendet werden und weiterhin bei jedem Aufruf zu unterschiedlichen Ergebnissen führen. Zusätzlich beziehen sich die SeedExtensions nicht auf alle Datentypen. Des Weiteren ist es nicht möglich, bereits während der Generierung die Befüllung von Pflichtfeldern zu erzwingen, wodurch eine erweiterte Filterfunktion und längere Laufzeiten bedingt werden.

FsCheck als Bibliothek für Eigenschaftsbasierte Tests verwendet eine eigene Generatoren-Struktur zur Generierung der Testdaten. Diese Generatoren können, über einen Seed-Wert gesteuert, replizerbare Werte generieren und unterstützen die Anwendung von Filterkriterien zur Generierungszeit. Über die bewusste Kombination von Generatoren und Filterkriterien kann ein Generator für das Response-Objekt erstellt werden, welcher Pflichtfelder berücksichtigt und zum Service Interface passende Responses erzeugt.

Aus diesen Gründen wurden zur Implementierung der Response-Generierung FsCheck Generatoren verwendet.

4.7.2 Aufbau des Response Generators

```
[System.ServiceModel.MessageContractAttribute(WrapperName="QueryByIdResponse",
    WrapperNamespace="urn://Namespace.TaskQueryService_1.0", IsWrapped=true)]
public class QueryByIdResponse
{
  [System.ServiceModel.MessageBodyMemberAttribute(Namespace="urn://Namespace.
      TaskQueryService_1.0", Order=0)]
  public System.Collections.Generic.List<TaskQueryService.V1_0.TaskContract>
      TaskListe;
}
```

Auszug 20: Vereinfachtes Beispiel einer generierten Response-Klasse

Zum Aufbau des Response Generators wird die aus dem Service Interface generierte Klassenhierarchie via Reflektion analysiert. Diese Analyse beginnt mit der Response-Klasse, welche über das *System.ServiceModel.MessageContractAttribute* Attribut als Message Contract gekennzeichnet ist. Im hier gezeigten Beispiel enthält die Klasse eine List-Property mit dem Elementtypen TaskContract. Um diese Klasse zu generieren, werden ein Generator für TaskContracts und ein Generator für Listen benötigt. Ein generischer Generator für Listen wird von FsCheck bereitgestellt, welcher Listen mit null oder mehr Elementen generiert. Für die Umsetzung des Service Interface müssen diese Listen gefiltert und möglicherweise angepasst werden, weil bei optionalen Feldern leere und gefüllte Listen sowie der Null-Wert und für Pflichtfelder nur gefüllte Listen zugelassen sind.

```
[System.Runtime.Serialization.DataContractAttribute(Name="TaskContract",
    Namespace="urn://Namespace.TaskQueryService_1.0")]
public class TaskContract
{
  [System.Runtime.Serialization.DataMemberAttribute(IsRequired=true)]
  public System.Guid Id { get; set; }

  [System.Runtime.Serialization.DataMemberAttribute(IsRequired=true)]
  public string Title { get; set; }

  [System.Runtime.Serialization.DataMemberAttribute()]
  public string Description { get; set; }
}
```

Auszug 21: Vereinfachtes Beispiel einer generierten Contract-Klasse

Für die TaskContract Klasse existiert kein fertiger Generator, weshalb dieser durch Analyse der Klasse zusammengebaut werden muss. Die Klasse besteht aus den drei Properties Id, Title und Description, wobei Id und Title Pflichtfelder sind. Anhand der Eigenschaft *IsRequired* des *System.Runtime.Serialization.DataMemberAttribute*

Attribut wird angegeben, ob es sich um ein Pflichtfeld handelt. Alle drei Properties verwenden Datentypen der Systembibliothek, welche durch bereitgestellte Generatoren von FsCheck abgedeckt werden. Die Generatoren für die einzelnen Properties werden zu einem Generator für die TaskContract Klasse kombiniert. Wenn eine Property kein Systemtyp ist, wird das beschriebene Verfahren rekursiv für den Datentypen dieser Property angewandt, bis Generatoren für alle Datentypen gefunden oder erzeugt wurden.

Nach dem beschriebenen Schema wird eine hierarchische Generatoren-Struktur aufgebaut, welche die Hierarchie der Response-Klasse widerspiegelt. Durch Berücksichtigung der Angaben zu Pflichtfeldern erzeugt der Geneator nur zum Service Interface passende Objekte. Weitere Einschränkungen für die generierten Objekte können über eine Filterfunktion umgesetzt werden.

4.7.3 Filterfunktion

Durch das Service Interface nicht abgedeckte Einschränkungen, welche für einen validen sinnvollen Response nötig sind, müssen durch eine Filterfunktion umgesetzt werden. Die Filterfunktion erhält als Argumente den Request und den generierten Response und gibt über die Rückgabe eines Booleans an, ob der generierte Response alle Anforderungen erfüllt. Entspricht der generierte Response nicht den Anforderungen der Filterfunktion, wird ein neuer Response generiert und erneut an die Filterfunktion übergeben. Dies wird wiederholt, bis ein passender Response gefunden wird.

Ein Beispiel für eine Situation, welche den Einsatz einer Filterfunktion erfordert, ist die Durchführung einer Suchanfrage. Wenn im Request explizit mit Ids gesucht wird, dann dürfen im Response nicht mehr von den gesuchten Objekten als Ids vorhanden sein und die Ids der gefundenen Objekte müssen zu den gesuchten passen.

Eine reine Filterfunktion für diese Einschränkungen reicht in diesem Beispiel nicht aus, weil diese zu Laufzeitproblemen führen wird. Die für Ids verwendeten *System.Guids* sind in C# als 128 Bit Integer implementiert [12] und die Wahrscheinlichkeit, die richtige Id zufällig zu generieren liegt bei $2^{-128} < 10^{-36}\%$. Aus diesem Grund muss bei deratigen Bedingungen, deren Erfüllung durch zufällige Generierung sehr unwahrscheinlich ist, das als Argument übergebene generierte Response-Objekt direkt modifiziert werden. Für obiges Beispiel könnte dies erreicht werden durch die Entfernung aller überflüssigen generierten Objekte, die Überschreibung der generierten Ids durch die gesuchten Ids und die Ablehnung aller generierten Responses mit zu wenigen Objekten. Die direkte Modifikation des Responses kann auch bei anderen Bedingungen eine Optimierungsmöglichkeit zur Laufzeitverbesserung darstellen.

4.8 Anschluss in zu testenden Komponenten

Für den Anschluss der Lösung muss in der zu testenden Komponente innerhalb des *ProcessArtifacts.Tests* Projekt eine neue Klasse erstellt werden, welche von der abstrakten *PropertyTestBase* Klasse erbt. Die PropertyTestBase Klasse verfügt über die beiden abstrakten Properties *ProcessArtifactsAssembly* und *ProcessArtifactsPath*. Erstere muss einen Verweis auf die ProcessArtifacts-Assembly und *ProcessArtifactsPath* den Pfad zum ProcessArtifacts-Ordner zurückliefern.

```
namespace Namespace.Der.Zu.Testenden.Komponente
{
  using NUnit.Framework;
  using Schleupen.CS.PI.Framework.Testing;
  using Schleupen.CS.PI.PR.PropertyTesting;
  using System.Reflection;

  [TestFixture]
  [Category(TestCategory.ContinuousIntegration)]
  public sealed class PropertyTests : PropertyTestBase
  {
    protected override Assembly ProcessArtifactsAssembly =>
      typeof(ProcessArtifactsDependency).Assembly;

    protected override string ProcessArtifactsPath =>
      GetProcessArtifactsPath();
  }
}
```

Auszug 22: Minimaler Anschluss in zu testender Klasse

Zur Ermittlung des Pfades zum ProcessArtifacts-Ordner wird die in der *PropertyTestBase* Klasse implementierte *GetProcessArtifactsPath* Funktion verwendet.

```
public static string GetProcessArtifactsPath(
  string relPath = @"..\ProcessArtifacts",
  [CallerFilePath] string path = null)
{
  var dir = Path.GetDirectoryName(path);
  return Path.GetFullPath(Path.Combine(dir, relPath));
}
```

Auszug 23: Funktion zum Abrufen des Pfad zum ProcessArtifacts-Ordner

Diese Funktion verfügt über zwei optionale String-Parameter. Über den ersten Parameter kann der Standardpfad zum ProcessArtifacts-Ordner relativ zum Ordner der aktuellen Datei überschrieben werden. Standardmäßig liegen die beiden Ordner im

gleichen Überordner und der Pfad zum ProcessArtifacts-Ordner kann durch einen Wechsel in den Überordner und dann in den ProcessArtifacts-Ordner ermittelt werden. Eine Überschreibung ist nötig, wenn die Testdatei in einem Unterordner des Test-Projekts liegt. Der zweite Parameter ist mit dem *System.Runtime.CompilerServices.CallerFilePathAttribute* Attribut ausgezeichnet. Dieses weist den Compiler an, den Parameter mit dem vollständigen Pfad der aufrufenden Datei, der erbenden Klasse, zur Zeit der Kompilierung zu füllen [11]. Mit diesen beiden Angaben bestimmt die Methode den absoluten Pfad.

5 Fallstudie

Die prototypische Lösung zur automatisierten Überprüfung der Portal UI auf Fehlerfreiheit und Stabilität wurde im Rahmen einer Fallstudie auf die sich im produktiven Einsatz befindliche GP-Komponente MWM.MML angewandt.

5.1 Analyse der vorhandenen Dialogschritte

Insgesamt werdern durch die GP-Komponente MWM.MML 32 Dialogschritte bereitgestellt. Acht der Schritte verfügen über Suchformulare, wobei alle von diesen einen Reset-Button bereitstellen. Nur zwei Schritte verfügen nicht über interaktive Elemente und drei verwenden keine Services. Bei diesen Dialogschritten ohne interaktive Elemente handelt es sich um Dialoge zur reinen Anzeige von Daten. Dies trifft auch auf zwei der drei Dialoge ohne Abhängigkeiten zu Services zu. Bei dem dritten handelt es sich um einen Dialogschritt zur Erfassung von Messwerten, wobei alle zur Erfassung benötigten Daten von dem übergeordneten Dialogablauf bereitgestellt und alle erfassten Daten wieder an diesen zurückgegeben werden. Durch diesen Aufbau ist die direkte Verwendung von Services nicht erforderlich. Wie die Tatsache, dass nur drei von 32 Schritten keine direkten Abhängigkeiten zu Services haben, zeigt, ist dieser Aufbau selten. Im Normalfall werden nur Ids oder eine Teilmenge der benötigten Daten als Input übergeben und die fehlenden Daten durch die Dialoge selbstständig nachgeladen.

5.2 Benötigte Anpassungen des Testaufbaus

Wie im vorherigen Abschnitt erläutert, erhalten viele der Dialogschritt eine Teilmenge ihrer benötigten Daten über Inputparameter. Im Kontext der Portal UI starten die meisten Dialogabläufe mit einem Schritt, welcher keinen Input benötigt und zur Auswahl einer zu bearbeitenden Entität dient. Die Daten zur ausgewählten Entität werden an den nächsten Dialog im Dialogablauf durchgereicht. Auf diese Weise entsteht eine Verkettung der einzelnen Schritte.

Durch die Betrachtung jedes Dialogschritts als eigenständige Entität entfällt diese

Verkettung. Aus diesem Grund müssen die Inputparameter in der erbenden Testklasse für die Schritte angegeben werden. Durch die Weitergabe von Daten entstehen weitere Annahmen, welche die Einzelbetrachtung der Dialogschritte aufweicht.

Spätere Schritte werden unter den Annahme implementiert, dass die zuvor ausgewählten Entitäten vorhanden sind. Diese Annahme ist als vertretbar zu erachten und bedingt somit die Berücksichtigung bei der Generierung von Service Responses. Eine Analyse der relevanten Serviceaufrufe ergab, dass zum Laden der Entitäten deren Ids oder eindeutige Bezeichner verwendet werden. Während dies nicht durch das Service Interface garantiert ist, sollen die Services mit generierten Antworten in diesen Fällen immer die angefragten Entitäten zurückliefern. Diese zusätzlichen Anforderungen wurden durch die Einführung von Filterfunktionen für die entsprechenden Services realisiert.

Das für Tests gegen installierte Services benötigte SessionToken konnte durch bereits vorhandene Test-Helferklassen abgerufen und festgelegt werden.

5.3 Gefundene Fehler

Durch die in der Lösung implementierten Tests zu den Eigenschaften konnten mehrere Fehler gefunden werden. Die unterschiedlichen Fehlerarten werden im Folgenden vorgestellt.

5.3.1 Optionale Listen-Felder können Null sein

Optionale Felder mit dem Datentypen *List<T>* können Listen beliebiger Länge oder den Null-Wert beinhalten. Durch die Tests zur Eigenschaft *Die Verarbeitung von beliebigen validen Service-Responses ist möglich.* wurden mehrere Assembler identifiziert, welche Null-Werte nicht verarbeiten konnten.

```
public List<ViewModel> ToViewModel(List<Contract> contracts){
  return contracts.Select(ToViewModel).ToList();
}

private ViewModel ToViewModel(Contract contract){
  return new VieModel(contract.Property, ...);
}
```

Auszug 24: Beispiel für fehlerhafte Verarbeitung von Null-Listen

Der fehlerhafte Assembler-Code folgte dem obigen Muster, welches in einer Methode die Liste entgegennimmt, über die einzelnen Elemente iteriert und die Ergebnisse als neue Liste zurückgibt. Wenn die übergebene Liste *contracts* ein Null-Wert ist, dann tritt eine NullPointerException auf, weil der Aufruf von Select auf diesem scheitert.

Um dieses Problem abzufangen, muss vor der Iteration geprüft werden, ob eine Liste vorhanden ist, sodass nur dann die Iteration durchgeführt wird.

5.3.2 Optionale Felder als Pflichtfelder verwenden

Das Typensystem von C# erlaubt für einige der eingebauten Datentypen, wie zum Beispiel Strings, die Verwendung von Null-Werten an Stelle von normalen Werten. Dadurch ist die Verwendung von optionalen Feldern als Pflichtfelder häufig ein schwer zu entdeckender Fehler. Ein fehlerhaft zustandegekommener Null-Wert führt bei reinen Tests des Assemblers häufig nicht direkt zu Fehlern. Erst bei der Verarbeitung der Werte in der Fachlogik treten NullPointerExceptions auf. Durch die Tests zur Eigenschaft *Die Verarbeitung von beliebigen validen Service-Responses ist möglich.* wird die gesamte Aufrufhierarchie von Empfang eines Responses bis zur Anzeige in der Oberfläche getestet und somit die Wahrscheinlichkeit der Identifikation derartiger Fehler erhöht.

In einem gefundenen Fall hat der dahinterliegende Service das optionale Feld in jedem Fall befüllt, jedoch hat das Service Interface dies nicht garantiert. Die korrekte Funktionsweise des Assembler beruhte daher auf Implementierungsdetails der Services, welche sich ohne Bekanntmachung über eine neue Version des Service Interface ändern könnten.

5.3.3 Extraktion von Daten aus generischen Datentypen

Nicht in jedem Fall enthält das Service Interface die benötigten Daten als einzelnes Feld mit dem für den eigenen Anwendungsfall optimalen Typen. In diesen Fällen ist es nötig, zum Beispiel die Jahresanzahl aus einem DateTime-Feld zu extrahieren. Zu Fehlern kann es kommen, wenn der verwendete Datentyp ein generischer Datentyp, wie ein String, ist und somit kein genaues Format garantieren kann.

```
Stichtag = $"{contractAbleseTurnus.Stichtag.Substring(0, 2)}.{
    contractAbleseTurnus.Stichtag.Substring(2, 2)}.";
```

Auszug 25: Extraktion von Datumsangaben aus generischem String

In einem Fall wird im Service Response ein Datum in dem Format “DDMM” dargestellt und sollte für die Anzeige in die Form “DD.MM.” überführt werden. Im Test zur Eigenschaft *Die Verarbeitung von beliebigen validen Service-Responses ist möglich.* kam es deshalb zu einem Fehler, weil im generierten Response anstatt eines korrekt formatierten Datums ein leerer String enthalten war.

5.3.4 Suchfelder werden nicht resettet

Im Suchformular des Dialogschritts zur Identifikation von Marktlokationen wurden durch den Test zur Eigenschaft *Die Betätigung eines eventuell vorhandenen Reset-Buttons überführt die Felder in den Initialzustand.* zwei Felder identifiziert, welche nicht resettet wurden.

```
System.Exception : Falsifiable, after 1 test (1 shrink) (StdGen (941732952,
    296838661)):
Original:
StdGen (1419326626, 1410215435)
[|Fill TextBox "ZaehlernummerTextBox" with "";
Fill DatePicker "GueltigAmDatePicker" with "03.06.2003 06:32:06"|]
Shrunk:
StdGen (1419326626, 1410215435)
[|Fill DatePicker "GueltigAmDatePicker" with "03.06.2003 06:32:06"|]
with exception:
NUnit.Framework.AssertionException:   GueltigAmDatePicker
Expected: null
But was:  2003-06-03 06:32:06.218
```

Auszug 26: Fehlermeldung zu nicht resettetem Feld

Die hier gezeigte Fehlermeldung bezieht sich auf ein Feld zur Einschränkung der Ergebnismenge auf Marktlokationen, welche zu einem ausgewählten Tag gültig sind. Anhand der Fehlermeldung ist zu erkennen, dass zur Auslösung des Fehlers ein DatePicker-Eingabefeld mit Namen *GueltigAmDatePicker* mit einem Datum gefüllt werden muss. Nach der Durchführung des Resets wird für den Feldwert der initiale Null-Wert erwartet, jedoch ist weiterhin das eingegebene Datum vorhanden.

5.3.5 Preview-Modus kann nicht geöffnet werden

Durch die Eigenschaft *Die Öffnung des Dialogschritts im Preview-Modus ist möglich.* wurden vier Dialogschritte identifiziert, welche nicht im Preview-Modus geöffnet werden konnten. Bei einem dieser Schritte wurden keine Inputdaten für den Preview-Modus festgelegt, welche jedoch benötigt werden, weil das Öffnen mit dem Default-Wert bei diesem Schritt nicht funktioniert. Bei zwei weiteren Dialogen waren die angegebenen Inputdaten selbst fehlerhaft, beziehungsweise nicht vollständig und führten zu Fehlern bei der Verarbeitung durch die Fachlogik. Der Fehler beim letzten Schritt trat durch die naive Verwendung von Services auf. Wenn der Response eines Services genau zu den Inputdaten passen muss, dann muss der echte Aufruf an den Service für den Preview-Modus ersetzt werden.

6 Ergebnisbewertung

Durch die Anwendung der entwickelten Lösung konnten in der GP-Komponente erfolgreich Fehler gefunden werden. Die gefundenen Fehler stellen keine schwerwiegenden Probleme dar und schränken somit die grundlegende Funktionalität der Software *aktuell* nicht ein. Besonders die gefundenen Fehler in Bezug auf die Assembler stellen jedoch mögliche Stabilitätsprobleme dar. Wie bereits zuvor angesprochen beruht das derzeitige Nichtauftreten von derartigen Fehlern auf Implementierungsdetails der Services und ist ein klares Optimierungspotenzial. Die Forschungsfrage **Können durch automatisierte Eigenschaftsbasierte Tests in produktiv eingesetzten Oberflächen Fehler gefunden werden?** kann somit eindeutig bejaht werden.

Die größten Probleme in der Anwendung und der Auswertung von Fehlern wurden durch Fehlermaskierung hervorgerufen. Durch den bewusst festgelegten gemeinsamen Testablauf unterscheiden sich die einzelnen Tests nur durch die unterschiedlichen Eingriffe an den Ankerpunkten. Dadurch werden Probleme an zentralen Stellen, wie den Assemblern, durch mehrere Tests aufgedeckt, auch wenn diese zur Überprüfung auf andere Fehler vorgesehen sind. Während dies die Wahrscheinlichkeit der Aufdeckung derartiger Fehler erhöht, wird dadurch die Analyse erschwert, weil mehrere fehlgeschlagene Testfälle auf eine Fehlerursache zurückzuführen sind und das Aufdecken von anderen Fehlern im späteren Verlauf des Programms verhindern. Besonders Fehler in Assemblern von häufig benutzten Services führen zu fehlgeschlagenen Testfällen für eine Vielzahl von Dialogschritten, obwohl dies keine Aussage über die Fachlogik des Dialogs selbst zulässt.

Für die Anwendung der Lösung im Rahmen der Fallstudie wurde aufgrund des zeitlichen Rahmens ein passiver Ansatz gewählt und nur die Fehler mit der größten Maskierung von weiteren Fehlern behoben. Für alle gefundenen Fehler wurden Fehlertickets für das entsprechende interne Team erstellt, welche bereits teilweise während der Entstehungszeit dieser Arbeit behoben wurden. Zur Beantwortung der Forschungsfrage dieser Arbeit ist dies der effektivste Ansatz. Zur erfolgreichen Einführung der Lösung in eine Komponente ist die parallele Einführung und direkte Beseitigung von gefundenen Fehlern empfehlenswert. Dies sollte jedoch separat untersucht werden.

Die entstandene prototypische Lösung kann eine Basis für zukünftige Entwicklungen und die Umsetzung von weiteren Eigenschaften sein. Sie hat gezeigt, dass automatisierte Tests für Dialogschritte möglich sind und Fehler entdecken können. Für eine Ausweitung auf weitere Teams muss diese Lösung weiter stabilisiert werden. Des Weiteren kann die Geschwindigkeit der Lösung weiter optimiert werden. Zum Beispiel stellt der Austausch der Mocking-Bibliothek *moq*, deren großer Featureumfang

nicht benötigt wird, jedoch aufgrund der einfachen Einbindung verwendet wird, durch eine einfachere Bibliothek eine Möglichkeit zur Performanceoptimierung dar.

Während die getesteten Dialogschritte zum Großteil seit mehreren Monaten bis Jahren im Produktiveinsatz sind und bereits Zeit zur Fehleridentifikation und Korrektur nach der initialen Entwicklungsphase hatten, können die umgesetzen Eigenschaftsbasierten Tests wahrscheinlich bei neuen und sich in der Entwicklung befindlichen Dialogschritten einen noch größeren Mehrwert liefern. Dies würde die Erfahrungen der Entwickler der QuickCheck Bibliothek für Eigenschaftsbasierte Tests in Haskell [10, Abs. 7] widerspiegeln.

Die umgesetzten Eigenschaften haben eine Vielzahl von Fehlern in Assemblern gefunden, obwohl die Tests auf einer höheren Ebene ansetzen. Deshalb sollte untersucht werden, wie effektiv Eigenschaftsbasierte Tests direkt für Assembler sind. Die für diese Arbeit implementierte Funktionalität zur Generierung von Response-Objekten kann wahrscheinlich zu diesem Zweck unverändert eingesetzt werden. Aufgrund der Verwendung von ähnlichen Angaben für Request- und Response-Objekte kann die Funktionalität wahrscheinlich mit geringen Änderungen auch für die Generierung von Request-Objekten eingesetzt werden.

7 Literaturverzeichnis

[1] *Anlage 1 zum Beschluss BK6-18-032: Geschäftsprozesse zur Kundenbelieferung mit Elektrizität (GPKE).*

[2] *Anlage 2 zum Beschluss BK6-18-032: Wechselprozesse im Messwesen Strom (WiM Strom).*

[3] T. Arts, K. Claessen, J. Hughes und H. Svensson, "Testing Implementations of Formally Verified Algorithms", 2005.

[4] T. Arts, J. Hughes, J. Johansson und U. Wiger, "Testing Telecoms Software with Quviq QuickCheck", in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, Ser. ERLANG '06, Portland, Oregon, USA: Association for Computing Machinery, 2006, 2–10, ISBN: 1595934901. DOI: 10.1145/1159789.1159792. Adresse: https://doi.org/10.1145/1159789.1159792.

[5] K. Claessen und J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs", *SIGPLAN Not.*, Jg. 35, Nr. 9, 268–279, Sep. 2000, ISSN: 0362-1340. DOI: 10.1145/357766.351266. Adresse: https://doi.org/10.1145/357766.351266.

[6] M. Cohn, *Succeeding with Agile: Software Development using Scrum.* Upper Saddle River, NJ [u.a.]: Addison-Wesley, 2010.

[7] P. Debois, J. Humble, G. Kim und J. Willis, *The DevOps Handbook: How to create world-class agility, reliability, & security in technology organizations.* Portland, OR: IT Revolution Press, LLC, 2016.

[8] *Gesetz über den Messstellenbetrieb und die Datenkommunikation in intelligenten Energienetzen (Messstellenbetriebsgesetz - MsbG).*

[9] R. Hildebrandt und A. Zeller, "Simplifying Failure-Inducing Input", in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, Ser. ISSTA '00, Portland, Oregon, USA: Association for Computing Machinery, 2000, 135–145, ISBN: 1581132662. DOI: 10.1145/347324.348938. Adresse: https://doi.org/10.1145/347324.348938.

[10] J. Hughes, "QuickCheck Testing for Fun and Profit", in *Practical Aspects of Declarative Languages*, M. Hanus, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, S. 1–32, ISBN: 978-3-540-69611-7.

[11] Microsoft, *CallerFilePathAttribute Klasse*, https://docs.microsoft.com/de-de/dotnet/api/system.runtime.compilerservices.callerfilepathattribute?view=net-5.0, Aufgerufen am 25. Januar 2021.

[12] Microsoft, *Hinweise zur C# Guid Struktur*, https://docs.microsoft.com/de-

de/dotnet/api/system.guid?view=net-5.0#remarks, Aufgerufen am 22. Dezember 2020.

[13] W3C, *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, https://www.w3.org/TR/wsdl20/, Aufgerufen am 12. Februar 2021.

[14] S. Wlaschin, *An introduction to property-based testing*, https://fsharpforfunandprofit.com/posts/property-based-testing/, Aufgerufen am 18. Januar 2021.

Selbständigkeitserklärung

Hiermit erkläre ich, Tobias Meyer, dass ich die hier vorliegende Arbeit selbstständig und ohne unerlaubte Hilfsmittel angefertigt habe. Informationen, die anderen Werken oder Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich kenntlich gemacht und mit exakter Quellenangabe versehen. Sätze oder Satzteile, die wörtlich übernommen wurden, wurden als Zitate gekennzeichnet. Die hier vorliegende Arbeit wurde noch an keiner anderen Stelle zur Prüfung vorgelegt und weder ganz noch in Auszügen veröffentlicht.

Moers, 05.04.2021 Tobias Meyer