Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Untersuchung der Schnittstellen des Konnektors für die Telematikinfrastruktur hinsichtlich Benutzbarkeit in der Anwendungsprogrammierung

Emilio Kempkes
Matrikelnummer: 5076983
emilio.kempkes@fu-berlin.de

Betreuer/in: Immanuel Sims
Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachter/in: Prof. Dr. Claudia Müller-Birn

Berlin, 13. September 2022

Zusammenfassung

Hintergrund: Der Konnektor ist eine Hardwarekomponente, die Praxen an die Telematikinfrastruktur, die zentrale digitale Infrastruktur im deutschen Gesundheitswesen anbindet. Clientsysteme auf den Computern der Praxen können auf Funktionalitäten der Telematikinfrastruktur über Schnittstellen des Konnektors zugreifen.
Ziele: In dieser Arbeit soll die Benutzbarkeit dieser Schnittstellen beim Programmieren von Clientsystemen untersucht werden.
Methoden: Zum eigenen Verständnis wurde ein einfaches CLI-Clientsystem geschrieben, das einen Teil der gebotenen Funktionalitäten anspricht. Außerdem wurden Probleme über eine heuristische Evaluation anhand zehn allgemeiner Benutzbarkeitsprinzipien ermittelt und Benutzbarkeitstests mit vier Teilnehmern durchgeführt.
Ergebnisse: Durch die Benutzbarkeitstests wurden fünf Probleme in Form von Redundanzen, schlecht gewählten Bezeichnern, zu allgemeinen Typen für Aufrufparameter und zu komplizierten Datenstrukturen entdeckt. Positive Aspekte fielen auch auf, drei von vier Teilnehmern beendeten alle Aufgaben erfolgreich. In der Evaluation wurden sechs Probleme ermittelt, die sich hauptsächlich durch Verstöße gegen die Prinzipien der Konsistenz, Nähe am mentalen Modell der Nutzer und Fehlervermeidung auszeichneten.
Schlussfolgerungen: Eine Betrachtung weiterer durch den Konnektor bereitgestellter Funktionalitäten ist notwendig, um zu einer abschließenden Bewertung zu gelangen. Die Benutzbarkeitstests könnten durch bessere Mitschriften optimiert werden.

Inhaltsverzeichnis

1 Einführung

In Deutschland soll das Gesundheitswesen digitalisiert werden. Diese Aufgabe wurde gesetzlich vorgegeben [28] und der nationalen Agentur für digitale Medizin, der gematik GmbH übertragen [24]. Deren Gesellschafter sind das Bundesministerium für Gesundheit mit 51 % der Anteile und viele weitere in der deutschen Medizinlandschaft relevante Organisationen, zum Beispiel die Bundesärztekammer oder der deutsche Apothekenverband [25].

Der genaue Auftrag der gematik nach §306 des Sozialgesetzbuches V ist das Erarbeiten der Telematikinfrastruktur. Diese ist „die interoperable und kompatible Informations-, Kommunikations- und Sicherheitsinfrastruktur, die der Vernetzung von Leistungserbringern, Kostenträgern, Versicherten und weiteren Akteuren des Gesundheitswesens sowie der Rehabilitation und der Pflege dient [...]“ [28]. Eine Auflistung weiterer Aufgaben findet sich in §311 [29].

Gleichzeitig ist die Telematikinfrastruktur auch als Plattform zu verstehen, auf der verschiedene Anwendungen laufen [23]. Ein anschauliches Beispiel ist die elektronische Patientenakte (kurz: ePA). In dieser sind wichtige Informationen zur Gesundheit von Patienten für das Fachpersonal gebündelt gespeichert [16]. Bei vielen der Anwendungen spielt die elektronische Gesundheitskarte [15] der Versicherten eine Rolle, z. B. als Datenspeicher für Notfalldaten im Rahmen des Notfalldatenmanagements (kurz: NFDM) [21].

Um an die Telematikinfrastruktur angebunden zu sein, müssen Arztpraxen, Krankenhäuser, etc. über einen sogenannten Konnektor verfügen [14]. Dieses Stück Hardware stellt eine sichere Verbindung zur zentralen Telematikinfrastruktur her. Gleichzeitig bündelt der Konnektor Schnittstellen, die von sogenannten Primärsystemen, also Client-Systemen wie z. B. Praxisverwaltungssystemen [22], genutzt werden können [19]. Ein Teil dieser Schnittstellen ist relevant, um Informationen von den mit dem Konnektor lokal verbundenen Kartenterminals weiterzuleiten [19, 14]. Mit den Terminals werden Daten aus Karten der Versicherten abgelesen, aber auch die Authentifizierung der Ärzte und Ärztinnen erfolgt mit speziellen Smartcards an den Terminals [17], [14].

Eine ausführlichere Beschreibung der Telematikinfrastruktur, mancher ihrer Anwendungen, des Konnektors und damit verbundenen Konzepten folgt im Grundlagenteil dieser Arbeit (Abschnitt 2). Auch relevante Technologien werden dort erklärt.

Ziel dieser Arbeit ist, die für Primärsysteme bereitgestellten Schnittstellen des Konnektors hinsichtlich ihrer Benutzbarkeit aus Sicht der Anwendungsprogrammierung zu bewerten.

Zu diesem Zwecke wurden zwei Methoden zur Untersuchung von Benutzbarkeit nach [35] angewandt. Zunächst habe ich selber einen Teil der Schnittstellen angesprochen im Rahmen des Programmierens einer einfachen CLI-Anwendung (Abschnitt 3.1). Als Referenzsystem wurde kein echter Konnektor genutzt, sondern ein Software-Simulator. Die Erkenntnisse aus diesem Teil der Arbeit wurden in einer sogenannten heuristischen Evaluation [35, Kap. 5.11] ausgewertet (Abschnitt 3.3). Außerdem wur-

den mit weiteren Programmierern Benutzbarkeitstests basierend auf zuvor definierten Aufgaben durchgeführt (Abschnitt 3.2). Eine Einführung in die Methoden und in Benutzbarkeit allgemein geschieht in Abschnitt 2.4.

Warum ist Benutzbarkeit wichtig? Zunächst führt gute Benutzbarkeit eines Systems zu Einsparungen hinsichtlich Zeit und Geld für die Organisationen, in denen diese Systeme genutzt werden [35, Kap. 1.1]. Stylos und Myers [34] beschreiben außerdem, dass sich schlechte Benutzbarkeit von Anwendungsschnittstellen negativ auf den Schutz des programmierten Systems auswirken kann und beziehen sich dabei auf [7].

Im Hinblick auf die Schnittstellen des Konnektors sind die Organisationen, in denen Zeit gespart werden kann, die Hersteller von Primärsystemen, also ebenjenen Systemen, die in den Praxen und Krankenhäusern auf die Gesundheitsdaten zugreifen und diese verarbeiten [22]. Damit die Digitalisierung des deutschen Gesundheitswesens erfolgreich und zügig abläuft, ist die Wirtschaftlichkeit bei der Entwicklung und Anpassung der Software an die Schnittstellen wichtig und somit auch die Benutzbarkeit jener.

Auch Aspekte der Sicherheit und des Schutzes sind vor dem Hintergrund der Sensibilität von Patientendaten und der Notwendigkeit eines ständig funktionierenden Gesundheitswesens relevant.

Nicht behandelt werden Prozesse innerhalb der gematik oder hinsichtlich des Ausrollens der Telematikinfrastruktur insgesamt. Auch wenn dies sicher das Zustandekommen guter bzw. schlechter Schnittstellen erklärt und Verbesserungspotentiale aufdecken kann, würde es den Rahmen dieser Arbeit übersteigen.

1.1 Benutzbarkeit von Programmierschnittstellen

Zum Themengebiet der Benutzbarkeit von Programmierschnittstellen wird sich im Verlaufe der Arbeit auf den Artikel von Stylos und Myers bezogen [34]. In diesem wird der bisherige Stand der Forschung in diesem Gebiet vorgestellt. Insbesondere übertragen die Autoren Erkenntnisse über die Benutzbarkeit von Computersystemen im Allgemeinen aus dem Bereich der Mensch-Computer-Interaktion auf den spezielleren Fall der Programmierschnittstelle. Benutzbarkeit als Qualitätsmerkmal von Programmierschnittstellen sollte nach ihnen ebenso beachtet werden, wie beispielsweise logische Korrektheit.

Zu Benutzbarkeit im Allgemeinen und Untersuchungsmethoden dieser wird sich in dieser Arbeit auf das Buch von Nielsen [35] bezogen, auf das auch Myers und Stylos in ihrer Arbeit referenzieren [34].

2 Grundlagen

2.1 Technische Grundlagen

In diesem Abschnitt sollen die Technologien erklärt werden, die es für die Arbeit mit dem Konnektor und dem Verständnis der anschließenden Abschnitte zu verstehen

gilt.

2.1.1 XML und XML Namensräume

Grundkenntnisse in XML werden für diese Arbeit als gegeben angenommen.

Viele der hiernach aufgezählten XML-basierten Technologien funktionieren über das Einführen weiterer XML-Elemente mit einem für die Technologie entsprechenden Namensraum (vgl. zum Beispiel [5, Abschn. 3]).

Auch XML-Namensräume sollen hier nicht detailliert erklärt werden. Sie funktionieren sehr ähnlich wie in gängigen Programmiersprachen und erklären sich beim Lesen der XML-Dokumente meist ohne weitere Hilfe aus dem Kontext. Eine genaue Spezifikation findet sich in [4].

2.1.2 SOAP

Für diese Arbeit ist die ältere Version 1.1 von SOAP relevant. In dem definierenden W3C Dokument wird SOAP als „Simple Object Access Protocol“ definiert [5].

Bei SOAP handelt es sich um ein XML-basiertes Protokoll für die Kommunikation zwischen Knoten in einem verteilten und dezentralen Netz [5, Abschn. 1]. Nachrichten in SOAP folgen einer bestimmten, typisierten Struktur [5, Abschn. 1]. Es soll schlicht sein und überlässt die Hoheit über die Bedeutung der Inhalte der Nachrichten den nutzenden Systemen und Anwendungen [5, Abschn. 1], [32, Abschn. 1.1, 2.1].

Als unterliegendes Transportprotokoll ist für diese Arbeit nur HTTP relevant. Es könnten aber auch andere Protokolle genutzt werden [5, Abstract].

An sich wird eine Nachricht in nur eine Richtung geschickt und zwar von einem Sender zu einem Empfänger [5, Abschn. 2]. Auf diesem Weg können aber auch weitere Knoten liegen, die möglicherweise auf die Nachricht reagieren [5, Abschn. 2]. Auf Anwendungsebene können SOAP-Nachrichten natürlich in kompliziertere Kommunikationsmuster eingebettet werden. Das wird in den entsprechenden spezifizierenden W3C-Dokumenten auch als häufige Anwendung erwähnt [5, Abschn. 2], [32, Abschn. 1.1].

Auf oberster Ebene des XML-Dokuments, dass eine SOAP-Nachricht enthält, ist das *Envelope*-Element. Innerhalb dieses Umschlags befinden sich zwei weitere Elemente [5, Abschn. 4]. Das erste ist der optionale *Header*, der Zusatzinformationen für die Art der Verarbeitung der Nachricht enthält, zum Beispiel mit Bezug auf die Rolle bestimmter Intermediärknoten auf dem Weg der Nachricht [5, Abschn. 4.2.2]. Das zweite Element, der *Body* muss immer vorhanden sein. Es enthält die Hauptinformation für den letztendlichen Empfänger der Nachricht [5, Abschn. 4]. Innerhalb des *Bodies* können auch Informationen zu Fehlern in einem *Fault*-Element benannt werden [5, Abschn. 4.4]. Diese Angaben müssen mindestens einen Fehler-Code, das *faultcode*-Element und einen deskripitiven Text, das *faultstring*-Element, enthalten [5, Abschn. 4.4].

Im Anhang finden sich Beispiele für SOAP-Nachrichten aus der Kommunikation zwischen Konnektor(-simulator) und CLI-Anwendung.

2.1.3 XML Schema

Für die im nächsten Abschnitt beschriebene Web Service Description Language wird zur Beschreibung genutzter Datentypen standardmäßig die Sprache *XML Schema* genutzt [1, Abschn. 1].

Mit dieser Sprache lassen sich Klassen von XML-Dokumenten definieren. Es handelt sich also um eine Sprache zur Erstellung von Schemata. Die Sprache basiert selbst wiederum auf XML [2, Abstract, Abschn. 1.1].

2.1.4 Web Services Description Language 1.1

Auch hier ist nicht die aktuellste Version [3] relevant, sondern die Version 1.1.

Bei WSDL-Dateien handelt es sich um eine Art von XML-Dateien zur Beschreibung von Web-Services [1, Abschn. 1]. Ein Web Service ist in diesem Kontext eine Menge von Endpunkten, mit denen über das Netz kommuniziert werden kann [1, Abschn. 1].

Um einen solchen Service zu beschreiben, werden u. a. folgende Elemente genutzt:

Als erstes können innerhalb des *types*-Elements Datentypen definiert werden, die für die kommunizierten Nachrichten wichtig sind [1, Abschn. 2.2]. Zur Beschreibung der Typen wird standardmäßig wie zuvor erwähnt XML Schema genutzt, es können aber auch andere Typsysteme hinzugefügt werden [1, Abschn. 2.2]. Typen können ebenfalls aus bestehenden Schema-Dateien importiert werden [1, Abschn. 2.1.2].

message-Elemente beschreiben die Informationen, die gesendet und empfangen werden [1, Abschn. 2.1]. Eine Nachricht enthält *parts*-Elemente als Kinder, die die Information in typisierte Teile zerlegen [1, Abschn. 2.3]. Die Art der Beschreibung ist abstrakt: Nur die logische Struktur der Nachrichten wird durch die *message*-Elemente beschrieben. Wie genau das Format einer übertragenen Nachricht aussieht, wird in weiter unten erklärten Elementen definiert [1, Abschn. 2.3.2].

Mehrere *message*-Elemente können als Attribute an die Ein- und Ausgabe-Elemente, also die Kinder von *operation*-Elementen gebunden werden [1, Abschn. 2.4]. In welcher Anzahl und Anordnung Ein- und Ausgabe-Elemente in einem *operation*-Element auftreten, hängt davon ab, was für ein Kommunikationsmuster dieser Operation zugrunde liegt. Besteht die Operation zum Beispiel aus nur einer Nachricht eines Senders oder handelt es sich um eine Abfolge aus Anfrage und Antwort [1, Abschn. 2.4]? Auch hier ist die Definition noch abstrakt: Es ist in den *operation*-Elementen noch nicht definiert, über wieviele Verbindungen, Pakete, etc. die einzelnen (ebenfalls noch abstrakt beschriebenen) Nachrichten versendet werden [1, Abschn. 2.4].

Die *operation*-Elemente tauchen als Kinder eines *portType*-Elements auf. Ein *portType* ist demnach eine Menge von Operationen [1, Abschn. 2.4].

Die Konkretisierung eines *portTypes* erfolgt in einem zugehörigen *binding*-Element [1, Abschn. 2.5]. Innerhalb jenes werden den Operationen und Nachrichten des *portTypes* konkrete Strukturen und Protokolle zugeordnet [1, Abschn. 2.5]. Wie genau das geschieht, ist abhängig von den genutzten Erweiterungen - auf Englisch „binding extensions“ [1, Abschn. 2.5]. Solche Erweiterungen gibt es zum Beispiel für das oben beschriebene SOAP 1.1 [1, Abschn. 1, 3]. Für einen abstrakten *portType* kann es mehrere *binding*-Elemente geben [1, Abschn. 2.5].

Die *binding*-Elemente werden als Attribute eines *port*-Elements um eine Adresse ergänzt, über die die nun konkreten Operationen angesprochen werden können [1, Abschn. 2.6]. Mehrere solcher Endpunkte sind letztendlich in einem *service*-Element gebündelt [1, Abschn. 2.7].

2.1.5 Jakarta XML Web Services

Die XML-basierte Spezifikation von Web-Services in Form von WSDL-Dateien ermöglicht es, automatisiert Code zu generieren, der den Web-Service darstellt.

Ein Java-Framework, das sowas ermöglicht, ist beispielsweise die Implementierung der Jakarta XML Web Service Spezifikation durch die Eclipse Foundation und die zugehörigen Tools und Plugins [27]. Die generierten Java-Klassen erlauben die Entwicklung von Klienten mit praktischem IDE-Support und ohne XML explizit prozessieren zu müssen.

2.2 Telematikinfrastruktur und Konnektor

In diesem Abschnitt sollen relevante Begriffe aus dem Gebiet der Telematikinfrastruktur und ihre Zusammenhänge erklärt werden.

Einen guten Einstieg bietet der *Implementeriungsleitfaden Primärsysteme - Telematikinfrastruktur* [10]. Dieses informative Dokument hilft bei der Einordnung der Konzepte aus den schwerer zu lesenden normativen Dokumenten.

2.2.1 Einordnung des Konnektors in die Telematikinfrastruktur

Die Umgebung, in der der Konnektor steht, sind die Einrichtungen der sogenannten *Leistungserbringer*, also die Räume in den Praxen und Kliniken. [13, S. 23], [10, Abschn. 2].

Die Primärsysteme kommunizieren nicht direkt mit den Kartenterminals. Nur der Konnektor kann mit diesen interagieren. Über das lokale Netzwerk können die Primärsysteme bei den Leistungserbringern auf die vom Konnektor bereitgestellten Schnittstellen zugreifen, um mit den Karten in den Kartenterminals zu arbeiten. Diese Terminals sind wiederum über das lokale Netz mit dem Konnektor verbunden. [10, Abschn. 2, 3.2.1].

Es gibt noch weitere mögliche Konstellationen, zum Beispiel die Nutzung von zwei Konnektoren. In dieser Variante ist ein Konnektor für die Anbindung der Praxissysteme verantwortlich, ein anderer für die Verbindung zur zentralen Telematikinfra-

struktur [10, Abschn. 3.2.2]. Wir gehen in dieser Arbeit allerdings von der einfachen Konstellation, dem *Online-Szenario* [10, Abschn. 3.2] aus.

Über den Konnektor erhalten die Primärsysteme Zugriff auf die Fachanwendungen, wie zum Beispiel das Versichertenstammdatenmanagement [10, Abschn. 2]. Der Konnektor stellt im Vokabular der gematik das dezentrale Glied einer Fachanwendung, ein *Fachmodul* zur Verfügung [13, S. 8, 15], wie zum Beispiel das Fachmodul VSDM [13, S. 40 f.], [10, Abschn. 2]. Die genauen Funktionen des Versichertenstammdatenmanagements (kurz: VSDM) [13, S. 40 f.] werden weiter unten beschrieben.

In den zentralen Teilen der Telematikinfrastruktur, zu der der Konnektor die Verbindung herstellt, befinden sich u. a. die zentralen Anteile der Fachanwendungen, die Fachdienste [13, S. 15], [10, Abschn. 2]. Außerdem sind dort auch nicht fachanwendungspezifische Dienste wie zum Beispiel der Namensdienst zu finden [13, S. 24], [10, Abschn. 2].

Abbildung 1 illustriert die in diesem Abschnitt beschriebenen Zusammenhänge.

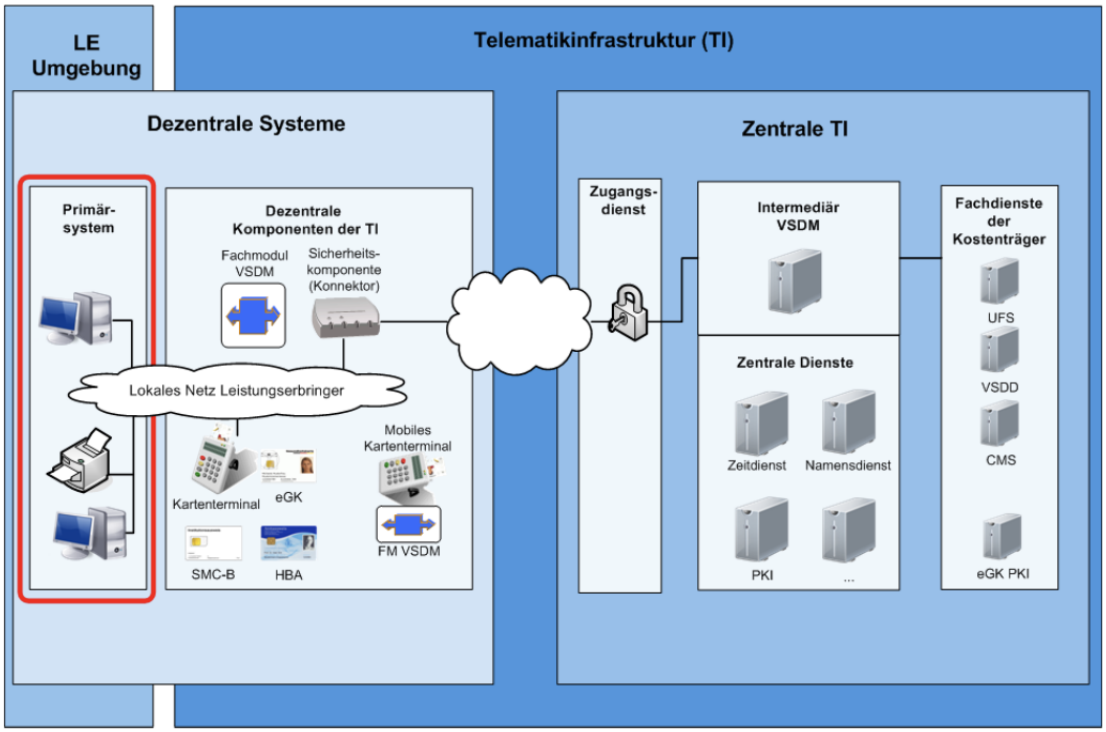

Abbildung 1: Überblick über den Konnektor im Kontext der Telematikinfrastruktur [10, S. 10]

2.2.2 Technische Aspekte der Kommunikation mit dem Konnektor

Die meisten der Konnektorschnittstellen werden über SOAP angesprochen [10, Abschn. 2, 3.2.1]. Das darunterliegende Transportprotokoll ist dabei HTTP, verschlüsselt oder unverschlüsselt und mit der Möglichkeit verschiedener Authentifizierungsme-

thoden zum Verbindungsaufbau [10, Abschn. 4.1]. Für die Betrachtung der fachlichen Schnittstellen ist das genaue Transportprotokoll nicht weiter relevant.

Die vom Konnektor angebotenen SOAP-Dienste werden über WSDL-Dateien von der gematik bereitgestellt [8, S. 45]. Mehr Information zu den genutzten Versionen dieser Dateien, der Dokumente der gematik insgesamt und eine Angabe zum Herunterladen der verschiedenen Materialien finden sich im Anhang. Im eigentlichen Fließtext werden die Versionen nicht näher angegeben, sondern nurnoch die Namen der Dateien, da sich stets auf die im Anhang erwähnten Versionen bezogen wird.

Zwei wichtige Ausnahmen von SOAP als Protokoll auf Anwendungsebene sind der Zugriff auf das Konnektordienstverzeichnis und den Ereignisdienst [10, Abschn. 2, 3.2.1].

Das Dienstverzeichnis ist über eine spezifische URL, die sich in der Dokumentiation des Herstellers findet, per HTTP(S) zu erreichen [10, Abschn. 4.1.1, 4.1.2]. In diesem Verzeichnis sind die angebotenen Endpunkte/Dienste samt Adresse und Version aufgelistet [10, Abschn. 4.1.2].

Der Ereignisdienst teilt über ein für ihn spezifisches Protokoll Informationen über Geschehnisse am Konnektor per Push an die Primärsysteme mit [10, Abschn. 4.1.4]. Er wird im Rest der Arbeit nicht näher betrachtet, da alle wichtigen Informationen auch über Pull-Mechanismen an den SOAP-Schnittstellen realisiert werden können. Beispielsweise lässt sich die Information über eine neu gesteckte Karte auch über ein Aufrufen von getCards (siehe unten) erhalten.

2.2.3 Kartentypen

Die vier für den Rest der Arbeit relevanten Kartentypen sollen hier erläutert werden.

Die erste ist die elektronische Gesundheitskarte (kurz: eGK) der gesetzlich Versicherten, die bereits in der Einleitung erwähnt wurde.

Für Ärzte und andere Heilberufler exisiert der personenbezogene Heilberufsausweis (kurz: HBA) [13, S. 18], mit dem sich diese identifizieren können.

Damit sich zum Beispiel ganze Praxen in der Telematikinfrastruktur identifizieren können, existiert die *Security Module Card Typ B* (kurz: SMC-B) [13, S. 31].

Zuletzt soll die gSMC-KT [13, S. 31] beschrieben werden. Diese Karte speichert die Schlüssel zur Identifikation eines Kartenterminals und wird dort gesteckt [11, S. 10, 36].

2.2.4 Aufrufkontext

Für die Kommunikation zwischen Konnektor und Primärsystem ist die Anordnung verschiedener Entitäten zueinander von Bedeutung.

Ein *Mandant* ist ein Teil einer Organisation mit eigenem Datenhaushalt innerhalb des Primärsystems [10, S. 12].

Bei einem *Arbeitsplatz* handelt es sich in diesem Kontext um einen gängigen Computer-Arbeitsplatz samt Kartenterminal [10, S. 12].

Verschiedene bei den Leistungserbringern vorhandene Primärsysteme, Terminals, Mandanten und Arbeitsplätze können in verschiedenen Konstellationen zueinander stehen.

Beispielsweise kann ein Kartenterminal mehreren Arbeitsplätzen zugeordnet sein [10, S. 152]. Ebenso können an einem mandantenfähigen [13, S. 24] Primärsystem an einem festen Arbeitsplatz mehrere Mandanten arbeiten. Alle möglichen Konstellationen samt Szenarien aus dem medizinischen Alltag, in dem diese auftreten könnten, finden sich in [10, Abschn. 9.1.2].

Den Anfragen des Primärsystems an den Konnektor sind Informationen über obige Einheiten als Context-Objekt mitzugeben [10, S. 17]. In diesem sind verpflichtend die Felder *MandantId*, *ClientSystemId* (für das Primärsystem) und *WorkplaceId* [10, S. 17].

Als optionales Feld für Benutzung des personenbezogenen HBA muss auch noch eine *UserId* angegeben werden [10, S. 17 f.]. Diese durch das Primärsystem erstellte ID gibt für einen HBA einen Nutzer des Primärsystems an [10, S. 18]. Die Zuordnung erfolgt bei der PIN-Eingabe, die den HBA freischaltet [10, S. 18]. Ein freigeschalteter HBA ist für manche Funktionen notwendig, zum Beispiel zum Signieren von Dokumenten [10, S. 43 f.]. Nur der entsprechende Nutzer kann für weitere Operationen diesen HBA verwenden [10, S. 18].

Die Angabe der existierenden Kontexte und Verwaltung der Zugriffsberechtigungen dieser muss durch einen Administrator am Konnektor geschehen. Dabei muss die Konfiguration auf die Einstellungen am jeweiligen Primärsystem abgestimmt werden, die Modelle müssen sich gleichen [10, S. 16 f.].

2.2.5 Dienste und Anwendungen

Um gesteckte Karten für weitere Anwendungen nutzen zu können, benötigen Primärsysteme für diese einen Karten-Handle [10, Abschn. 4.2.1]. Um an einen Handle zu gelangen, gibt es verschiedene Möglichkeiten:

Zunächst können mittels der Operation *getCards* für den angegebenen Aufrufkontext zugreifbare Karten aufgelistet werden [10, S. 45]. Neben dem Aufrufkontext können als optionale Parameter zur Einschränkung der Ergebnisse *CardType*, *CtId* (Id des Kartenterminals) und *SlotId* (Nummer des Kartensteckplatzes im Terminal) mitgegeben werden [10, S. 45 f.]. Die Angabe einer *CtId* kann bei mehreren Terminals an einem Arbeitsplatz sinnvoll sein.

Einem Mandanten können mehrere Arbeitsplätze zugeordnet sein [10, S. 13]. Mit dem booleschen Parameter *mandant-wide* lassen sich für einen Mandant und ein Primärystem die Karten an allen zugehörigen Arbeitsplätzen in allen verbundenen Kartenterminals anzeigen [10, S. 48].

An die Information über die vorhandenen Kartenterminals und deren Steckplätze kann das Primärsystem über *getCardTerminals* gelangen. Die Aufrufparameter sind

fast identisch zu denen von getCards, es muss auch ein Aufrufkontext angegeben werden [8, S. 183 f.].

Die Operationen getCards und getCardTerminals sind Teil des Systeminformationsdiensts (EventService) [10, S. 45, 49]. Auch der *CardTerminalService* bietet die Möglichkeit, Karten-Handles steckplatzbezogen zu erlangen. Dabei kann sogar über ein festgelegtes Zeitintervall auf noch zu steckende Karten gewartet werden und zwar über die Operation requestCard [10, S. 49].

Wie zuvor bereits erwähnt, können Karten wie der HBA freigeschaltet werden [10, S. 43]. Die dafür notwendige Funktionalität stellt der *CardService* zur Verfügung [8, S. 158 f.]. Zwei Operationen sind für diese Arbeit relevant: Mit *getPinStatus* kann unter anderem unterschieden werden, ob eine Karte bereits freigeschaltet ist, oder nicht [10, S. 43]. Mit *verifyPin* wird der Prozess der PIN-Eingabe am Kartenterminal eingeleitet [8, S. 159 f.]. Allgemein erfolgen alle PIN-Eingaben stets an den Kartenterminals selber [10, S. 42].

Als Parameter erfordern die beiden Operationen neben einem Aufrufkontext den Karten-Handle der freizuschaltenden Karte. Darüber hinaus ist der *PinTyp* anzugeben [8, S. 159, 164]. Dieser ist Abhängig vom Kartentyp, beispielsweise muss für das Freischalten einer SMC-B eine *PIN.SMC* eingegeben werden [10, S. 41, 43 f.].

Zuletzt soll in diesem Abschnitt auf die Funktionalitäten eines Fachmoduls eingengagen werden. Das betrachtete Fachmodul ist das Fachmodul VSDM, die Funktionalitäten sind gebündelt im *VSDService* [10, S. 55].

Bei den Versichertenstammdaten handelt es sich um die grundlegenden Daten eines Versicherten und seiner Versicherung. Dazu gehören zum Beispiel Name und Adresse oder Angaben zur Gültigkeit des Versicherungsschutzes [26], [10, S. 72].

Die Stammdaten liegen auf der eGK und werden durch die jeweilige Krankenversicherung verwaltet [26]. Die Daten auf der Karte können über die Telematikinfrastruktur bei Änderung aktualisiert werden. Dieser Prozess kann durch den Konnektor beim Einführen der eGK in ein angeschlossenes Terminal angestoßen werden [26]. So können Versicherte ihre eGK zum Beispiel auch nach einem Umzug weiterbenutzen [26].

Die Schnittstelle am Konnektor zum Lesen der Stammdaten durch das Primärsystem wird durch die Operation *readVSD* des *VSDService* gebildet [10, S. 55].

Die beiden wichtigsten Eingabeparameter sind dabei ein Handle der zu lesenden eGK und ein Handle für den HBA oder die SMC-B, also die Karte, die zum Zugriff auf die Stammdaten berechtigt [9, S. 13 ff.]. Diese Karte muss freigeschaltet sein [9, S. 24], [9, S. 24]. Außerdem muss ein Aufrufkontext mitgegeben werden, bei Nutzung eines HBA auch mit einer UserId [9, S. 13]. Ob eine Aktualisierung der Daten auf Grundlage der Daten in der zentralen Telematikinfrastruktur geschehen soll, kann durch entsprechende boolesche Parameter ebenfalls kodiert werden [9, S. 13]. Dieser Aspekt wurde in dieser Arbeit jedoch nicht weiter betrachtet.

Das VSDM ist die einzige Fachanwendung, die in dieser Arbeit aus Zeitgründen näher betrachtet wurde. Andere Anwendungen wie die ePA sind daher an dieser Stelle nicht

ausführlicher beschrieben.

Im nächsten Abschnitt ist unser Test-System näher beschrieben.

2.3 Konnektorsimulator für Primärsysteme (KoPS)

Damit Primärsystemhersteller ihre Anwendungen gegen die Vorgaben der gematik testen können, stellt diese auf ihrer Website den Konnektorsimulator für Primärsysteme (kurz: KoPS) zur Verfügung [20], [6]. Mit diesem lässt sich die Anbindung an die gegebenen Schnittstellen genau nach den Spezifikationen der gematik testen [20]. Dadurch entfällt die Anschaffung eines echten Hardware-Konnektors; auch Karten und Kartenterminals können simuliert werden [20].

Der Konnektorsimulator lässt sich unter Vorgabe einer Adresse und eines Ports starten, unter der eine Web-Oberfläche zu seiner Steuerung erreichbar ist. In dieser lassen sich zum Beispiel vorkonfigurierte Karten in Kartenterminals stecken. Für die Karten kann keine PIN eingegeben werden. Dies wird durch den Simulator abstahiert: Für manche der vorgefertigten Karten schlägt *verifyPin* fehl, für manche funktioniert es. In der Weboberfläche lassen sich außerdem in einem Log die konkreten SOAP-Anfragen und Antworten ausgeben [6]. Ein Beispiel für eine solche Anfrage findet sich im Anhang.

2.4 Benutzbarkeit

Die Begriffe *Irrtum*, *Fehler*, *Defekt* und *Versagen* sollen nach der Definition aus der einführenden Vorlesung in die Softwaretechnik am Institut für Informatik der FU Berlin verstanden werden [37]. Werden die Begriffe in bestimmten Zusammenhängen anders genutzt, so ist dies vermerkt.

Nielsen beschreibt den Begriff der Benutzbarkeit eines Systems, das von Menschen benutzt wird, als Eigenschaft, die durch folgende Merkmale bestimmt ist [35, Kap. 2.2]:

- Erlernbarkeit
- Effizienz: Potential des Systems, von erfahrenen Nutzern sehr effektiv genutzt werden zu können.
- Einprägsamkeit bzw. Merkbarkeit
- Seltenes Auftreten von Fehlern durch den Nutzer: Für Nielsen ist ein Fehler dabei eine nicht zum beabsichtigten Ergebnis führende Handlung [35, S. 32].
  - Insbesondere schwerwiegende Fehler, die zu Defekten führen, sollten nicht auftreten.
  - Die Möglichkeit der Korrektur des nicht beabsichtigten Ergebnis bzw. der Wiederherstellung eines gewünschten Zustands sollte bestehen.
- Eine angeneheme Nutzung: Die Nutzer sollten gerne mit dem System interagieren.

Welche konkreten Elemente eines Systems zu guter Benutzbarkeit führen, ist stark vom Einzelfall abhängig und schwer vorauszusagen [35, S. 16]. Bekannt hingegen ist der *Prozess*, der zu guter Benutzbarkeit führt [35, S. 16] und sehr detailiert im Buch von Nielsen beschrieben wird [35]. Darin werden verschiedene Techniken vorgestellt [35, S. 224]. Auf zwei dieser Techniken zur Anwendung auf ein bereits bestehendes System wird in den nächsten Abschnitten eingegangen.

2.4.1 Heuristische Evaluation

Bei der heuristischen Evaluation handelt es sich um eine Technik zur Aufdeckung von Benutzbarkeitsproblemen [35, S. 155]. Evaluiert wird das System dabei von einer oder mehreren Personen jeweils alleine anhand einer Liste von allgemein akzeptierten Heuristiken zu guter Benutzbarkeit [35, S. 155 ff., 19 f.]. Das Ergebnis der Evaluation ist eine Sammlung von Problemen mit Angaben der Heuristiken, gegen die jeweils verstoßen wird. Mithilfe dieser Liste können die Probleme unter Beachtung der Heuristiken im Anschluss behoben werden [35, S. 159].

Die Heuristiken, die in dieser Arbeit genutzt werden, sind die zehn von Nielsen vorgeschlagenen Prinzipien in der überarbeiteten Form von 2020 [36]. Myers und Stylos haben diese aufgegriffen und auf den speziellen Fall von der Benutzbarkeit von APIs übertragen, d. h. für jede der Heuristiken eine Anwendung auf APIs erklärt [34]. Die folgende Aufzählung enthält sowohl die Heuristiken nach [36] als auch die Anpassungen für APIs nach [34]. Die Reihenfolge der Grundsätze ist dabei beibehalten.

1. Schneller und kontinuierlicher Zugang zu Informationen über den Zustand des Systems

   Anpassung: Als Beispiel führen Myers und Stylos eine API an, die mit Dateien arbeitet. Es sollte dabei ersichtlich sein, ob die Datei für das Beschreiben bereit ist und wenn nicht beim Versuch eine Fehlermeldung geben.

2. Anpassung des Systems und zugehöriger Begriffe an die natürliche Umgebung und die Nutzer, für die es vorgesehen ist:

   Anpassung: Für APIs sollten die Namen der Methoden und Klassen zum mentalen Modell der Nutzer passen.

3. Bereitstellung von Wegen um ungewollt ins Rollen gebrachte Prozesse abzubrechen und in den Ausganszustand zurückzukommen. Als eigenes Beispiel führe ich hier den Abbruch von einem Programm an der Mikrowelle an.

   Anpassung: Über solche Operationen zur Wiederherstellung und vorzeitigen Beendigung sollte eine API nach Myers und Stylos verfügen.

4. Konsistenz, sowohl innerhalb der API als auch im Bezug auf außerhalb geltende Standards und Konventionen

   Anpassung: Ein Beispiel aus [34] ist, dass die Reihenfolge von Parametern in überladenen Methoden gleich bleiben sollte.

5. Verhinderung von Fehlern bei Nutzung, unter anderem dadurch, dass den Nutzern sinnvolle Grenzen in der Benutzung des Systems gesetzt werden, um Versehen entgegenzuwirken. Bei konfigurierbaren Systemen sollten aus dem gleichen Grund sinnvolle Standardkonfigurationen gegeben sein. Um schwerere Fehler aufgrund falschen Versändnisses zu vermeiden, wird empfohlen, dass Nutzer sich nicht zuviel merken müssen sollten, die Möglichkeit des Rückgängigmachens haben und gewarnt werden bzw. Handlungen nochmals bestätigen müssen.

   Anpassung: Myers und Stylos betonen für diese Heuristik nochmals die Wichtigkeit von korrekt funktionierenden Standardkonfigurationen.

6. Benötigte Information zur Nutzung eines Teils des Systems sollte an den richtigen Stellen (wieder-)erkennbar platziert werden, sodass die Nutzer sich nicht alles merken müssen. Die Nutzer sollten möglichst wenig im Gedächnis behalten müssen, um das System zu benutzen.

   Anpassung: Aussagekräftige und unterscheidbare Bezeichner sind wichtig, um zum Beispiel sinnvolle Hilfe an der richtigen Stelle durch die automatische Vervollständigung einer IDE zu erhalten.

7. Durch Bereitstellung vielfältiger Weisen der Nutzung sollte das System sowohl auf Anfänger abgestimmt sein, als auch Experten in die Lage versetzen, durch individuell anpassbare Abläufe und Abkürzungen das System zügiger nutzen zu können.

   Anpassung: In diesem Punkt fügen Myers und Stylos nichts hinzu.

8. (Visueller) Fokus auf relevante Informationen und Teile des Systems. Unwichtige Information sollte nicht Teil der Benutzeroberfläche sein.

   Anpassung: Es wird in [34] argumentiert, dass diese Heuristik nicht unbedingt eine kleinere und kompaktere API impliziert. Auch größere APIs können zum Beispiel durch Präfixe in den Bezeichnern zur Unterscheidung der verschiedenen Teile gut benutzbar sein, Myers und Stylos verweisen dabei auf weitere Literatur [39] (zitiert nach [34]).

9. Folgen von Fehlern sollten erkennbar und nachvollziehbar sein. Außerdem sollten Lösungsmöglichkeiten zur Behebung gegeben werden.

   Anpassung: Welche Art von Fehlermeldungen am nützlichsten sind, ist bisher wenig belegt.

10. Gutes Hilfsmaterial: Sollte das System zusätzliches Informationsmaterial benötigen, sollte dieses gut durchsuchbar sein und klare Handlungsanweisungen geben, mit denen Nutzer ihre Ziele erreichen. Information sollte außerdem nach Möglichkeit nah am problematischen Teil des Systems auffindbar sein. Als Beispiel wird in Nielsens Liste ein Informationsschalter am Flughafen genannt, an den sich Reisende mit ihrem konkreten, orts- und zeitgebundenen Problem wenden können.

Anpassung: Hier bemerken Myers und Stylos unter Verweis auf weitere Literatur [38], dass Dokumentation ein häufiger Kritikpunkt von APIs ist.

2.4.2 Benutzbarkeitstests

Ein Benutzbarkeitstest ist das Beobachten von Teilnehmern beim Benutzen des Systems beim Lösen von Aufgaben [33].

Das Testen des Systems mit echten Nutzern ist nach Nielsen die wichtigste Technik für gute Benutzbarkeit [35, S. 165].

Durch Benutzbarkeitstests können sowohl qualitative wie auch quantitative Daten gesammelt werden. Quantitative Daten sind beispielsweise gemessene Zeiten während des Erfüllens der Aufgaben [35, S. 192]. Qualitative Daten sind zum Beispiel Erkenntnisse über Differenzen zwischen dem mentalen Modell der Nutzer und dem System [35, S. 195].

Die Teilnehmer des Tests sollten die tatsächlichen Nutzer des Systems so stark wie möglich abbilden [35, S. 175]. Dabei sollten sowohl solche mit mehr und weniger Erfahrung unter ihnen sein [35, S. 177]. Von Nielsen genannte Dimensionen von Erfahrung sind dabei Domänenwissen, Kenntnisse des Systems und allgemeine Erfahrung mit Computern [1] [35, S. 43 f.].

Die Person, die den Test durchführt, der Experimentator, sollte sich gut mit dem getesteten System auskennen, um Handlungen der Teilnehmenden richtig einordnen zu können [35, S. 180]. Ein Experte in Benutzbarkeit muss der Experimentator nicht unbedingt sein, um Benutzbarkeitstests gewinnbringend durchführen zu können [35, S. 180].

Es ist zu beachten, dass die Teilnehmenden Menschen mit Gefühlen sind. Aus diesem Grunde sollte klargestellt werden, dass das System evaluiert wird und nicht die Person [35, S. 182]. Außerdem sollten die Ergebnisse in einer Weise beschrieben werden, die keine Rückschlüsse auf die Identität der einzelnen Person erlaubt [35, S. 182]. Insbesondere die Vorgesetzten der Teilnehmer sollten keinen Einblick in den Testvorgang und die Ergebnisse des einzelnen bekommen [35, S. 184]. Über mögliche Aufnahmen des Geschehens ist vorher ebenfalls aufzuklären [35, S. 184]. Zur Schaffung einer guten Atmosphäre während und nach dem Test nennt Nielsen weitere Vorschläge in [35, S. 184].

Die Aufgaben sollten ebenso wie die Teilnehmenden möglichst den realen Bedingungen, in denen das System genutzt wird gleichen [35, S. 185]. Um Teilnehmenden kein Gefühl der Inkompetenz zu geben, wenn sie von mehreren Aufgaben nicht alle schaffen, empfiehlt Nielsen die Aufgaben einzeln vorzustellen, statt gesammelt zu Beginn. Ebenso sollte die erste Aufgabe für einen angenehmen Einstieg einfach sein [35, S. 187].

Nach einer kurzen Einführung in den Ablauf und Zweck des Tests lässt der Experimentator den Teilnehmenden größtenteils alleine arbeiten [35, S. 190]. Geholfen wer-

[1] Nielsen bezieht sich in seinem Buch [35] größtenteils auf Computersysteme.

den sollte nur bei Frustration, wenn die Teilnehmenden überhaupt keine Fortschritte machen [35, S. 190]. Viele wichtige Beobachtungen über die Benutzbarkeit des Systems können erst gemacht werden, wenn Nutzer ihre Probleme selbstständig lösen müssen [35, S. 183]. Nach den Aufgaben können die Teilnehmenden im Dialog Anmerkungen und Verbesserungsvorschläge machen [35, S. 191].

Die Teilnehmenden werden gebeten, während der Arbeit laut zu denken. Dadurch können Rückschlüsse auf ihr mentales Modell und ihre Trugschlüsse gezogen werden [35, S. 195].

Der Testplan, also die konkreten Entscheidungen hinsichtlich Experimentator, Aufgaben, Ort, Zeit, etc., sollte vorab vorbereitet und zu Papier gebracht werden [35, S. 170 f.].

2.4.3 Vor- und Nachteile der Methoden

Ein Vorteil der heuristischen Evaluation ist, dass sie durch eine Person anhand der entsprechenden Leitfäden alleine durchgeführt werden kann und keine zusätzlichen Nutzer benötigt werden [35, S. 155 f., 224 ff.]. Möglicherweise werden dabei aber Probleme übersehen, die den typischen Nutzern begegnen würden [35, S. 224].

Benutzbarkeitstests mit Verbalisierung der Gedanken der Teilnehmenden wie oben beschrieben eignen sich weniger zur Messung quantitativer Daten wie der Arbeitsgeschwindigkeit, da das kontinuierliche Reden während des Tests für viele Personen unnatürlich ist [35, S. 195 f.]. Im Gegenzug ist diese Methode sehr effektiv, um viele qualitative Daten zu erhalten, selbst bei einer kleinen Zahl von Teilnehmenden [35, S. 195].

Beide Methoden werden von Nielsen als günstige und pragmatische Techniken aufgezählt [35, S. 17 ff.], die sich gut ergänzen, da sich unter anderem die Mengen der aufgedeckten Benutzbarkeitsprobleme wenig überlappen [35, S. 225 f.].

3 Hauptteil

3.1 Eigene Anwendung

Der vollständige Code findet unter `https://git.imp.fu-berlin.de/emi41/thesis` mit dem Commit-Hash a4ffe46165e28aaff711e087efaf62515ae8a96f.

Die für diese Arbeit programmierte Anwendung bündelt fünf der durch den Konnektor und seine Dienste bereitgestellten Operationen als Kommandozeilenoperationen. Das sind die bereits in 2.2.5 beschriebenen Operationen getCards, getCardTerminals, verifyPin, requestCard und readVSD.

Die Optionen der einzelnen Operationen gleichen den zugehörigen Signaturen, es ist also nicht viel abstrahiert.

Abbildung 2 zeigt die Optionen beim Nutzen der Operation getCards. Neben den erlaubten Datentypen ist auch angegeben, was die jeweilige Option für die Operation bedeutet und ob die Angabe eines Werts Pflicht ist oder nicht.

```
Usage: testPrimaersystem getCards options_list
Options:
    --mandantId, -m -> Id des Mandanten (always required) { String }
    --clientId, -c -> Id des Klienten (always required) { String }
    --workplaceId, -w -> Id des Arbeitsplatzes (always required) { String }
    --userId, -u -> Id des Users { String }
    --cardType, -C -> Filter nach Kartentyp { Value should be one of [EGK, HBA-qSig, HBA, SMC-B, HSM-B, SMC-KT, KVK, ZOD_2.0, UNKNOWN, HBAx, SM-B] }
    --mandantWide, -mw [false] -> Ist diese Option gesetzt, werden Karten für alle Arbeitsplätze zurückgegeben, die dem aktuellen Mandanten und Klientensystem zugeordnet sind
    --cardTerminalId, -ctId -> Filter nach einem bestimmten Kartenterminal per entsprechender Id { String }
    --slotId, -sId -> Nummer des Slots des angegebenen Kartenterminals (gezählt wird ab 1). Ist kein Terminal angegeben, tritt ein Fehler auf { Int }
    --help, -h -> Usage info
```

Abbildung 2: Optionen zum Aufruf der Operation getCards

```
---- Karte 1 (Typ: EGK) ----
Card Handle: cc6a719e-8478-4d15-9476-aee92bd84c0f
Kartenterminal: T2 (Slot 1)
Kartenhalter: Martin Niño Gômez
---- Karte 2 (Typ: EGK) ----
Card Handle: 23bf5954-6c0e-4be3-8765-df47f69a051c
Kartenterminal: T3 (Slot 1)
Kartenhalter: Sieglinde Blücher
```

Abbildung 3: Rückgabe für getCards

Abbildung 3 zeigt die Antwort auf getCards gegen den Simulator. Dabei wurden neben einem Aufrufkontext noch die EGK als Kartentyp über die *cardType*-Option angegeben. Außerdem sind Karten aus verschiedenen Terminals in der Antwort enthalten, da auch die *mandantWide*-Option gesetzt wurde.

Die Struktur des Codes wird in den folgenden Abschnitten erklärt, zunächst sind die genutzten Werkzeuge beschrieben.

3.1.1 Bibliotheken und Werkzeuge

Die genutzte Programmiersprache ist Kotlin. Innerhalb von Kotlin lässt sich Java-Code aufrufen, somit auch die aus den WSDL-Dateien generierten Java-Klassen.

Ein weiterer Vorteil ist, dass mein Betreuungsumfeld in der Firma gute Kenntnisse in Kotlin hat und bei Bedarf Hinweise geben konnte. So auch die Idee für die Nutzung der nachfolgend beschriebenen Bibliothek:

Als Parser für die Argumente auf der Kommandozeile wird kotlinx-cli [31] genutzt. Damit lassen sich einfach und schnell Subkommandos mit jeweils eigenen Argumenten definieren und auch, ob diese Argumente notwendigerweise anzugeben sind. Ein Nachteil ist, dass die Bibliothek bis jetzt nicht in stabiler Version veröffentlicht ist und sich möglicherweise in ihrer Schnittstelle noch verändert.

Zur Verwaltung der Abhängigkeiten während des Bauens nutze ich Maven. Das aufgesetzte Maven-Projekt folgt dabei größtenteils der vorgeschlagenen Standardkonfiguration von IntelliJ, der verwendeten IDE.

Über Maven wird auch das zu den Jakarta XML Web Services gehörige Plugin konfiguriert. Hier müssen die zu übersetzenden WSDL-Dateien angegeben werden:

```
<wsdlDirectory>
    src/main/resources/META-INF/gematik_schema-f_r312_final
       /conn
</wsdlDirectory>
<wsdlFiles>
    <wsdlFile>EventService.wsdl</wsdlFile>
    <wsdlFile>vsds/VSDService.wsdl</wsdlFile>
    <wsdlFile>CardService.wsdl</wsdlFile>
    <wsdlFile>CardTerminalService.wsdl</wsdlFile>
</wsdlFiles>
```

Codebeispiel 1: Angabe der zu prozessierenden WSDL-Dateien durch das Jakarta XML Web Services Plugin in Maven

3.1.2 Aufbau des Codes

Die Struktur zum Ansprechen der verschiedenen Operationen ist stets gleich und soll im Folgenden am Beispiel von getCardTerminals beschrieben werden.

```
1  val context = ContextType()
2  context.mandantId = mandantId
3  context.clientSystemId = clientId
4  context.workplaceId = workplaceId
5  context.userId = userId
6
7  val query = GetCardTerminals()
8  query.context = context
9  query.isMandantWide = mandantWide
10
11 val port = EventService().eventServicePort
```

Codebeispiel 2: Setzen der Felder des Eingabeobjekts für die Operation getCardTerminals und des benötigten Endpunkts

Zunächst ist zu betonen, dass die Aufrufparameter mancher Operationen nicht einzeln, sondern in einem Objekt gebündelt übergeben werden. Für getCardTerminals ist das das GetCartTerminals-Objekt (vgl. Zeile 7 in Codebeispiel 2). Auf diesen Aspekt wird in der späteren Diskussion zu Benutzbarkeit weiter eingegangen.

Um die Operation nun mit dem vollständig befüllten Objekt aufzurufen, ist noch ein Endpunkt notwendig (vgl. Zeile 11 in Codebeispiel 2).

Auf diesem kann dann die Operation aufgerufen werden:

```
1 val result = port.getCardTerminals(query)
```

Codebeispiel 3: Aufruf der Operation getCardTerminals auf einem Endpunkt des EventService

Teile der zurückgegebenen Ergebnisobjekte werden dann in menschenlesbarer Form auf die Konsole gedruckt.

Eine technische Besonderheit ist, dass Operationen wie readVSD oder verifyPin in den generierten Java-Klassen den Rückgabetyp void haben. Die Ergebnisse werden in Holder-Objekte geschrieben, die als Eingabe in den Aufruf mitgegeben werden. Diese Objekte sind teil der Jakarta XML Web Services API. Warum manche der in den WSDL spezifizierten Operationen wie verifyPin auf diese Art in Code übersetzt werden, während beispielsweise das oben beschriebene getCardTerminals einen Rückgabewert hat, konnte ich nicht abschließend erklären.

3.1.3 Qualitätssicherung

Da der Zweck dieser Implementierung hauptsächlich die Illustration einiger Konzepte und das Sammeln von Erfahrung für die Bewertung der Benutzbarkeit ist, habe ich keine Tests geschrieben.

Eine rudimentäre Behandlung von Versagen des Systems ist durch try-catch-Blöcke um die Aufrufe der Konnektoroperationen implementiert. Für den bisherigen Betrieb während der eigenen Beobachtungen und der Benutzbarkeitstests genügte das.

3.1.4 Umfang und Reifegrad der Anwendung

Wie bereits erwähnt, wurde in dieser Arbeit eine Auswahl aus den Anwendungen und Diensten des Konnektors betrachtet, um einen Eindruck über die Schnittstellen zu erhalten. Nicht abgedeckt ist z. B. die elektronische Patientenakte.

Die programmierte Anwendung ist hinsichtlich der bereits beschriebenen Funktionalitäten noch weit entfernt von einem Primärsystem, das die Anforderungen der gematik erfüllt. Ein solches muss beispielsweise die Endpunkte aus dem Dienstverzeichnis lokal zwischenspeichern und bei Problemen der Erreichbarkeit neu abfragen und aktualisieren [10, S. 28]. Das implementierte System enthält hingegen fest kodiert die Standardadressen des Konnektorsimulators.

Allgemein hält die Anwendung zwischen den Operationen keinen Zustand.

3.1.5 Schwierigkeiten

Auf Herausforderungen hinsichtlich der Spezifikation der gematik bei der Entwicklung wird in der heuristischen Evaluation in einem der folgenden Abschnitte eingegangen.

Auf technischer Ebene hat die Einarbeitung in SOAP, WSDL, Maven und dem Java-Framework zu vielen Problemen geführt und mit Abstand den größten Anteil der Entwicklungsarbeit eingenommen. Diese Probleme sind an dieser Stelle nur anhand zwei kurzer Beispiele beschrieben, da sie höchstwahrscheinlich auf meine zuvor fehlende Kenntnis über diese Technologien zurückzuführen sind und nicht auf die logische Struktur und Konzepte der Telematikinfrastruktur.

Erstens war nicht klar, wie die Werkzeuge des Frameworks in Maven einzubinden und in welcher Form die zu übersetzenden WSDL-Dateien anzugeben sind.

Zweitens hat es längere Zeit gedauert, bis ich verstanden habe, wie die Adressen der Endpunkte für die Services in dem generierten Code auf die Adressen des Konnektorsimulators angepasst werden können.

3.2 Benutzbarkeitstests

3.2.1 Aufbau

Die leitende Fragestellung hinter den durchgeführten Benutzbarkeitstests war, wie gut mit einer kurzen Einführung in das Themengebiet und anhand schrittweise schwierigerer Aufgaben das Arbeiten mit den Konnektorschnittstellen funktioniert. Von den in 2.4 gennanten Aspekten von Benutzbarkeit wurden also hauptsächlich Erlernbarkeit, eine angenehme Nutzung und die Art und Häufigkeit von Fehlern untersucht. Effizienz und Einprägsamkeit wurden durch die Tests wenig abgedeckt, da die Teilnehmenden erstens allesamt Anfänger in der Arbeit mit dem Konnektor waren und zweitens pro Person nur eine Testsitzung absolviert wurde, sich also nichts über längere Zeit gemerkt werden musste.

Die Tests fanden am 17., 18. und 19.08.2022 in einem abgetrennten Raum in der Geschäftsstelle der akquinet tech@spree statt. Die Teilnehmenden saßen an einem Arbeitsplatz mit Laptop, zusätzlichem Bildschirm, Maus und Tastatur. Ich, der Experimentator, saß mit Abstand an einem Platz daneben vor einem Bildschirm, der das Geschehen auf dem Arbeitsbildschirm des Teilnehmers spiegelt. Dadurch konnte ich die Interaktionen beobachten, ohne den Teilnehmenden das Gefühl zu geben, direkt über die Schulter beobachtet zu werden.

Auf dem Laptop war IntelliJ als IDE bereits geöffnet mit einem vorkonfigurierten Kotlin/Maven-Projekt. Darin befanden sich bereits die notwendigen WSDL und XSD-Dateien. Um die Konfiguration und Eigenschaften des Plugins der Jakarta XML Web Services mussten sich die Teilnehmer nicht kümmern. Der Konnektorsimulator lief mit geöffneter Weboberfläche ebenfalls mit bekannter Adresse und Port.

Die Teilnehmenden waren zwei Kommilitonen und zwei Kollegen aus der Firma, allesamt mit mehrjährigen Progammierkenntnissen. Den Begriff der *Telematikinfrastruktur* hatten alle schon gehört, zum Teil auch einige für die Domäne relevante Begriffe mehr. Wissen über das betrachtete System, die Schnittstellen des Konnektors, bestand zuvor bei keinem. Das Arbeiten mit Java und Kotlin stellte für keinen der Teilnehmenden ein Problem dar. Die Erfahrung im Umgang mit IntelliJ variierte zwischen den Teilnehmenden: Manche benutzten Werkzeuge wie das Springen zu der Definition einer Funktion oder das Suchen eines Begriffs über die gesamte Code-Basis oft und aus Routine und Intuition heraus, während andere mit weniger Erfahrung nur sehr selten darauf zurückgriffen.

Jede Sitzung dauerte etwas weniger als die eingeplanten drei Stunden. Begonnen wurde mit einer ca. halbstündigen Einführung meinerseits. Zunächst wurden die gematik, die Telematikinfrastruktur, der Konnektor und die verschiedenen Karten kurz beschrieben. Ohne diese Einführung wäre das Bearbeiten von Aufgaben an der Schnittstelle in diesem komplexen Themenfeld in der gegebenen Zeit nicht möglich gewesen. Außerdem wurden die Begriffe Mandant, Primärsystem und Arbeitsplatz

im Rahmen der Vorstellung des Konnektorsimulators und seiner Weboberfläche erklärt. Es wurde auch gezeigt, wie man Karten in die verschiedenen vorbereiteten Terminals steckt. Ebenfalls wurde in die heruntergeladenen Dokumente eingeführt. Darunter war der Implementierungsleitfaden [10], die vollständige normative Spezifikation des Konnektors [8] und ein Glossar der Gematik [13]. Nach der Einführung gab es zwei Stunden Zeit zur Bearbeitung der drei im Folgeabschnitt beschriebenen Aufgaben. Im Anschluss sollten die Teilnehmenden fünf Minuten ihre Gedanken zum System nennen. Abgeschlossen wurde jede Sitzung mit weiteren fünf Minuten freier Diskussion im Dialog.

Eine Übersicht bzw. Benennung der bereitgestellten Dienste und aufrufbaren Funktionen wie z. B. getCards wurde bewusst *nicht* gegeben, da dies durch lesen der Dokumentation und des Codes eigenständig herausgefunden werden sollte.

Während der Beobachtung der Teilnehmenden habe ich Notizen gemacht, aufgenommen wurde in keiner Form. Hilfe wurde zu technischen Aspekten wie z. B. Kotlin oder der IDE gegeben, da diese mit dem untersuchten logischen Konzept des Konnektors und seiner Schnittstellen nichts zu tun haben. Fachliche Hilfe wurde nur gegeben, wenn die Teilnehmer über längere Zeit an einer Stelle festhingen. Die Teilnehmenden sollten während der Arbeit laut denken.

3.2.2 Aufgaben

Für jede der Aufgaben sind im Konnektorsimulator zuvor einige Karten gesteckt worden, um ein bestehendes Szenario im Praxisalltag darzustellen.

Für die erste Aufgabe musste nicht programmiert werden, sondern lediglich meine bereitgestellte CLI-Anwendung genutzt werden, um einen bestimmten Aufruf zusammenzubauen. Konkret sollte ermittelt werden, welche Karten von unserem Primärsystem und Mandanten aus zugreifbar sind, also über alle Arbeitsplätze hinweg. Abgezielt wurde auf eine Nutzung von getCards durch die Teilnehmenden.

Für die zweite Aufgabe sollte vorbereiteter Code ergänzt werden, in dem sich bereits ein leeres Aufrufkontext-Objekt und ein vorbereiteter Endpunkt des EventService befanden. Ziel der Aufgabe war, die Namen aller Patienten für alle gesteckten EGKs über alle Arbeitsplätze hinweg auf der Konsole anzuzeigen. Das Wissen um getCards aus Aufgabe 1 sollte dabei eine Hilfe sein.

Die letzte und schwerste Aufgabe bestand ebenfalls aus der Ergänzung von Code. In diesem war bereits ein vollständig funktioneller Aufruf von readVSD zum Lesen der Versichertendaten aufgeschrieben. Im Konnektorsimulator steckte allerdings neben der zu lesenden EGK eine noch freizuschaltende SMC-B, die zur Authentifizierung genutzt werden sollte. Die Teilnehmenden bekamen daher nach der ersten Ausführung des Codes eine entsprechende Fehlermeldung. Ziel der Aufgabe war es, durch Hinzufügen eines Aufrufs von verifyPin das Programm erfolgreich zum Laufen zu bringen.

Die genaue Formulierung der gestellten Aufgaben findet sich im Anhang.

3.2.3 Ergebnisse

In diesem Abschnitt sind die Benutzbarkeitsprobleme benannt, die in den Tests beobachtet worden sind. Auf einige Beispiele für gute Benutzbarkeit wird ebenfalls eingegangen.

Während der ersten und zweiten Aufgabe war es für die meisten Teilnehmenden nicht plausibel, dass auch bei gesetzter *mandantWide*-Option zum Erhalten *aller* Karten über *alle* Arbeitsplätze hinweg dennoch ein vollständiger Aufrufkontext angegeben werden muss, da in diesem ein spezifischer Arbeitsplatz benannt wird.

Quelle einiger Verwirrung und negativer Anmerkungen in der Abschlussdiskussion war die Signatur von getCards im Rahmen von Aufgabe 2. Die in Abschnitt 2 erwähnten Eingabeparameter werden der Funktion nicht direkt übergeben, sondern in einem GetCards Objekt als Felder gesetzt und dann gesammelt übergeben. Die Operation heißt also getCards mit kleinem Anfangsbuchstaben, der Parameter mit großem [10, S. 45]. GetCards als Name des Objekts liest sich dabei wie eine Funktion und nicht wie ein Aufrufparameter.

Einer der Teilnehmenden verstand fälschlicherweise zu Beginn der Arbeit an Aufgabe 2 den Parameter *CtId* (Id des Kartenterminals) [10, S. 45 f.] als Id es Aufrufkontextes. Dass eine solche nicht existiert, sondern der Kontext komplett als Objekt übergeben wird [10, S. 45 f.], wurde erst nach einigen Minuten realisiert. Auch wenn der Fehler wahrscheinlich auf geringes Domänenwissen zurückgeht, hätte er möglicherweise durch eine bessere Bennenung von *CtId*, beispielsweise als *CardTerminalId*, verhindert werden können.

Ein weiteres Problem ist der Typ des Parameters *PinTyp* zum Freischalten von Karten durch verifyPin in Aufgabe 3. Dieser wird als String übergeben, wobei die möglichen Strings „PIN.CH“ und „PIN.SMC“ sind. Für die SMC-B in der Aufgabe ist „PIN.SMC“ zu nutzen [8, S. 159]. Die Nutzung eines Enum-Typs bei dieser begrenzten Anzahl an Optionen wäre aus mehreren Gründen besser gewesen: Zum einen können dadurch (beispielsweise über Unterstützung der IDE) alle möglichen Optionen schnell ermittelt werden. Die Teilnehmenden mussten dafür im Fließtext des Implementierungsleitfadens [10] zum Teil länger suchen. Zum anderen würden durch den Enum Tippfehler als solche erkannt. Ein Teilnehmer tippte zum Beispiel zunächst „PIN.SCM“ statt „PIN.SMC“.

Das letzte bei der Mehrheit der Teilnehmer beobachtete Problem hat mit der Struktur der Rückgabe von getCards zu tun, wie sie in Abbildung 4 gezeigt ist. Es ist zu sehen, dass das Element des Eintrags für die Karten in der GetCardsResponse eine Sequenz von Karten ist. Übeträgt man dies auf eine objektorientierte Sicht entsteht das Problem. Das Cards-Objekt *enthält* im generierten Java-Code eine Liste von Karten namens Card als Feld, ist aber selber ein Objekt und keine Liste. Um über alle Karten zu iterieren, muss deshalb ein kontraintutiver Ausdruck der Form

```
for card in response.cards.card
```

genutzt werden. Dies ist den Teilnehmenden zum Teil erst nach mehreren fehlerhaften Aufrufen aufgefallen. Da Cards-Elemente sich als Sequenz von Elementen aus

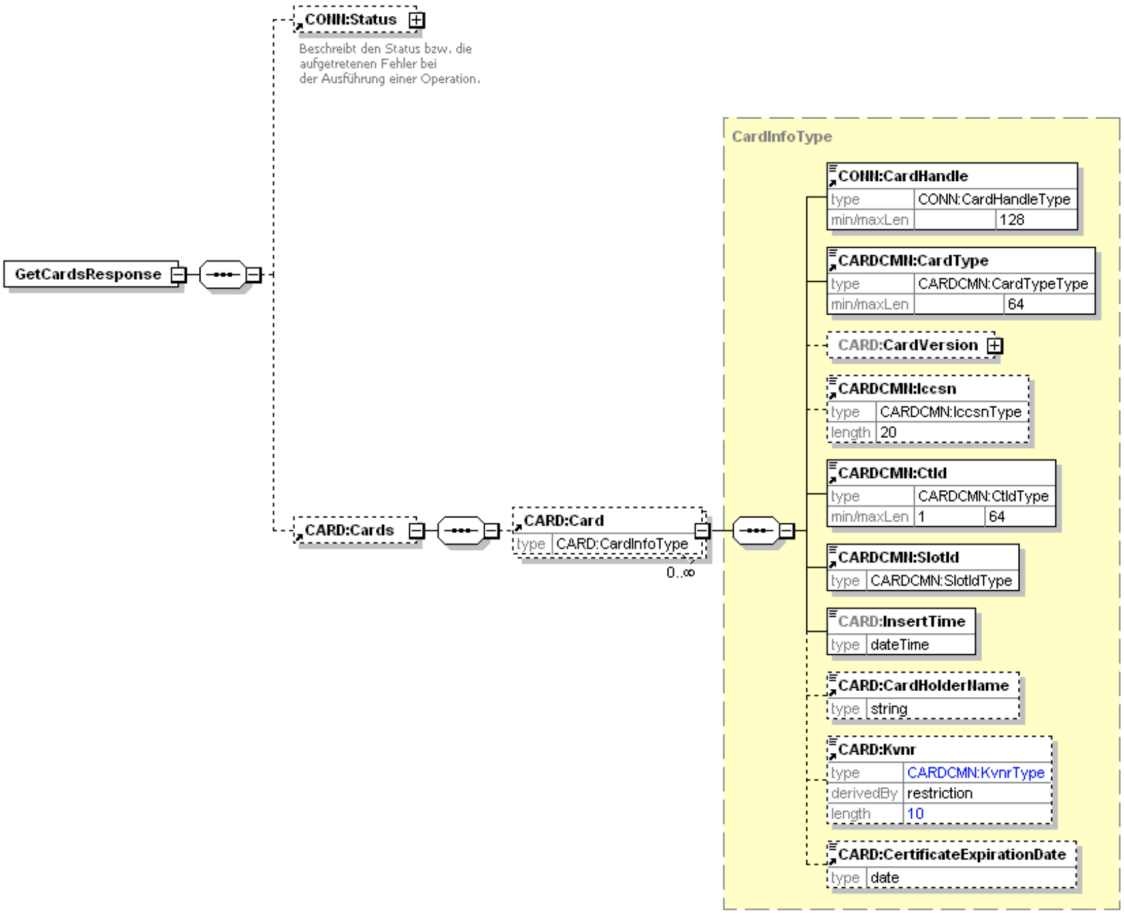

Abbildung 4: Schema der Rückgabe von getCards [10, S. 47]

nur einem Typen zusammensetzen, könnte man die Indirektion weglassen und das Card-Element direkt dem Antwortelement unterordnen. Ob solche Optimierungen vielleicht durch bessere Anwendung der genutzten Java-Werkzeuge zur Übersetzung von WSDL-Dateien automatisch geschehen, wurde nicht weiter untersucht.

Während der Tests sind auch einige positive Beispiele beobachtet worden.

Ein Vorteil der maschinenlesbaren Spezifikation der Operationen in WSDL und der daraus generierten Codebasis ist die Übersicht, die mit der Hilfe einer IDE möglich ist. Alle Teilnehmenden nutzten Funktionen wie das Springen zu Definitionen. Die erfahreneren Nutzer konnten über die Möglichkeit der Textsuche über alle Dateien hinweg herausfinden, in welchen Services eine gewünschte Funktion zu finden ist und so weniger Dokumentation lesen.

Auch das Glossar [13] war eine gute Hilfe und wurde an mehreren Stellen benutzt, um schnell Begriffe nachzuschlagen.

Geholfen hat zudem, dass eine Volltextsuche im Implementierungsleitfaden nach dem

Text der Fehlermeldung bei nicht freigeschalteter SMC-B in Aufgabe 3 zu einem Abschnitt mit passendem Lösungsvorschlag führt [10, S. 122 f.].

Drei von vier Teilnehmenden sind mit allen Aufgaben fertig geworden, die vierte Person hatte die Aufgabe zum Ende fast fertig. Es muss dabei beachtet werden, dass wie oben angegeben Hilfe gegeben wurde, wenn die Teilnehmenden über lange Zeit nicht vorangekommen sind.

3.3 Heuristische Evaluation

3.3.1 Anmerkungen zur Methodik

Die heuristische Evaluation ist hauptsächlich in einer ca. zweistündigen Sitzung *nach* der Auswertung der Benutzbarkeitstests geschehen. Nielsen beschreibt zwar eine umgekehrte Reihenfolge der Methoden, eine klare Notwendigkeit wird allerdings nicht ausgedrückt [35, S. 225 f.]. Außerdem kann sich die Erfahrung durch das Lernen der Benutzbarkeitsmethoden und ausführlichere Beschäftigung mit den Schnittstellen durch Beobachten der Teilnehmer positiv auf die Anzahl entdeckter Probleme in der Evaluation auswirken [35, S. 160 ff., 225].

Um einer Dopplung der benannten Probleme entgegenzuwirken, werden nur solche genannt, die kein Ergebnis der Benutzbarkeitstests sind. Allerdings ist bei manchen der folgenden Probleme angemerkt, wie die Erkenntnis aus den Benutzbarkeitstests ihre Entdeckung begünstigt hat. Da es in dieser Arbeit um die Benutzbarkeitsprobleme und nicht um die Bewertung des Anteils der einzelnen Methoden geht, stellt diese Voreingenommenheit während der Evaluation kein Problem dar.

3.3.2 Ergebnisse

Das erste Problem ist Ergebnis der Ausarbeitung des Kommentars eines der Testteilnehmer in der abschließenden Diskussion. In dieser wurde der Begriff „stringly typed“ erwähnt. Dieser humoristische Begriff bezeichnet die übermäßige Nutzung von Strings als Datentyp [30]. Eine Ausprägung dieses Problems, der Typ des *PinTyp*, ist bereits erwähnt worden. Eine weitere Folge ist, dass z. B. für readVSD zwei Parameter des gleichen Typs nacheinander eingegeben werden müssen [10, S. 55], was zu Verwechslungen führen kann. Verstoßen wird damit gegen das Prinzip Nummer 5 der sinnvollen Grenzen zur Fehlervermeidung.

Sowohl für die Operation requestCard als auch für getCards können die Id eines Kartenterminals und zugehörigen Steckplatzes angegeben werden [10, S. 45 f., 50]. Für requestCard werden beide als Teil eines Slot-Objekts zusammen übergeben [10, S. 50], für getCards jeweils einzeln [10, S. 45]. Das ist nicht konsistent und verstößt gegen Heuristik 4. Außerdem ist ein Steckplatz in der Realität gekoppelt an ein Kartenterminal, im Sinne der Heuristik 2 würde die Bündelung in einem Objekt dies stärker zum Ausdruck bringen.

Ebenfalls ein Verstoß gegen Heuristik 2 und 4 ist die Bennenung des Systeminformationsdienstes im Implementierungsleitfaden. Dieser wird technisch als EventService

bezeichnet, obwohl die deutsche Übersetzung, der Ereignisdienst nur ein Teil des Systeminformationsdienstes ist [10, S. 31].

Hinsichtlich des Pin-Typs PIN.CH gibt es ein weiteres Problem, und zwar die Abkürzung „CH“. Dies steht für *Card Holder* und ist für Versicherte relevant [13, S. 28], muss aber auch für verifyPin bei Freischaltung einer SMC-B übergeben werden [8, S. 159]. Auf die SMC-B wird im Glossar unter dem entsprechenden Eintrag kein Bezug genommen [13, S. 28], das ist inkonsistent (Heuristik 4).

Außerdem ist durch den Typen der übergebenen Karte in verifyPin klar impliziert, welcher PinTyp übergeben werden muss, die Angabe ist redundant [8, S. 159]. Es könnten stattdessen zwei Funktionen *verifyPinHBA* und *verifyPinSMCB* definiert werden, die jeweils nur einen Kartentypen akzeptieren und keine Angabe eines PinTyp erfordern. Das würde unnötige Informationen verstecken, durch die Suffixe dennoch klar unterscheidbar sein (vgl. Heuristik 8) und ein Beispiel für das Setzen von Grenzen zur Fehlervermeidung sein (Heuristik 5).

Das letzte Problem ist eine weitere Inkonsistenz im Implementierungsleitfaden. Darin wird eine Operation wie getCards zum Teil mit großem Anfangsbuchstaben geschrieben [10, S. 75], an anderer Stelle mit kleinem [10, S. 54]. Das ist zunächst intern inkonsistent, aber auch gegenüber gängiger Konventionen, Funktionen klein zu schreiben (Heuristik 4). Möglicherweise führt dieses Problem im Zusammenspiel mit dem in den Tests beschriebenen Problem zu dem Namen des Aufrufobjekts GetCards zu zusätzlicher Verwirrung.

Es ist zu beobachten, dass die letzten beiden genannten Probleme jeweils im Zusammenhang zur Qualität der Dokumentation stehen (vgl. Heuristik 10).

4 Zusammenfassung und Ausblick

Insgesamt wurden durch die heuristische Evaluation und die Benutzbarkeitstests elf Probleme aufgedeckt.

Zugleich sind bei den Tests auch einige positive Aspekte aufgefallen und die Teilnehmenden haben fast alle gestellten Aufgaben erfolgreich beendet. Eine abschließende Bewertung der Schnittstellen gestaltet sich daher schwer. Mehr Arbeit zur Einordnung (weiterer Teile) der Schnittstelle ist notwendig [2]:

Einige Anwendungen wie die elektronische Patientenakte wurden noch nicht behandelt. Darüber hinaus gibt es Verbesserungspotential bei der Anwendung der Benutzbarkeitsmethoden. Meine Notizen während der Benutzbarkeitstests waren durch die Aufregung und fehlende Erfahrung teilweise nicht mehr nachvollziehbar. Dadurch konnten einige Probleme nicht mit Sicherheit rekonstruiert werden und wurden deshalb hier nicht genannt. Mit mehr Teilnehmenden wäre es beispielsweise möglich gewesen, eine erste Testsitzung nur zur Optimierung des Testverfahrens zu nutzen [35, S. 174 f.]. Mit diesem Test des Tests könnten Verbesserungen wie eine standardisierte Struktur der Notizen eingearbeitet werden, bevor mit den eigentlichen Tests

[2] Im Titel dieser Arbeit wurde daher das erste Wort von „Bewertung“ zu „Untersuchung“ geändert.

begonnen wird.

Literatur

[1] Erik Christensen et al. *Web Services Description Language (WSDL) 1.1*. Ariba, International Business Machines Corporation, Microsoft, 15. März 2001. URL: https://www.w3.org/TR/2001/NOTE-wsdl-20010315 (besucht am 27. 08. 2022).

[2] Henry S. Thompson et al., eds. *XML Schema Part 1: Structures Second Edition*. World Wide Web Consortium, 28. Okt. 2004. URL: https://www.w3.org/TR/2004/REC-xmlschema-1-20041028/ (besucht am 13. 09. 2022). Die aktuellste Version ist https://www.w3.org/TR/xmlschema-1/.

[3] Roberto Chinnici et al., eds. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. World Wide Web Consortium, 26. Juni 2007. URL: https://www.w3.org/TR/2007/REC-wsdl20-20070626/ (besucht am 27. 08. 2022). Die aktuellste Version ist https://www.w3.org/TR/wsdl20/.

[4] Tim Bray et al., eds. *Namespaces in XML 1.0 (Third Edition)*. World Wide Web Consortium, 8. Dez. 2009. URL: https://www.w3.org/TR/2009/REC-xml-names-20091208/ (besucht am 27. 08. 2022). Die aktuellste Version ist http://www.w3.org/TR/xml-names/.

[5] Don Box et al. *Simple Object Access Protocol (SOAP) 1.1*. DevelopMentor, International Business Machines Corporation, Lotus Development Corporation, Microsoft, UserLand Software, 8. Mai 2000. URL: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/ (besucht am 27. 08. 2022). Die aktuellste Version ist www.w3.org/TR/SOAP.

[6] eHealth Experts GmbH. *Konnektorsimulator für Primärsysteme*. Software veröffentlicht durch gematik GmbH. Version 3.1.15. 21. Juni 2022. URL: https://cloud.gematik.de/index.php/s/QNGzF4wCd5jtoSd (besucht am 13. 09. 2022).

[7] Sascha Fahl et al. „Rethinking SSL Development in an Appified World“. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: Association for Computing Machinery, 2013, S. 49–60. ISBN: 9781450324779. DOI: 10.1145/2508859.2516655. URL: https://doi.org/10.1145/2508859.2516655 (besucht am 12. 09. 2022).

[8] gematik GmbH. *Spezifikation Konnektor*. Dokument [Online]. Version 4.11.1. 27. Apr. 2017. URL: https://fachportal.gematik.de/fileadmin/Fachportal/Downloadcenter/Historie/Versionsstand_vom_04.02.2020_-_Release_3.1.2/OPB3.1_R3.1.2_Spezifikationen_20191122.zip (besucht am 13. 09. 2022). Dateiname im Paket ist gemSpec_Kon_V4.11.1.pdf.

[9] gematik GmbH. *Schnittstellenspezifikation Primärsysteme VSDM*. Dokument [Online]. Version 1.5.0. 14. Mai 2018. URL: https://fachportal.gematik.de/fileadmin/Fachportal/Downloadcenter/Historie/Versionsstand_vom_04.02.2020_-_Release_3.1.2/OPB3.1_R3.1.2_Spezifikationen_20191122.zip (besucht am 13. 09. 2022). Dateiname im Paket ist gemSpec_SST_PS_VSDM_V1.5.0.pdf.

[10] gematik GmbH. *Implementierungsleitfaden Primärsysteme – Telematikinfrastruktur (TI) (einschließlich VSDM, QES-Basisdienste, KOM-LE)*. Dokument [Online]. Version 2.5.0. 2. Okt. 2019. URL: https://fachportal.gematik.de/fileadmin/Fachportal/Downloadcenter/Historie/Versionsstand_vom_04.02.2020_-_Release_3.1.2/OPB3.1_R3.1.2_Spezifikationen_20191122.zip (besucht am 13. 09. 2022). Dateiname im Paket ist gemILF_PS_V2.5.0.pdf.

[11] gematik GmbH. *Spezifikation eHealth-Kartenterminal*. Dokument [Online]. Version 3.11.0. 2. Okt. 2019. URL: https://fachportal.gematik.de/fileadmin/Fachportal/Downloadcenter/Historie/Versionsstand_vom_04.02.2020_-_Release_3.1.2/OPB3.1_R3.1.2_Spezifikationen_20191122.zip (besucht am 13. 09. 2022). Dateiname im Paket ist gemSpec_KT_V3.11.0.pdf.

[12] gematik GmbH. *Dokumentenlandkarte Online-Produktivbetrieb - Festlegung der Versionsstände*. Dokument [Online]. Version 5.1.3, Release 3.1.2. 4. Feb. 2020. URL: https://fachportal.gematik.de/fileadmin/Fachportal/Downloadcenter/Historie/Versionsstand_vom_04.02.2020_-_Release_3.1.2/gemDokLK_R3.1.2_V5.1.3.pdf (besucht am 12. 09. 2022).

[13] gematik GmbH. *Glossar der Telematikinfrastruktur*. Dokument [Online]. Version 5.2.0. 20. Jan. 2022. URL: https://fachportal.gematik.de/fileadmin/Fachportal/Glossar/gemGlossar_V5.2.0.pdf (besucht am 13. 09. 2022).

[14] gematik GmbH. *Technische Ausstattung einer medizinischen Einrichtung*. Informationsblatt [Online]. Berlin, Deutschland, Juni 2022. URL: https://fachportal.gematik.de/fileadmin/Fachportal/Leistungserbringer/gematik_Infoblatt_Technische_Ausstattung_Medizinische_Einrichtung_web_20220615.pdf (besucht am 25. 08. 2022).

[15] gematik GmbH. *Die elektronische Gesundheitskarte*. gematik.de. URL: https://www.gematik.de/telematikinfrastruktur/egk (besucht am 29. 05. 2022).

[16] gematik GmbH. *E-Patientenakte*. gematik.de. URL: https://www.gematik.de/anwendungen/e-patientenakte (besucht am 28. 05. 2022).

[17] gematik GmbH. *eHealth-Kartenterminal*. gematik Fachportal. URL: https://fachportal.gematik.de/hersteller-anbieter/komponenten-dienste/ehealth-kartenterminal (besucht am 29. 05. 2022).

[18] gematik GmbH. *Historie*. gematik Fachportal. URL: https://fachportal.gematik.de/downloadcenter/historie (besucht am 13. 09. 2022).

[19] gematik GmbH. *Konnektor*. gematik Fachportal. URL: https://fachportal.gematik.de/hersteller-anbieter/komponenten-dienste/konnektor (besucht am 29. 05. 2022).

[20] gematik GmbH. *KoPS*. gematik Fachportal. URL: https://fachportal.gematik.de/toolkit/kops (besucht am 27. 08. 2022).

[21] gematik GmbH. *Notfalldaten*. gematik.de. URL: https://www.gematik.de/anwendungen/notfalldaten (besucht am 28. 05. 2022).

[22] gematik GmbH. *Primärsysteme*. gematik Fachportal. URL: https://fachportal.gematik.de/hersteller-anbieter/primaersysteme (besucht am 28. 05. 2022).

[23] gematik GmbH. *Telematikinfrastruktur*. gematik.de. URL: https://www.gematik.de/telematikinfrastruktur (besucht am 28. 05. 2022).

[24] gematik GmbH. *Über uns - Gesetzliche Grundlagen*. gematik.de. URL: https://www.gematik.de/ueber-uns/gesetzliche-grundlagen (besucht am 24. 08. 2022).

[25] gematik GmbH. *Über uns - Struktur*. gematik.de. URL: https://www.gematik.de/ueber-uns/struktur (besucht am 28. 05. 2022).

[26] gematik GmbH. *Versichertenstammdaten-Management*. gematik Fachportal. URL: https://fachportal.gematik.de/anwendungen/versichertenstammdatenmanagement (besucht am 13. 09. 2022).

[27] *Jakarta XML Web Services. Eclipse Implementation of XML Web Services*. Version 4.0.0. Eclipse Foundation. URL: https://eclipse-ee4j.github.io/metro-jax-ws/ (besucht am 13. 09. 2022).

[28] Bundesamt für Justiz. *SGB 5 §306*. Gesetze im Internet. URL: https://www.gesetze-im-internet.de/sgb_5/__306.html (besucht am 28. 05. 2022). (nichtamtliche Fassung).

[29] Bundesamt für Justiz. *SGB 5 §311*. Gesetze im Internet. URL: https://www.gesetze-im-internet.de/sgb_5/__311.html (besucht am 28. 05. 2022). (nichtamtliche Fassung).

[30] Suzanne Kemmer. *stringly typed*. The Rice University Neologisms Database. URL: https://neologisms.rice.edu/index.php?a=term&d=1&t=14876 (besucht am 12. 09. 2022).

[31] *kotlinx-cli*. GitHub-Repository. JetBrains. URL: https://github.com/Kotlin/kotlinx-cli (besucht am 13. 09. 2022).

[32] Mitra, Nilo and Lafon, Yves, eds. *SOAP Version 1.2 Part 0: Primer (Second Edition)*. World Wide Web Consortium, 27. Apr. 2007. URL: https://www.w3.org/TR/2007/REC-soap12-part0-20070427/ (besucht am 27. 08. 2022). Die aktuellste Version ist https://www.w3.org/TR/soap12-part0/.

[33] Kate Moran. *Usability Testing 101*. nngroup.com. URL: https://www.nngroup.com/articles/usability-testing-101/ (besucht am 01. 12. 2019).

[34] Brad A. Myers und Jeffrey Stylos. „Improving API Usability". In: *Commun. ACM* 59.6 (Mai 2016), S. 62–69. ISSN: 0001-0782. DOI: 10.1145/2896587. URL: https://doi.org/10.1145/2896587 (besucht am 13. 09. 2022).

[35] Jakob Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. ISBN: 9780080520292. URL: https://dl.acm.org/doi/book/10.5555/2821575 (besucht am 13. 09. 2022).

[36] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. nngroup.com. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (besucht am 13. 09. 2022).

[37] Lutz Prechelt. *Analytische Qualitätssicherung 1*. Vorlesung "Softwaretechnik". Vorlesungsfolien. 2020. URL: http://www.inf.fu-berlin.de/inst/ag-se/teaching/V-SWT-2020/14_Analytische-QS.pdf. (aufgerufen am 25.08.2022).

[38] Martin P. Robillard und Robert Deline. „A Field Study of API Learning Obstacles". In: *Empirical Softw. Eng.* 16.6 (Dez. 2011), S. 703–732. ISSN: 1382-3256. DOI: 10.1007/s10664-010-9150-8. URL: https://doi.org/10.1007/s10664-010-9150-8.

[39] Thomas Scheller und Eva Kühn. „Automated measurement of API usability: The API Concepts Framework". In: *Information and Software Technology* 61 (Mai 2015), S. 145–162. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.01.009. URL: https://www.sciencedirect.com/science/article/pii/S0950584915000178.

A Anhang

A.1 Genutzte Version der gematik-Dokumente

Der Konnektorsimulator simuliert laut seinem Dienstverzeichnis einen Konnektor der Version 4.3.1. Diese Version ist Teil des Gesamtreleases 3.1.2 von der gematik [12].

Die spezifizierenden und informativen Dokumente sowie die benötigten WSDL- und XSD-Dateien dieses Releases finden sich unter [18].

Sind in dieser Arbeit zu bestimmten Diensten gehörige XSD- oder WSDL-Dateien erwähnt, so wird sich immer auf die zum Release gehörige Version bezogen.

A.2 Beispiel für Aufruf über SOAP

Folgende Abbildung zeigt eine SOAP-Nachricht beim Aufruf von getCards durch den Klienten an den Konnektorsimulator. Die Parameter entsprechen dem in Abschnitt 3.1 beschriebenen Beispiel.

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header />
    <S:Body xmlns:ns10="urn:oasis:names:tc:dss:1.0:core:schema"
        xmlns:ns11="http://www.w3.org/2000/09/xmldsig#"
       xmlns:ns12="urn:oasis:names:tc:SAML:1.0:assertion"
       xmlns:ns2="http://ws.gematik.de/conn/ConnectorCommon/v5
       .0" xmlns:ns3="http://ws.gematik.de/tel/error/v2.0"
       xmlns:ns4="http://ws.gematik.de/conn/EventService/v7.2"
       xmlns:ns5="http://ws.gematik.de/conn/ConnectorContext/v2
       .0" xmlns:ns6="http://ws.gematik.de/conn/
       CardServiceCommon/v2.0" xmlns:ns7="http://ws.gematik.de/
       conn/CardService/v8.1" xmlns:ns8="http://ws.gematik.de/
       int/version/ProductInformation/v1.1" xmlns:ns9="http://
       ws.gematik.de/conn/CardTerminalInfo/v8.0">
        <ns4:GetCards mandant-wide="true">
            <ns5:Context>
                <ns2:MandantId>Mandant1</ns2:MandantId>
                <ns2:ClientSystemId>ClientID1</
                   ns2:ClientSystemId>
                <ns2:WorkplaceId>Workplace1</ns2:WorkplaceId>
            </ns5:Context>
            <ns6:CardType>EGK</ns6:CardType>
        </ns4:GetCards>
    </S:Body>
</S:Envelope>
```

Codebeispiel 4: SOAP-Nachricht beim Aufruf von getCards

A.3 Gestellte Aufgaben im Originallaut

1. Finde heraus, welche Karten momentan von unserem Primärsystem/Mandanten aus zugreifbar sind. Nutze dafür die bereitgestellte Anwendung in der Kommandozeile. Der erste Aufruf ist bereits vorbereitet (siehe Terminal).
2. Ergänze den Code in der Datei `aufgabe2.kt`, sodass die Namen der Patienten für alle gesteckten EGKs über alle Arbeitsplätze hinweg auf der Konsole angezeigt werden. Für andere Kartentypen sollen keine Informationen ausgegeben werden.
3. Betrachte das Ergebnis der Ausführung des bisherigen Codes für diese Aufgabe (unter Kenntnisnahme des Zustands des Konnektors und der gesteckten Karten). Ergänze den Code in der Datei `aufgabe3.kt`, sodass die Versichertenstammdaten der gesteckten EGK auf der Konsole angezeigt werden, ohne dass eine weitere Karte neben den bereits vorhandenen gesteckt oder entfernt werden muss.