CARL VON OSSIETZKY universität OLDENBURG

Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik, Abteilung Softwaretechnik

Projektgruppe „DORI“

Projektdokumentation

Themensteller: Prof. Dr. Andreas Winter, Jan Jelschen, Johannes Meier

Vorgelegt von: Christopher Bischopink, Stephan Bogs, Hauke Fischer, Hannah Meyer
Felix Kempa, Nancy Kramer, Thomas Sprock, Lisa Ripke

Abgabetermin: 28.03.2018

Inhaltsverzeichnis

1. Inhalt der Projektdokumentation

Der erste Teil dieser Dokumentation – die Projektdokumentation – stellt die Projektgruppe DORI vor und beschreibt ihre Motivation und Ziele. Anschließend werden dort wichtige Begrifflichkeiten textuell sowie in Form eines Domänenmodells erläutert. Über verschiedene Use Cases werden die Anforderungen abgeleitet, die im Rahmen der Projektgruppe zu erfüllen sind. Die Use Cases und Anforderungen wurden aus diversen Quellen abgeleitet, die näher im zugehörigen Kapitel erläutert werden. Abschließend folgt dort eine Erläuterung der Projektorganisation. Dazu zählt die Beschreibung der einzelnen Rollen und des Vorgehensmodells. Außerdem werden die verwendeten Tools und Regeln erläutert, die innerhalb der Projektgruppe von jedem Mitglied zu beachten sind.

Im nächsten Teil – der Systemdokumentation – wird beschrieben, wie sich das System der Projektgruppe DORI zusammensetzt. Dazu wird das Prinzip der Siemensarchitektur angewandt. Es werden anhand der globalen Analyse verschiedene Faktoren herausgefiltert, die einen Einfluss auf die Entscheidungsfindungen für verschiedene Technologien haben. Anhand der getroffenen Technologie-Entscheidungen konnten die Konzeptsicht und die Modulsicht beschrieben werden, um die Aufteilung der Komponenten auf ihre Funktionalitäten und die Aufteilung der Komponenten in ihre Teilkomponenten zu beschreiben.

Der darauffolgende Teil der Dokumentation – das Projekttagebuch – beschreibt unsere Sprints und zeigt somit die Fortschritte, die erreicht wurden. Anhand dieser Dokumentation kann der Prozess für die Entwicklung des DORI-Gesamtsystems nachvollzogen werden. Die Beschreibungen der einzelnen Sprints enthalten eine Beschreibung der Ziele sowie, ob diese umgesetzt wurden. Die Retrospektiven sowie die Sprint Reviews geben Aufschluss darüber, was während der Sprints gut und was schlecht gelaufen ist.

Im nächsten Teil – der Testdokumentation – wird die Herangehensweise der Erstellung von Tests für die Projektgruppe sowie deren Funktion dargestellt. Dazu wird zuerst das allgemeine Testkonzept erläutert. Anschließend ist eine Beschreibung der einzelnen Testmodelle zu finden, auf die sich das Testkonzept stützt. Um eine komplette Testabdeckung zu zeigen, wird das Metamodell mit den einzelnen Testmodellen abgeglichen. Der Editor wurde speziell anhand seiner Use Cases getestet und die Beschreibung dieser Tests sowie eine Beschreibung der Tests zu den verschiedenen Sichten aufgeführt. Ebenfalls befinden sich

dort Beschreibungen der Anmerkungen zu der konkreten Syntax und zu der Usability des Editors. Diese entstammen aus dem erstellten Anwendungsbeispiel des Telefonbuches sowie der Erstellung der Testmodelle. Abschließend folgt dort eine Beschreibung des großen Anwendungsbeispiels – der Konferenzverwaltung. Anhand dieser wird eine finale Evaluation des Editors sowie es Interpreters beschrieben.

Zuletzt – im Ausblick und Fazit – soll ein Überblick über die von der Projektgruppe erreichten Ziele gegeben und diese bewertet werden. Ebenso werden Ideen für eine Erweiterung der Produkte der Projektgruppe gegeben.

Teil I.

Projektdokumentation

2. Motivation

Stephan, Hannah, Nancy, Thomas

Softwaresysteme brauchen heutzutage intensive Interaktionsmöglichkeiten. Unter einer Interaktion wird hier die Anpassung der graphischen Oberfläche als Reaktion auf Benutzereingaben oder Systemveränderungen verstanden. Das Umsetzen komplexer Interaktionsmöglichkeiten stellt die Beteiligten der Softwareentwicklung vor große Herausforderungen. Im Folgenden werden diese Herausforderungen und mögliche Lösungsansätze näher erläutert. Es gibt jedoch viele Möglichkeiten zur Umsetzung einer interaktiven Software. Daher wird sich hier auf jene Software beschränkt, die sich in der Architektur auf die vier Schichten Datenzugriff, Programmlogik, Interaktion und GUI bedient.

Tab. 2.1.: Vier-Schichten-Architektur

Schicht	**Beschreibung**	**Rolle**
Datenzugriffsschicht	Diese Schicht stellt die relevanten Daten für die Programmlogik durch persistente Datenhaltung zur Verfügung.	Datenbank-Entwickler
Programmlogik	Diese Schicht stellt die Funktionalität der Software zur Verfügung.	Programmlogik-Entwickler
Interaktionsschicht	Diese Schicht definiert die Reaktion der Software auf die Benutzereingaben.	Interaktionsentwickler
GUI-Schicht	Diese Schicht stellt die graphischen Oberflächen für den Benutzer zur Verfügung.	GUI-Entwickler

Die Entwickler der vier Schichten arbeiten an ihren individuellen Lösungen, die jedoch in eine einheitliche Gesamtsoftware überführt werden müssen. Dadurch ergibt sich ein Zusammenspiel der Rollen. Für die Datenzugriffsschicht entwirft der Datenbank-Entwickler ein Datenhaltungsmodell und realisiert die dazu passende Datenstruktur. Die Programmlogik wird vom

Programmlogik-Entwickler entwickelt, der die Funktionen bereitstellt und diese gegebenenfalls durch eine API bereitstellt. Die entwickelte Programmlogik setzt auf den Komponenten des Datenbank-Entwicklers auf. Der GUI-Entwickler schreibt die Widgets der GUI-Schicht. Unter einem Widget wird eine graphische Maske verstanden, die Nutzerinteraktionen und Systemveränderungen entgegen nimmt und Informationen darstellt.

Für den Übergang von einem Widget zum nächsten und die dafür notwendigen API-Funktionsaufrufe als Reaktion auf Nutzereingaben gibt es jedoch keine allgemeingültige Beschreibung. Daher wird für diesen Übergang zur Zeit für jede Software eine neue Individuallösung geschaffen. Der Interaktionsentwickler existiert nur in der Theorie, seine in Tabelle 2.1 durch die Interaktionsschicht beschriebenen Aufgaben werden je nach verwendeter Plattform vom GUI- oder Programmlogik-Entwickler umgesetzt [vgl. 4, 8].

Ein Nachteil, der sich dadurch ergibt, ist dass die Interaktionsmöglichkeit nur implizit im Quellcode fest verankert wird. Dadurch entsteht das Problem, dass der Code und somit auch die in Tabelle 2.1 beschriebenen Rollen verschwimmen. Daraus ergibt sich keine klare Trennung der Kompetenzbereiche. Hoher Koordinations- und Kommunikationsaufwand sind die Folge. Ein weiteres Problem stellt die schwierige nachträgliche Abänderung der Gesamtinteraktion dar. Unter einer Gesamtinteraktion wird hier die Verkettung aller einzelnen Interaktionen verstanden. Eine praktische Realisierung des Interaktionsentwicklers würde die Komplexität der anderen Rollen reduzieren, besonders die an der Schnittstelle befindlichen Rollen des GUI- und Programmlogik-Entwicklers. Weiterhin ist es derzeit noch nicht möglich, eine Gesamtinteraktion plattformunabhängig zu modellieren und anschließend auf verschiedenen Plattformen zu realisieren.

DORI („Do your Own Reactive Interface“) ist eine Projektgruppe von der Abteilung Softwaretechnik an der Carl von Ossietzky Universität Oldenburg. Diese Projektgruppe hat das Ziel, ein Konzept zu entwickeln, welches das Modellieren der Interaktion isoliert von den anderen Schichten ermöglicht. Dadurch soll die theoretische Rolle des Interaktionsentwicklers auch eine reale Rolle in der Softwareentwicklung bekommen. Zur Umsetzung und Validierung des Konzeptes müssen folgende vier Ziele umgesetzt werden:

1. Entwicklung eines plattformunabhängigen Konzeptes zur graphischen Darstellung einer Gesamtinteraktion und hierfür eine plattformunabhängige Beschreibungsmöglichkeit für Widgets und Funktionalitäten. Dabei ist zu beachten, dass Widgets auch geschachtelt auftreten können.

2. Erstellung eines Modellierungswerkzeuges zur Eingabe einer Gesamtinteraktion. Dieses Werkzeug wird im Folgenden als DORI-Editor bezeichnet.

3. Erstellung eines Werkzeuges zur Übersetzung der modellierten Gesamtinteraktion in mehrere Endanwendungen auf verschiedenen Plattformen. Dieses Werkzeug wird im

Folgenden Übersetzer genannt. Die Übersetzung kann als Generator oder Interpreter realisiert werden.

4. Validierung des Konzeptes und der Werkzeuge anhand eines konkreten Anwendungsbeispiels. Dies beinhaltet die konkrete Implementierung dieser Software mithilfe des DORI-Editor und des Übersetzers.

Die Auswirkungen der Gesamtinteraktion auf die Datenbank ist kein Ziel der Projektgruppe, da die Programmlogik eine Schnittstelle zur Datenzugriffsschicht bereitstellt. Dies hat zur Folge, dass die Rolle des Datenbank-Entwicklers für die Projektgruppe nicht weiter relevant ist. Gleiches gilt für die Realisierung der Funktionalität, hier ist nur die Schnittstelle zur Programmlogik wichtig, sodass eine Validierung möglich ist.

Für die Umsetzung dieser Ziele gibt es verschiedene Lösungsansätze. Wir legen den Fokus auf die folgenden:

Lösungsansatz zu Ziel 1 Für die Programmlogik-Entwicklung gibt es einen modellgetriebenen Ansatz namens SENSEI („Software Evolution Services“), welcher Flussdiagramme zur Orchestrierung von Services verwendet [vgl. 3]. Angelehnt an diesen Ansatz wird das Paradigma der Zustandsautomaten verwendet, um eine Gesamtinteraktion darzustellen. Mit Zustandsautomaten kann modelliert werden, wie sich das System in bestimmten Zuständen bei gewissen Ereignissen verhält [11]. Die Realisierung des Verhaltens der graphischen Benutzeroberfläche erfolgt mittels der Verbindung auf einen zuvor modellierten Zustandsautomaten. In diesem Fall kann die Beschreibung der Widgets und Funktionalitäten der Benutzeroberfläche an die Zustände und Zustandsübergänge gebunden werden. Die abstrakte Syntax der gewählten DSL (Domain Specific Language) wird in einem Metamodell festgehalten. Die DSL wird im Folgenden als DORI-DSL bezeichnet.

Lösungsansatz zu Ziel 2 Es wird eine Anwendung geschrieben, mit welcher die Gesamtinteraktion in der festgelegten Sprache entworfen, editiert und gespeichert werden kann. Es gibt bereits Softwarelösungen, wie den RSAD (Rational Software Architect Designer), Visual Paradigm und Kotlett, die die graphische Modellierung von UML Modellen ermöglichen. Der Editor wird eine ähnliche graphische Modellierung für die DORI-DSL beinhalten. Dies wird durch eine Eigenimplementierung oder durch die Einbindung bestehender Modellierungstools realisiert.

Lösungsansatz zu Ziel 3 Für die Übersetzung in eine Endanwendung stehen die Möglichkeiten eines Generators oder Interpreters zur Verfügung. Während der Generator den Code der Endanwendung aus dem Modell generiert, führt ein Interpreter auf Basis des Modells die Endanwendung aus. Eine Entscheidung für eine dieser Möglichkeiten muss von der Projektgruppe getroffen werden. Es kann die Möglichkeit geschaffen werden, den Übersetzer aus dem Editor heraus aufrufen zu können.

Lösungsansatz zu Ziel 4 Zur Validierung stehen der Projektgruppe verschiedene Anwendungsszenarien zur Verfügung. Hierzu zählen etwa Projekte wie NEMo[vgl. 1] (Verbesserung der Mobilität im ländlichem Raum) oder eine selbst definierte Anwendung. Eine Entscheidung für eines dieser Anwendungsszenarien muss von der Projektgruppe getroffen werden. Dieses Anwendungsszenario soll grundlegende Interaktionsmöglichkeiten beinhalten.

Im folgenden Kapitel werden die zuvor eingeführten Begrifflichkeiten anhand eines Domänenmodells sowie eines erklärenden Textes miteinander in Beziehung gesetzt und näher beschrieben. Zudem werden hier einige neue Begrifflichkeiten eingeführt, die das Gesamtkonzept vervollständigen.

3. Domänenmodell

Christopher, Hauke , Lisa, Stephan

Das Domänenmodell ist in Form eines Klassendiagramms in Abbildung 3.1 zu finden. Es veranschaulicht die zentralen Begriffe und deren Beziehungen untereinander. Für eine präzisere Begriffsabgrenzung beschreibt der folgende Text die Begrifflichkeiten des Metamodells fettgedruckt, die in diesem Dokument benutzt werden.

Zentral für die Entwicklung einer **Endanwendung** sind die **Entwickler**. Von diesen Entwicklern gibt es im Kontext unserer Domäne drei verschiedene Rollen, deren Arbeitsgebiete stark variieren. Diese sind **Interaktionsentwickler**, **GUI-Entwickler** und **Programmlogikentwickler**. Da ihre Kompetenzen klar voneinander abgegrenzt sind (wie schon in Kapitel 2 beschrieben), werden sie an dieser Stelle separat betrachtet. Begonnen wird mit dem Interaktionsentwickler, da seine Rolle zentral für das Verständnis der meisten Begrifflichkeiten ist:

Ein **Interaktionsentwickler** möchte in Verbindung mit anderen Entwicklern eine Endanwendung entwickeln. Er entwirft die Interaktionsmöglichkeiten von potenziellen Nutzern mit dieser Anwendung und definiert so eine **Gesamtinteraktion**. Unter einer Gesamtinteraktion wird hier die Verkettung aller einzelner Interaktionen verstanden. Diese Gesamtinteraktion wird auf einer abstrakten Ebene durch ein **Interaktionsmodell** repräsentiert. Für die Erstellung und Bearbeitung dieses Interaktionsmodells benutzt der Interaktionsentwickler den **DORI-Editor**. Dies geschieht im **Arbeitsbereich** des Editors, welcher außerdem das Interaktionsmodell visualisiert. Das Interaktionsmodell, dessen Entwicklung Aufgabe des Interaktionsentwicklers ist, ist in der **DORI-Domain-Specific Language (DORI-DSL)** verfasst. Die in der DORI-DSL enthaltenen Modelle bestehen aus verschiedenen **Sprachelementen**, die jeweils eine **konkrete Syntax** (also eine Definition des Aussehens) besitzen. Diese konkrete Syntax wird von dem Arbeitsbereich des DORI-Editors verwendet, um konkreten Sprachelemente zu visualisieren.

Die Bezeichnungen für die Elemente der DORI-DSL orientiert sich – wie bereits in den Lösungsansätzen angeschnitten – an Zustandsautomaten. So gibt es **AbstractWidgetInstances** und **Transitionen** zwischen diesen AbstractWidgetInstances. Eine AbstractWidgetInstance beschreibt alles, was ein Nutzer zu einem bestimmten Zeitpunkt in der GUI sehen

kann und welche Interaktionsmöglichkeiten es gibt sowie die dafür benötigten Informationen. Eine AbstractWidgetInstances liegt hierbei in einem **WidgetSocket**, ein WidgetSocket kommt dabei eine ähneliche Rolle wie einer Region in einem Zustandsautomaten zu. In einem WidgetSocket sind eine spezifische Anzahl an AbstractWidgetInstances beheimatet. Die Transitionen, welche definieren, wie zwischen Interaktionszuständen gewechselt werden kann, werden über **Bedingungen** kontrolliert. Von diesen Bedingungen gibt es drei Arten, **Events**, **Trigger** und den **Vergleich** zweier Terme (**Guard**). Events sind Nutzereingaben oder Systemevents. Die Terme der Guards arbeiten auf Daten, die im Interaktionszustand vorhanden sind. Trigger sind Mengen von Events und leiten Aktionen ein.

Die zuvor genannten Widgets sind graphische Masken, welche Nutzerinaktionen und Systemveränderungen entgegennehmen und Informationen graphisch darstellen. **AbstractWidgets** sind plattformunabhängige Beschreibungen dieser graphischen Masken und der dafür benötigten Daten. Diese werden an AbstractWidgetInstances gebunden. Dieser Vorgang wird „**Bindung zum Abstarkten Widget**“ genannt. **AbstractFunctions** sind plattformunabhängige Beschreibungen von Funktionsaufrufen der Programmlogik. An Transitionen lassen sich **Actions** und **Parameterbindings** binden. An Actions lassen sich **AbstractFunctions** binden, was als „**Bindung zur abstrakten Funktion**“ beschrieben wird. Parameterbindings sind für die Verknüpfung von Paramtern mit Actions, AbstractFunctions und weiteren Elementen der DORI-DSL zuständig. Dies entspricht einer Bindung der Daten an ihr jeweiliges Ziel und dient der Verwaltung und Modellierung des Datenflusses.

Sowohl auf eine abstrakte Funktion als auch an ein abstraktes Widget (zusammengefasst bezeichnet als **abstrakter Inhalt**) lässt sich eine **ConcreteFunction** bzw. ein **ConcrteWidget** (zusammengefasst **konkreter Inhalt**) binden, was als „**Mapping zur Widget-**“ bzw. „**Mapping zur Funktionsimplementierung**“ beschrieben wird. Unter einer konkreten Funktion bzw. einem konkreten Widget wird hier die Verlinkung auf eine plattformspezifische Implementierung verstanden.

Durchgeführt werden alle vier Vorgänge der Mappings durch den Interaktionsentwickler. Ein detailierter Blick auf alle Elemente der DORI-DSL ist im Abschnitt Benuterdokumentation 8.3 zu finden.

Möchte der Interaktionsentwickler aus seinem Modell eine Endanwendung erstellen, so geschieht dies mit Hilfe eines **Interpreters** oder **Generators** (zusammengefasst bezeichnet als **Übersetzer**). Dieser benötigt dafür das Modell sowie die plattformspezifischen Implementierungen, auf die von den konkreten Inhalten verlinkt wird.

Der **GUI-Entwickler** ist für die Implementierung der konkreten Widgets im Hinblick auf die jeweilige **Plattform** zuständig. Diese konkreten Widgets implementieren abstrakte Widgets, welche der Interaktionsentwickler erstellt hat. Der **Programmlogikentwickler** ist für die Implementierung der konkreten Funktionen im Hinblick auf die jeweilige Plattform zuständig.

Diese konkreten Funktionen implementieren abstrakte Funktionen, welche der Interaktionsentwickler erstellt hat. Daher besitzt ein abstrakter Inhalt eine **Beschreibung**, die angibt, was der entsprechende Entwickler zu implementieren hat.

Unter dem **Gesamtsystem** (nicht in der Abbildung) wird die Kombination aus DORI-DSL, DORI-Editor und Übersetzter verstanden.

Der lila Bereich beschreibt das Binding zu ConcreteFunctions und das Mapping zu AbstractFunctions. Der grüne Bereich beschreibt das Binding zum AbStarctWidget und Mapping zum AbstractWidget.

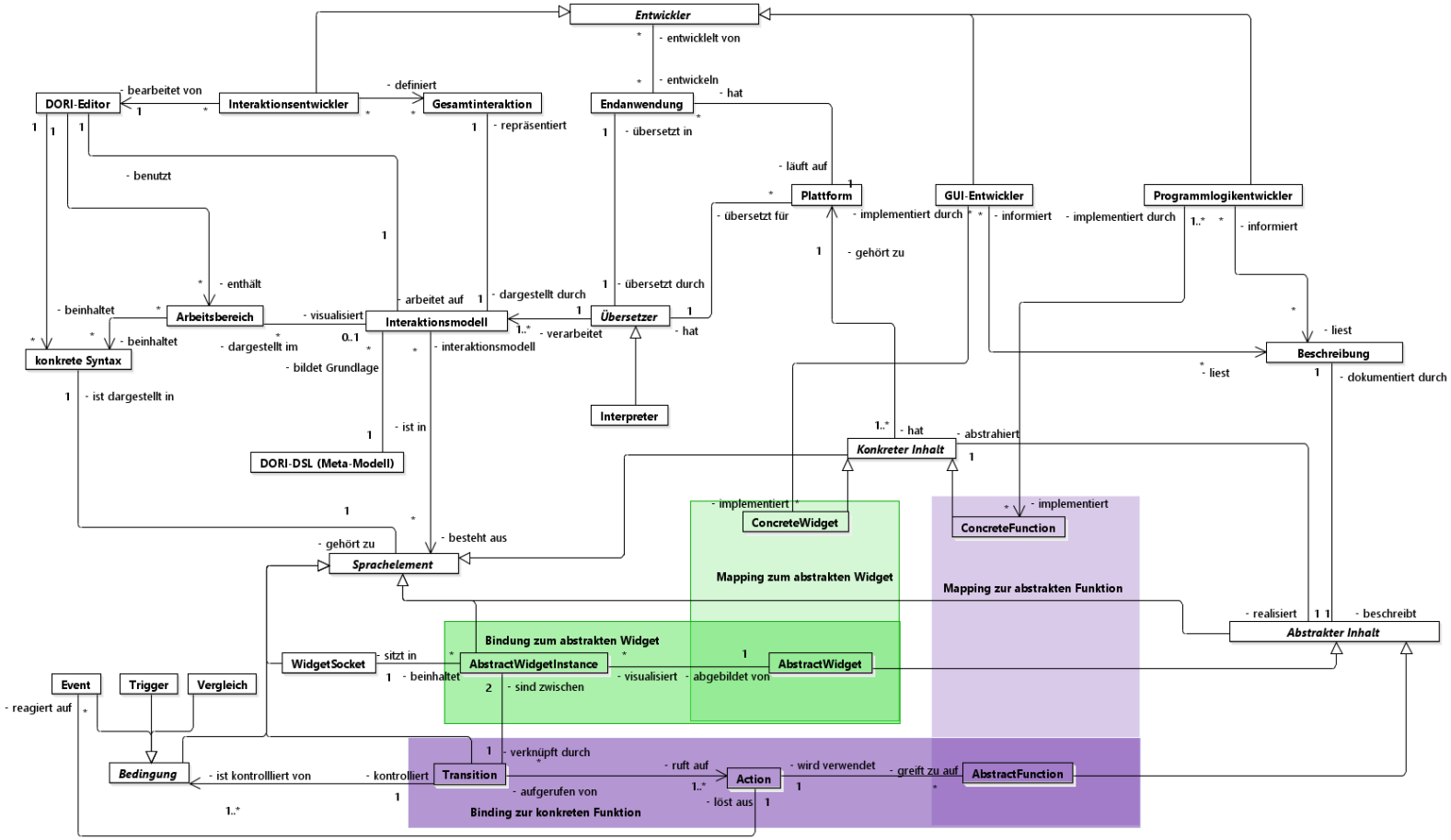


Abb. 3.1.: Domänenmodell

4. Beschreibung der Ziele und Anforderungen

Lisa, Thomas

Die folgenden Abschnitte dienen dazu die in Kapitel 2 aufgeführten Ziele in Use Cases zu untergliedern. Im Abschnitt 4.2 wird Ziel 1: „Entwicklung eines plattformunabhängigen Konzeptes zur graphischen Darstellung einer Gesamtinteraktion und hierfür eine plattformunabhängige Beschreibungsmöglichkeit für Widgets und Funktionalitäten“ näher betrachtet, darauf folgt das Ziel 2: „Erstellung eines Modellierungswerkzeuges zur Eingabe einer Gesamtinteraktion“. In Abschnitt 4.4 wird Ziel 3: „Erstellung eines Werkzeuges zur Übersetzung der modellierten Gesamtinteraktion in mehrere Endanwendungen auf verschiedenen Plattformen“ untersucht.

Auf diese drei Ziele folgen die Anforderungen an das Gesamtsystem in Abschnitt 4.5. Diese sind nicht durch ein einzelnes Ziel motiviert, sondern sammeln die Anforderungen an das gesamte System. Abschließend wird Ziel 4: „Validierung des Konzeptes und der Werkzeuge anhand eines konkreten Anwendungsbeispiels“ aufgeführt. Zu jedem Ziel werden passende Use Cases aufgestellt und falls notwendig mit Aktivitätsdiagrammen verfeinert. Direkt nach der Aufführung eines Aktivitätsdiagramms werden die Anforderungen formuliert, die sich aus dem Use Case ableiten lassen. Anforderungen, die sich nicht direkt aus Use Cases ableiten lassen, befinden sich am Ende des jeweiligen Abschnittes. Im Anhang A finden sich alle Anforderungen in einer Liste zusammengefasst. Da der zu erstellende Editor die DORI-DSL verwendet, sind Ziel 1 und Ziel 2 eng miteinander verknüpft. Daher wird im Abschnitt DORI-DSL der Fokus auf die Verwendung der Sprache gelegt. Die technischen Aspekte folgen dann im Abschnitt DORI-Editor.

Die abgeleiteten Anforderungen verwenden die Begrifflichkeiten aus dem Domänenmodell in Kapitel 3. Außerdem haben einige Anforderungen auch noch eine weitere Beschreibung, wenn sich die Anforderung nicht aus dem Domänenmodell erklärt.

4.1. Erläuterung der Ziel- und Use Case-Schablonen

Lisa, Thomas

In folgenden werden die einzelnen Zeilen der Ziel- sowie Use Case-Schablonen erläutert. Die Schablonen sind angelehnt an [5] und bilden im gesamten Kapitel „Beschreibung der Ziele und Anforderungen“ die Grundlage für die Ziele und Use Cases.

Tab. 4.1.: Ziel: Muster

Ziel	Was soll erreicht werden?
Betroffene	Welche Betroffenen sind in das Ziel involviert? Ein Ziel ohne Betroffene macht kein Sinn.
Auswirkung	Welche Veränderungen werden für die Betroffenen erwartet?
Randbedingung	Welche unveränderlichen Randbedingungen müssen bei der Zielerreichung beachtet werden?
Abhängigkeiten	Hängt die Erfüllung dieses Ziels mit der Erfüllung anderer Ziele zusammen? Dies kann einen positiven Effekt haben, indem die Erfüllung von Anforderungen zur Erreichung mehrerer Ziele beiträgt. Es ist aber auch möglich, dass ein Kompromiss gefunden werden muss, da Ziele unterschiedliche Schwerpunkte haben.

Tab. 4.2.: Use Case: Muster

Name	Kurze prägnante Beschreibung, meist aus Nomen und Verb
Nummer	Eindeutige Nummer zur Verwaltung, sollte von einer gesetzten Entwicklungsumgebung vergeben werden
beteiligte Aktoren	Welche Aktoren (Stakeholder) sind beteiligt, wer stößt den Use Case an?
Vorbedingungen	Was muss erfüllt sein, damit der Use Case starten kann?
Nachbedingungen	Wie sieht das mögliche Ergebnis aus? Im nächsten Schritt sind auch die Ergebnisse alternativer Aktivitäten zu berücksichtigen
betrachtete Aktivitäten	Welche einzelnen Schritte werden im Use Case durchlaufen, dabei wird nur der gewünschte typische Ablauf dokumentiert
alternative Aktivitäten	Welche Alternativen existieren zu den betrachteten Aktivitäten?
Kritikalität	Wie wichtig ist die Funktionalität im Gesamtsystem? Mögliche Werte sind hier: sehr hoch, hoch, mittel, niedrig, sehr niedrig
Verknüpfungen	Zusammenhang zu anderen Use Cases

4.2. DORI-DSL

Lisa, Stephan, Christopher, Hannah

An dieser Stelle wird das erste Ziel betrachtet: „Entwicklung eines plattformunabhängigen Konzeptes zur graphischen Darstellung einer Gesamtinteraktion und hierfür eine plattformunabhängige Beschreibungsmöglichkeit für Widgets und Funktionalitäten“. Dieses Konzept wird im folgenden DORI-DSL genannt. Auf Basis des Ziels in 4.3 konnten Use Cases identifiziert werden, welche im Folgenden in den Schablonen aufgeführt sind.

Tab. 4.3.: Ziel 1: Modellierung der Gesamtinteraktion

Ziel	Die DORI-DSL muss die Darstellung und Beschreibung einer Gesamtinteraktion ermöglichen.
Betroffene	Interaktionsentwickler
Auswirkungen	Interaktionsentwickler: Die Gesamtinteraktion kann modelliert werden.
Randbedingungen	Die DORI-DSL soll sich von der Aufmachung her an anderen graphischen Modellierungssprachen orientieren.
Abhängigkeiten	Die DORI-DSL bildet die vollständige Grundlage für die Beschreibung der Gesamtinteraktion im Editor.

Der folgende Use Case beschreibt die nötigen Schritte zur Definition und Verwendung einer abstrahierten GUI-Komponente.

Tab. 4.4.: Use Case: Definition und Verwendung einer abstrahierten GUI-Komponente

Name	Definition und Verwendung einer abstrahierten GUI-Komponente
Nummer	DSL-AGK
beteiligte Aktoren	Interaktionsentwickler
Vorbedingungen	Der Interaktionsentwickler hat eine Vorstellung von der abstrahierten GUI-Komponente.
Nachbedingungen	Eine abstrahierte GUI-Komponente ist definiert und kann im Modell verwendet werden.
typische Aktivitäten	1. Definition der Anzahl der Subkomponenten 2. Definition von Daten 3. Definition von Events
alternative Aktivitäten	keine Alternativen
Kritikalität	sehr hoch
Verknüpfungen	• Use Case: Modellierung von Datenflüssen

Anforderungen

- **DSL-AGK-1:** Die DORI-DSL muss ein Konstrukt bereitstellen, um eine GUI-Komponente zu abstrahieren. Dieses Konzept wird im Folgenden „Abstraktes Widget“ genannt.
- **DSL-AGK-2:** Einem abstrakten Widget muss eine Beschreibung zugeordnet werden können, aus der der Zweck der abstrahierten GUI-Komponente hervorgeht.
- **DSL-AGK-3:** GUI-Komponenten setzen sich gegebenenfalls aus weiteren GUI-Komponenten zusammen. Ein abstraktes Widget muss daher beschreiben können, wie viele weitere abstrakte Widgets es enthält.

- **DSL-AGK-4:** In einem abstrakten Widget müssen die zur abstrahierten GUI-Komponente gehörigen und für die Interaktionsbeschreibung relevanten Daten beschrieben werden können.
- **DSL-AGK-5:** In einem abstrakten Widget müssen die Events beschrieben werden können, welche die abstrahierte GUI-Komponente emmittieren kann und welche für die Interaktionsbeschreibung relevant sind.
- **DSL-AGK-6:** Die DORI-DSL muss ein Konstrukt bereitstellen, um eine abstrahierte GUI-Komponente im Kontext der Interaktionsbeschreibung einzusetzen. Dieses Konstrukt wird im Folgenden „Instanz eines abstrakten Widgets“ genannt.
- **DSL-AGK-7:** Die DORI-DSL muss darstellen können, welche Daten und Events die Instanz eines abstrakten Widgets verwalten können muss.
- **DSL-AGK-8:** Eine Instanz eines abstrakten Widgets muss weitere Instanzen von abstrakten Widgets beinhalten können.

Der folgende Use Case beschreibt die nötigen Schritte zur Definition von abstrahierter Programmlogik.

Tab. 4.5.: Use Case: Definition von abstrahierter Programmlogik

Name	Definition von abstrahierter Programmlogik.
Nummer	DSL-APL
beteiligte Aktoren	Interaktionsentwickler
Vorbedingungen	Der Interaktionsentwickler hat eine Vorstellung von der abstrahierten Programmlogik.
Nachbedingungen	Abstrahierte Programmlogik ist definiert.
typische Aktivitäten	1. Definition des Namens, der Parameter und des Rückgabetyps
alternative Aktivitäten	Keine Alternative
Kritikalität	sehr hoch
Verknüpfungen	• Use Case: Modellierung von Datenflüssen

Anforderungen

- **DSL-APL-1:** Die DORI-DSL muss ein Konstrukt bereitstellen, um Programmlogik zu abstrahieren. Dieses Konzept wird im Folgenden „Abstrakte Funktion“ genannt.
- **DSL-APL-2:** Einer abstrakten Funktion muss eine Beschreibung zugeordnet werden können, aus der die Funktionalität der abstrahierten Programmlogik hervorgeht.
- **DSL-APL-3:** Die DORI-DSL muss ein Konstrukt bereitstellen, um abstrahierte Programmlogik im Kontext der Interaktionsbeschreibung einzusetzen.

Für eine Interaktion muss der Interaktionsentwickler beschreiben können, anhand welcher Bedingungen von einer abstrahierte GUI-Komponente in die nächste gewechselt wird. Weiterhin wird definiert, welche Funktionen der Programmlogik aufgerufen werden. Der folgenden Use Case bildet dies ab.

Tab. 4.6.: Use Case: Modellierung des Wechsels zwischen GUI-Komponenten

Name	Modellierung des Wechsels zwischen GUI-Komponenten.
Nummer	DSL-WGK
beteiligte Aktoren	Interaktionsentwickler
Vorbedingungen	Der Interaktionsentwickler hat eine Vorstellung von der Gesamtinteraktion. Es existieren abstrahierte GUI-Komponenten.
Nachbedingungen	Zwei Interaktionszustände sind mittels Transition verknüpft.
typischer Ablauf	1. Der Nutzer verbindet die abstrahierten GUI-Komponenten. 2. Der Nutzer versieht den Zustandsübergang mit einer Bedingung.
alternative Abläufe	Der Nutzer verknüpft den Zustandsübergang mit einer abstrakten Funktion. Der Nutzer hebt die Verknüpfung auf. Der Nutzer ändert die Bedingung. Der Nutzer löscht den Zustandsübergang.
Kritikalität	sehr hoch
Verknüpfungen	• Use Case: Modellierung von Datenflüssen

Anforderungen

- **DSL-WGK-1:** Die DORI-DSL muss gerichtete Transitionen enthalten.
- **DSL-WGK-2:** Die DORI-DSL muss ein Konstrukt bereitstellen, um das Nehmen von Zustandsübergängen zu beschränken.
- **DSL-WGK-3:** Die DORI-DSL muss ein Konstrukt bereitstellen, um während eines Zustandsübergangs Programmlogik auszuführen.
- **DSL-WGK-4:** Die DORI-DSL muss ein Konzept bereitstellen, um das Eintreten von Ereignissen zu beschreiben. Dieses Konzept wird im Folgenden „Events“ genannt.

Der Interaktionsentwickler muss definieren können, welche Datenflüsse er in seiner Endanwendung realisieren möchte. Der folgende Use Case beschreibt die dafür benötigten Abläufe.

Tab. 4.7.: Use Case: Modellierung von Datenflüssen

Name	Modellierung von Datenflüssen
Nummer	DSL-DFL
beteiligte Aktoren	Interaktionsentwickler
Vorbedingungen	Der Interaktionsentwickler hat eine Vorstellung vom Datenfluss in der Gesamtinteraktion. Es sind wenigstens zwei Elemente der Sprache vorhanden.
Nachbedingungen	Der Datenfluss ist im Modell definiert.
typischer Ablauf	1. Anbinden eines Datums an ein anderes Datum.
alternative Abläufe	Der Nutzer bindet das Datum an mehrere Elemente der Sprache an. Der Nutzer entfernt Daten. Der Nutzer hebt die Bindung auf.
Kritikalität	sehr hoch, weil sowohl GUI-Komponenten, als auch Funktionsaufrufe Daten benötigen und erzeugen
Verknüpfungen	• Use Case: Definition und Verwendung einer abstrahierten GUI-Komponente • Use Case: Definition von abstrahierter Programmlogik • Use Case: Modellierung des Wechsels zwischen GUI-Komponenten

Anforderungen

- **DSL-DFL-1:** Die DORI-DSL muss ein Konstrukt bereitstellen, um Datenübergaben zwischen Instanzen abstrakter Widgets und abstrakten Funktionen zu beschreiben.

Auch wenn konkrete Inhalte nicht nötig sind um eine Gesamtinteraktion zu definieren, muss die DORI-DSL eine Möglichkeit bieten, um abstrakte mit konkreten Inhalten zu verknüpfen. Dies bildet eine Grundlage für die Funktionsweise des DORI-Editors. Der folgende Use Case beschreibt diese Verknüpfung.

Tab. 4.8.: Use Case: Verknüpfung von Abstraktion mit Implementierung

Name	Verknüpfung von Abstraktion mit Implementierung.
Nummer	DSL-VAI
beteiligte Aktoren	Interaktionsentwickler
Vorbedingungen	Eine Abstraktion ist definiert.
Nachbedingungen	Die Abstraktion ist mit der Implementierung verknüpft.
typischer Ablauf	1. Der Nutzer verknüpft eine Abstraktion mit der Implementierung.
alternative Abläufe	–
Kritikalität	sehr hoch
Verknüpfungen	• Use Case: Definition und Verwendung einer abstrahierten GUI-Komponente • Use Case: Definition von abstrahierter Programmlogik

Anforderungen

- **DSL-VAI-1:** Die DORI-DLS muss einer abstrahierten GUI-Komponente (Abstraktes Widget) eine konkrete Implementierung dieser GUI-Komponente zuordnen können.
- **DSL-VAI-2:** Die DORI-DLS muss einem abstrahierten Programmlogikaufruf (Abstrakte Funktion) eine konkrete Implementierung dieses Programmlogikaufrufs zuordnen können.

Sonstige Anforderungen

- **DSL-SON-1:** Die DORI-DSL muss plattformunabhängige Datentypen (im Folgenden „primitive Datentypen“) bereitstellen.
- **DSL-SON-2:** Die DORI-DSL muss die Definition von zusammengesetzten Datentypen ermöglichen.

- **DSL-SON-3:** Syntax und Semantik der DORI-DSL sind in der Projektdokumentation festgehalten.

4.3. DORI-Editor

Lisa, Nancy, Thomas, Stephan

An dieser Stelle wird das zweite Ziel betrachtet: „Erstellung eines Modellierungswerkzeuges zur Eingabe einer Gesamtinteraktion. Dieses Werkzeug wird im Folgenden als DORI-Editor bezeichnet.“ Daraus lässt sich das folgende allgemeine Ziel ableiten.

Tab. 4.9.: Ziel 2: Eingabe der Gesamtinteraktion

Ziel	Der DORI-Editor muss die Eingabe der Gesamtinteraktion auf Basis der DORI-DSL ermöglichen.
Betroffene	Interaktionsentwickler
Auswirkungen	Interaktionsentwickler: Die Gesamtinteraktion kann graphisch modelliert werden. Sie wird im DORI-Editor durch ein Modell dargestellt.
Randbedingungen	Der Editor soll den gängigen, noch festzulegenden, Designrichtlinien und Grundlagen zur Softwareergonomie entsprechen.
Abhängigkeiten	Das Modell und dessen graphische Darstellung basieren vollständig auf der DORI-DSL (4.3).

Das Ziel lässt sich in die nachfolgend beschriebenen Use Cases aufteilen.

Der nachfolgende Use Case beschreibt das Anlegen und Bearbeiten eines Modells durch den Interaktionsentwickler. Dafür kann er ein neues Modell anlegen oder ein bereits angelegtes Modell in den DORI-Editor laden.

Tab. 4.10.: Use Case: Modellierung der Gesamtinteraktion

Name	Modell bearbeiten
Nummer	E-MDG
beteiligte Aktoren	Interaktionsentwickler
Vorbedingungen	Der DORI-Editor ist gestartet.
Nachbedingungen	Der Nutzer hat ein geöffnetes Modell, welches seine Bearbeitung enthält.
betrachtete Aktivitäten	1. Nutzer wählt Funktionalität zum Erstellen eines neuen Modells 2. Nutzer bearbeitet das Modell, dies bedeutet der Nutzer fügt Elemente zum Modell hinzu, entfernt Elemente oder definiert Beziehungen zwischen Elementen gemäß der DORI-DSL. 3. Nutzer beendet die Bearbeitung
alternative Aktivitäten	Nutzer kann bestehendes Modell öffnen
Kritikalität	sehr hoch, dies ist die Kernfunktionalität des DORI-Editors
Verknüpfungen	

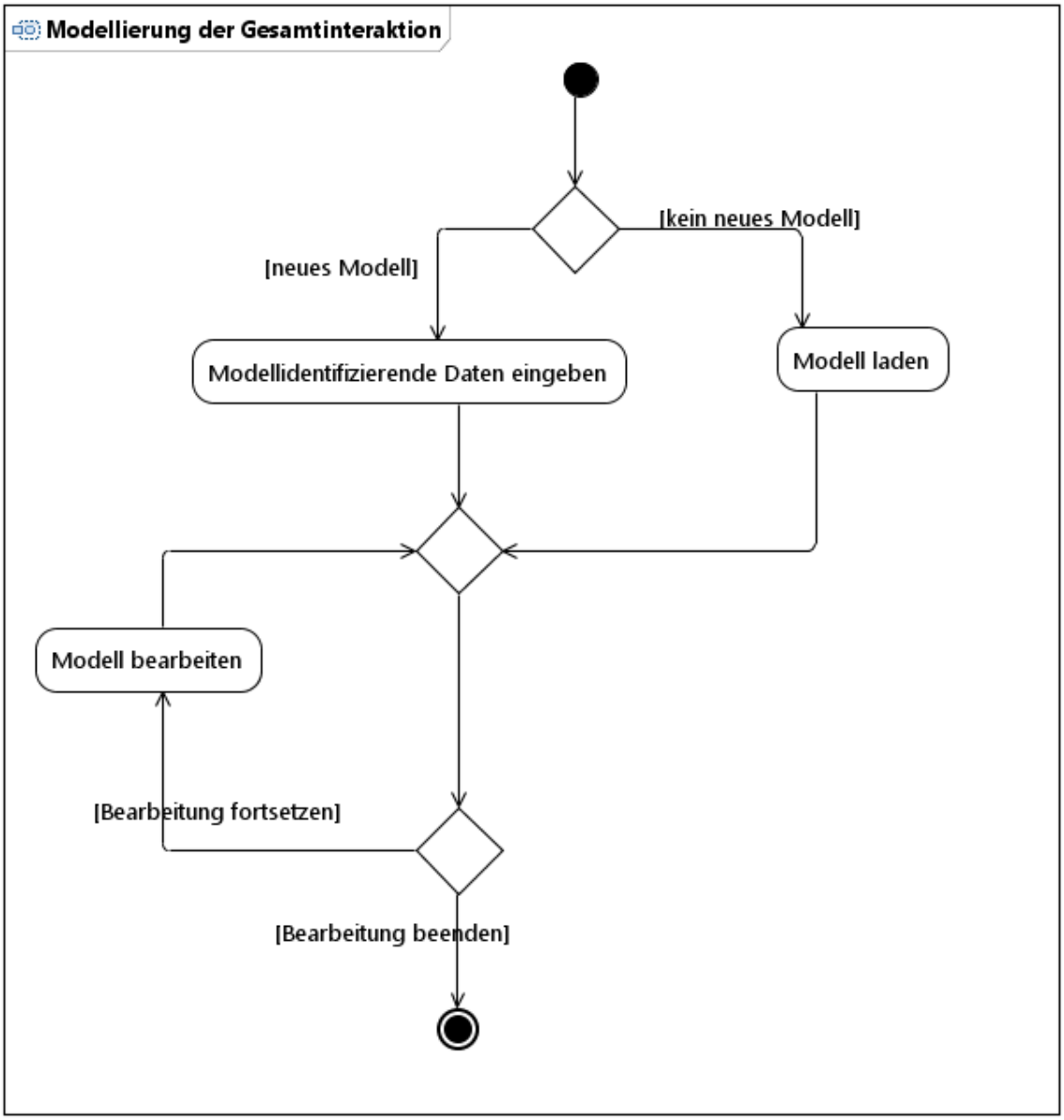


Abb. 4.1.: Aktivitätsdiagramm: Modellierung der Gesamtinteraktion.

Anforderungen

- **E-MDG-1:** Der DORI-Editor muss die Erstellung eines neuen Modells ermöglichen.
- **E-MDG-2:** Der DORI-Editor muss bereits erstellte Modelle öffnen können.
  Beschreibung: Da Modelle dafür gespeichert werden müssen besteht eine Verknüpfung zum Use Case E-MSP (siehe Tabelle 4.11

- **E-MDG-3:** Der DORI-Editor muss die Bearbeitung eines geöffneten Modells ermöglichen. Dies bedeutet der Nutzer fügt Elemente zum Modell hinzu, entfernt Elemente oder definiert Beziehungen zwischen Elementen, gemäß der DORI-DSL.

Der folgende Use Case beschreibt das Speichern eines Modells durch den Interaktionsentwickler. Er kann seine Arbeit direkt bevor er den DORI-Editor schließt speichern, aber auch zwischenzeitlich während er ein Modell bearbeitet. Wird der DORI-Editor vom Interaktionsentwickler geschlossen, so hat dieser die Möglichkeit seine Arbeit vor dem Schließen zu speichern oder das Modell ohne speichern zu schließen.

Tab. 4.11.: Use Case: Modell speichern

Name	Modell speichern
Nummer	E-MSP
beteiligte Aktoren	Interaktionsentwickler
Vorbedingungen	Im DORI-Editor ist ein Modell geöffnet.
Nachbedingungen	Die Änderungen am Modell werden vom System persistent gespeichert.
betrachtete Aktivitäten	1. Nutzer wählt Funktionalität zum Speichern des Modells 2. Nutzer schließt DORI-Editor
alternative Aktivitäten	Nutzer kann den DORI-Editor offen behalten, Nutzer schließt DORI-Editor ohne zu speichern
Kritikalität	sehr hoch, Nutzer muss seine Arbeit sichern können
Verknüpfungen	

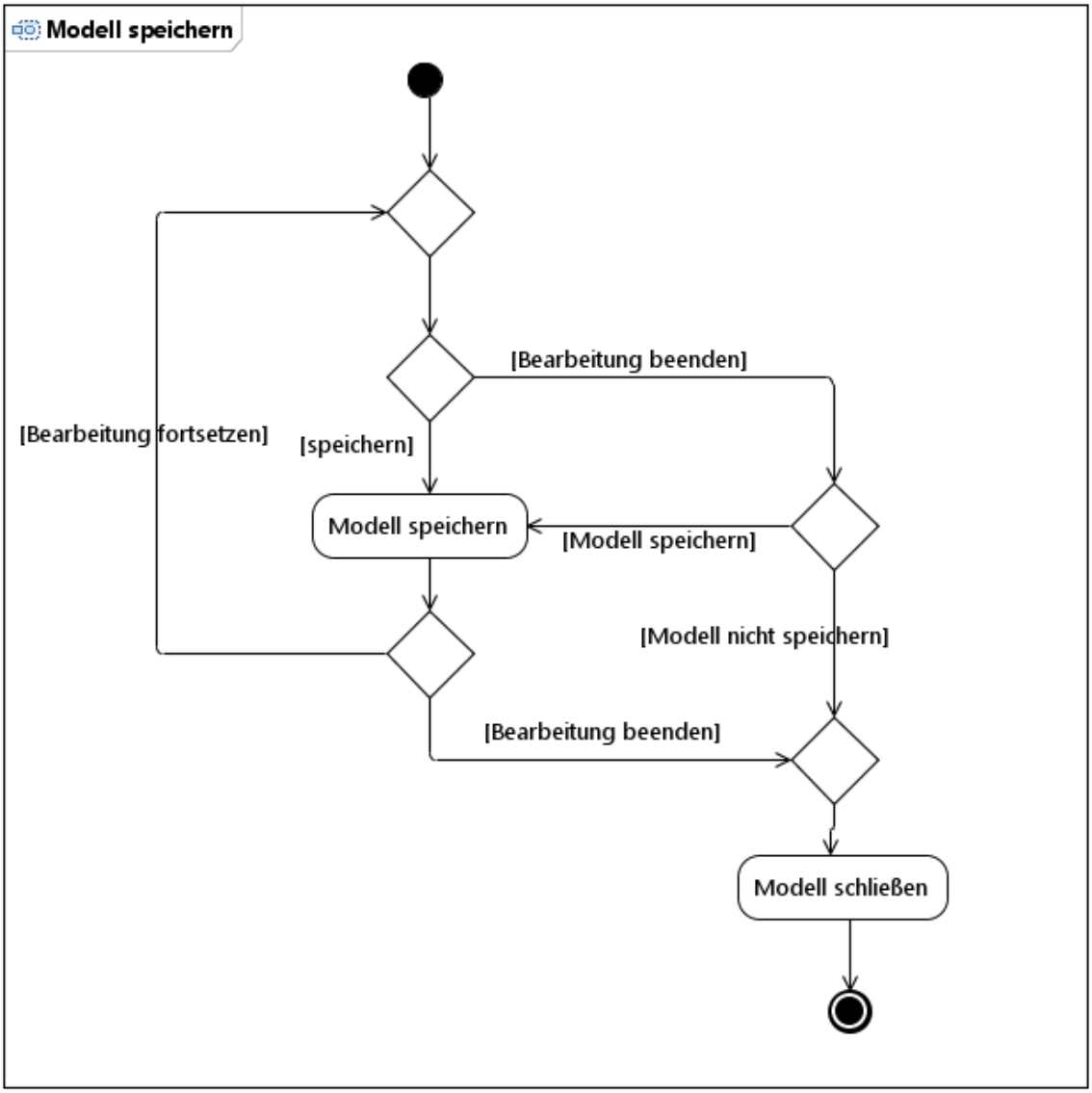


Abb. 4.2.: Aktivitätsdiagramm: Modell speichern

Anforderungen

- **E-MSP-1:** Der DORI-Editor muss ein geöffnetes Modell persistent speichern können. **Beschreibung:** Das Modell muss bei der Bearbeitung gespeichert werden können und auch beim Schließen.
- **E-MSP-2:** Der DORI-Editor muss beim Schließen dem Nutzer die Möglichkeit zum Speichern bieten.

- **E-MSP-3:** Der DORI-Editor muss das Schließen ohne Speichern zulassen.
- **E-MSP-4:** Der DORI-Editor muss die persistente Speicherung des Modells in einem versionierbaren Format ermöglichen.
  Quelle: interne Gruppengespräche
- **E-MSP-5:** Der DORI-Editor muss den Speicherstand des aktuell geöffneten Modells überwachen.
- **E-MSP-6:** Der DORI-Editor muss den Nutzer auf ungespeicherte Änderungen hinweisen.

Der folgende Use Case stellt die Verknüpfung zur DORI-DSL dar. Die detaillierten Use Cases finden sich daher im Abschnitt DORI-DSL.

Tab. 4.12.: Use Case: Gesamtinteraktion bearbeiten

Name	Gesamtinteraktion bearbeiten
Nummer	E-GIB
beteiligte Aktoren	Interaktionsentwickler
Vorbedingungen	Im DORI-Editor ist ein Modell geöffnet.
Nachbedingungen	Der Nutzer hat ein geöffnetes Modell, welches seine Bearbeitung enthält.
betrachtete Aktivitäten	1. Nutzer bearbeitet Modell
alternative Aktivitäten	An dieser Stelle finden sich keine alternativen Abläufe, da die Verfeinerungen in den verknüpften Use Cases abgedeckt sind.
Kritikalität	sehr hoch, den dies ist eine Kernfunktionalität des DORI-Editor
Verknüpfungen	Dieser Use Case ist zu den Use Cases der DORI-DSL verknüpft: • Use Case: Definition und Verwendung einer abstrahierten GUI-Komponente • Use Case: Definition von abstrahierter Programmlogik • Use Case: Modellierung des Wechsels zwischen GUI-Komponenten • Use Case: Modellierung von Datenflüssen • Use Case: Verknüpfung von Abstraktion mit Implementierung

Anforderungen

- **E-GIB-1:** Der DORI-Editor muss die abstrakte Syntax der DORI-DSL einhalten.
- **E-GIB-2:** Der DORI-Editor muss die konkrete Syntax der DORI-DSL darstellen können.
  Beschreibung: Die grafische Darstellung des Modells auf Basis der konkreten Sytanx kann beispielsweise in einem Arbeitsbereich erfolgen. Der Zugriff auf die Elemente der

DORI-DSL kann beispielsweise über eine Toolbox erfolgen. Die konkrete Syntax wird von der DORI-DSL bereitgestellt und in den Editor eingebunden.

- **E-GIB-3:** Der DORI-Editor muss die Bearbeitung eines Modelles auf Basis der DORI-DSL ermöglichen.
  Beschreibung: Der DORI-Editor muss die Bearbeitung von Modellen auf Grundlage aller bereitgestellten Sprachmittel der DORI-DSL ermöglichen.

Der folgende Use Case beschreibt die Prüfung der Konsistenz eines Modells. Diese Funktionalität kann vom Interaktionsentwickler verwendet werden um ein Modell zu prüfen. Im Anschluss an die Prüfung wird dem Interaktionsentwickler das Ergebnis dieser angezeigt.

Tab. 4.13.: Use Case: Konsistenz des Modells prüfen

Name	Konsistenz des Modells prüfen
Nummer	E-KMP
beteiligte Aktoren	Interaktionsentwickler
Vorbedingungen	Im DORI-Editor ist ein Modell geöffnet, welches mindestens ein Element enthält.
Nachbedingungen	Der Nutzer erhält eine Übersicht über die Fehler oder Fehlerfreiheit des Modells.
betrachtete Aktivitäten	1. Nutzer wählt Funktionalität zum Prüfen der Konsistenz des Modells
alternative Aktivitäten	das Modell hat keine Fehler
Kritikalität	niedrig, ohne die Funktion kann der Nutzer seine Arbeit lediglich händisch prüfen
Verknüpfungen	Einschränkungen der Sprache (constraints)

Anforderungen

- **E-KMP-1:** Der DORI-Editor muss dem Nutzer eine Funktionalität zum Prüfen der Konsistenz eines Modells bereitstellen.
- **E-KMP-2:** Die Konsistenzprüfung muss auf Basis der DORI-DSL das Modell prüfen.

- **E-KMP-3:** Die Konsistenzprüfung muss dem Nutzer eine Übersicht über die Fehler des geprüften Modells bereitstellen können.
- **E-KMP-4:** Die Konsistenzprüfung muss dem Nutzer eine Übersicht über die Fehlerfreiheit des geprüften Modells bereitstellen können.

Der folgende Use Case beschreibt die Möglichkeit, dass der Interaktionsentwickler eine modellierte Gesamtinteraktion an den Übersetzer übergeben kann. An dieser Stelle ist der reine Datenaustausch zwischen DORI-Editor und Übersetzer im Fokus. Der Vorgang des Übersetzens ist im Abschnitt 4.4 näher beschrieben.

Tab. 4.14.: Use Case: Modell an Übersetzer übergeben

Name	Modell an Übersetzer übergeben
Nummer	E-MUU
beteiligte Aktoren	Interaktionsentwickler
Vorbedingungen	Im DORI-Editor ist eine Gesamtinteraktion modelliert.
Nachbedingungen	Der Nutzer kann den Übersetzer starten.
betrachtete Aktivitäten	1. Der Nutzer übergibt die Modelldatei an den Übersetzer
alternative Aktivitäten	
Kritikalität	sehr hoch, ohne die Funktion kann der Übersetzer nicht starten
Verknüpfungen	Abschnitt 4.4 Übersetzer

Anforderungen

- **E-MUU-1:** Der DORI-Editor muss die modellierte Gesamtinteraktion an den Übersetzer übergeben können.
- **E-MUU-2:** Der DORI-Editor und der Übersetzer müssen eine gemeinsame Schnittstelle zum Übergeben des Modells haben.

Sonstige Anforderungen

- **E-SON-1:** Der DORI-Editor muss die gängigen Designrichtlinien einhalten.
  Beschreibung: Die Festlegung der verwendeten Richtlinien erfolgt später im Projektverlauf und hängt von der Festlegung des Editor-Tools ab.
  Quelle: interne Gruppengespräche
- **E-SON-2:** Der DORI-Editor muss die gängigen Richtlinien der Software-Ergonomie einhalten.
  Quelle: interne Gruppengespräche
- **E-SON-3:** Der DORI-Editor muss eine Funktionalität zum Erstellen eines Berichtes bieten, welcher die Beschreibungen der abstrakten Inhalte beinhaltet.
  Beschreibung: Der Bericht dient dazu dem GUI- und Programmlogikentwickler eine Übersicht über zu implementierende Widgets und Funktionen zu bieten.
  Quelle: interne Gruppengespräche
- **E-SON-4:** Zum DORI-Editor muss es ein Installations- und Benutzerhandbuch geben.
  Beschreibung: Das Installationshandbuch muss es dem Nutzer ermöglichen die Software zu installieren. Das Benutzerhandbuch muss den Umgang mit dem DORI-Editor für den Nutzer erläutern.
  Quelle: Anforderung der Betreuer

4.4. Übersetzer

Lisa, Felix, Thomas, Stephan

An dieser Stelle wird das dritte Ziel „Erstellung eines Werkzeuges zur Übersetzung der modellierten Gesamtinteraktion in mehrere Endanwendungen auf verschiedenen Plattformen. Dieses Werkzeug wird im Folgenden Übersetzer genannt“ beschrieben. Daraus lässt sich das folgende allgemeine Ziel ableiten.

Tab. 4.15.: Ziel 3: Übersetzung des Interaktionsmodells

Ziel	Der Übersetzer muss die Übersetzung des Interaktionsmodells in mehrere Endanwendungen auf verschiedenen Plattformen ermöglichen.
Betroffene	Interaktionsentwickler, Endnutzer
Auswirkungen	Interaktionsentwickler: Die modellierte Gesamtinteraktion kann in eine Endanwendung überführt werden. Endutzer: Die Endanwendung kann verwendet werden.
Randbedingungen	
Abhängigkeiten	Die Gesamtinteraktion wurde im DORI-Editor modelliert (siehe Abschnitt 4.9).

Der Übersetzer ist an dieser Stelle noch nicht auf eine verwendete Technik festgelegt. Daher gibt es zwei Hauptunterscheidungen innerhalb des Ziels 3: Der Übersetzer kann als Generator oder als Interpreter realisiert werden.

Tab. 4.16.: Use Case: Interaktionsmodell in Endanwendung übersetzen

Name	Interaktionsmodell in Endanwendung übersetzen
Nummer	U-IEU
beteiligte Aktoren	Interaktionsentwickler
Vorbedingungen	vollständiges Interaktionsmodell im DORI-Editor vorhanden
Nachbedingungen	lauffähige Endanwendung auf Zielplattform
typische Abläufe	betrachtete Aktivitäten Interpreter: 1. Plattform auswählen 2. Interpreter auf Zielgerät installieren 3. Modell und zugehörige Dateien aus dem Editor exportieren 4. Modell und Dateien in den Interpreter laden betrachtete Aktivitäten Generator 1. Plattform auswählen 2. Modell und zugehörige Dateien, aus dem Editor, exportieren 3. Quellcode-Dateien generieren 4. Dateien zu Endanwendung kompilieren 5. Endanwendung auf Zielgerät installieren
alternative Aktivitäten	Alternative im Generatorablauf: bei Skriptsprachen kein Kompilieren notwendig
Kritikalität	sehr hoch, da ohne diese Funktionalität keine Endanwendungen erstellt werden können
Verknüpfungen	• mit DORI-Editor verknüpft, da in diesem das Interaktionsmodell erstellt wird (siehe Abschnitt 4.3) • mit DORI-DSL verknüpft, da die Übersetzung konform zur DORI-DSL sein muss (siehe Abschnitt 4.2)

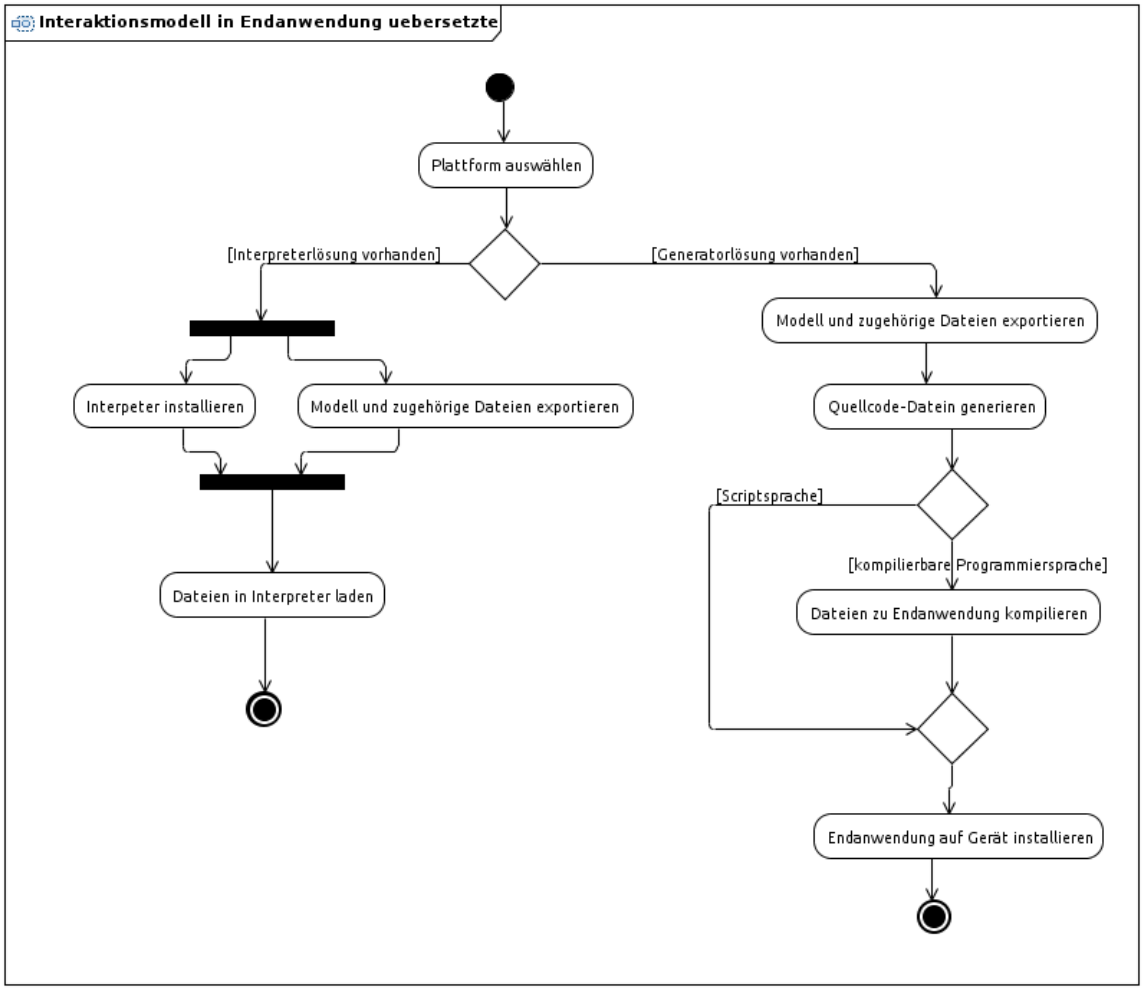


Abb. 4.3.: Aktivitätsdiagramm: Interaktionsmodell in Endanwendung übersetzen

Anforderungen

- **U-IEU-1:** Ein Interaktionsmodell muss in eine Endanwendung auf mehreren Plattformen übersetzbar sein.
- **U-IEU-2:** Der Übersetzer muss die Auswahl der Zielplattform ermöglichen.
  Beschreibung: Es ist denkbar, dass für jede Plattform ein neuer Übersetzer geschrieben wird. Diese Übersetzer können entweder in einem alleinstehenden Programm oder durch den DORI-Editor als Übersetzer-Sammlung gekapselt werden.
  Quelle: interne Gruppengespräche
- **U-IEU-3:** Der Übersetzer muss die Endanwendung in Konformität zur DORI-DSL übersetzen.

- **U-IEU-4:** Der Interpreter muss die Endanwendung auf der Zielplattform ausführen.
- **U-IEU-5:** Der Generator muss die Endanwendung für die Zielplattform übersetzen.
- **U-IEU-6:** Der Übersetzvorgang muss dokumentiert sein.
  Beschreibung: Bevor das Übersetzen für eine Zielplattform durchgeführt werden kann müssen eventuell zusätzliche Dateien oder Programme installiert werden. Die Installation der zusätzlichen Software muss daher in einem Dokument beschrieben werden.
- **U-IEU-7:** Der Übersetzer muss dokumentiert werden.

4.5. Gesamtsystem

Lisa, Thomas, Stephan

Im Folgenden werden die Use Cases und Anforderungen an das Gesamtsystem beschrieben.

Der nachfolgende Use Case beschreibt die Möglichkeit eines Entwicklern inner- und außerhalb der Projektgruppe das Gesamtsystem zu erweitern. In diese Zusammenhang liegt der Fokus auf den Abhängigkeiten der Subsysteme untereinander.

Tab. 4.17.: Use Case: Gesamtsystem erweitern

Name	Gesamtsystem erweitern
Nummer	GS-ERW
beteiligte Aktoren	Entwickler (alle zukünftigen Entwickler, die die Software oder DORI-DSL erweitern wollen)
Vorbedingungen	Das Gesamtsystem existiert.
Nachbedingungen	Der Entwickler hat das Gesamtsystem erweitert.
betrachtete Aktivitäten	1. DORI-DSL erweitern und die Dokumentation dazu anpassen 2. DORI-Editor erweitern und Dokumentation dazu anpassen 3. Übersetzer erweitern und Dokumentation dazu anpassen
alternative Aktivitäten	nur den Übersetzer erweitern und die Dokumentation dazu anpassen, nur den DORI-Editor anpassen und die Dokumentation dazu anpassen, DORI-Editor und Übersetzer erweitern und die Dokumentation dazu anpassen
Kritikalität	mittel, das Gesamtsystem funktioniert auch ohne diesen Ablauf
Verknüpfungen	

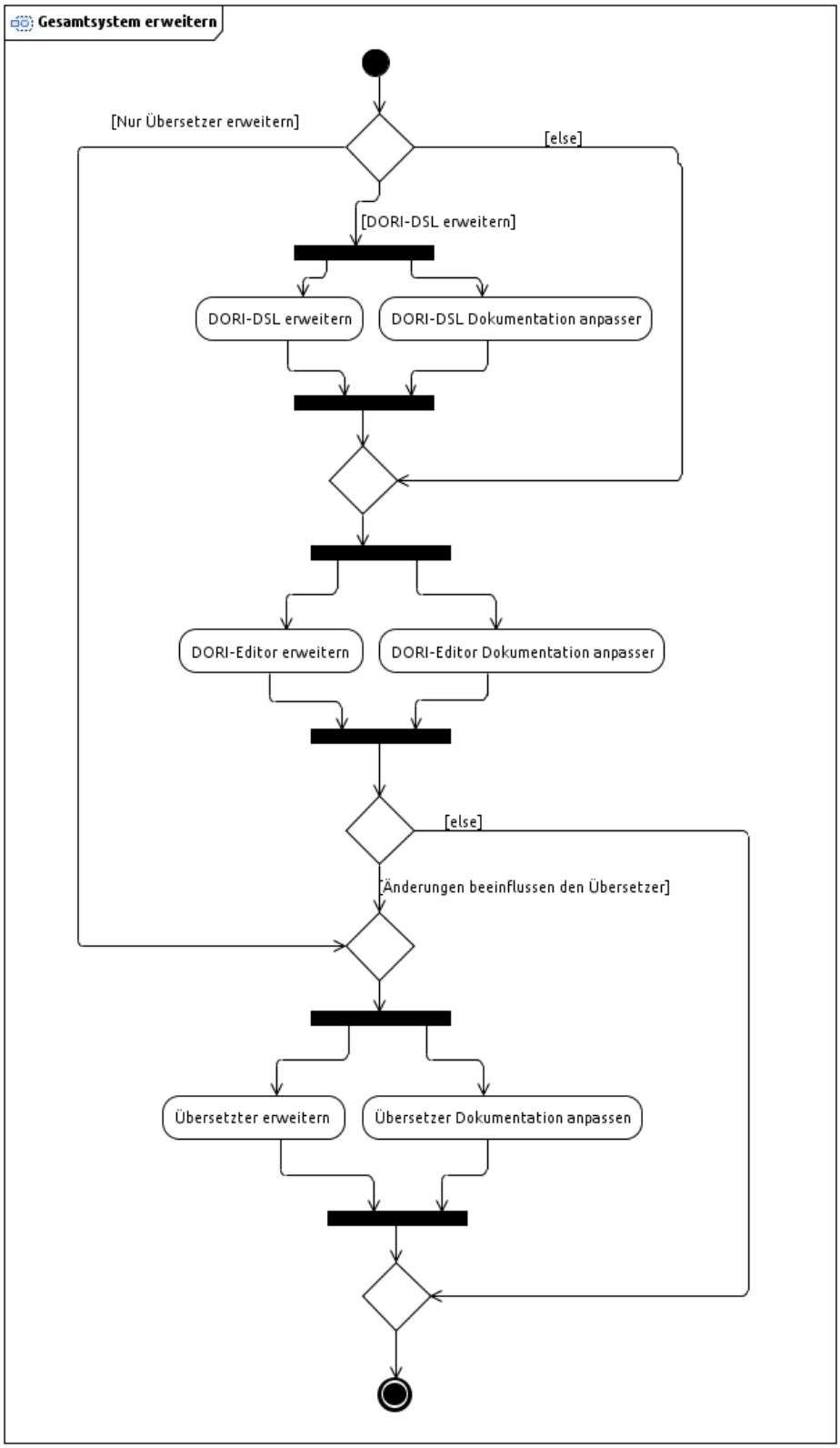


Abb. 4.4.: Aktivitätsdiagramm: Gesamtsystem erweitern

Anforderungen

- **GS-ERW-1:** Das Gesamtsystem muss erweiterbar sein.
  Beschreibung: Der Entwicklungsprozess in der Projektgruppe muss stets ein Augenmerk auf die Erweiterbarkeit des Systems legen. Das bedeutet, dass zum Beispiel Software modular aufgebaut sein muss und stets ausreichend Dokumentation vorliegt. In dieser Dokumentation muss ein besonderer Fokus auf Beschreibungen zu Auswirkungen von Änderungen auf andere Systembestandteile gelegt werden.
  Quelle: Interview mit den Betreuern
- **GS-ERW-2:** Es muss eine Architekturdokumentation mitgeliefert werden.

4.6. Validierung

Lisa, Jan (ehemaliges Gruppenmitglied), Thomas, Stephan

An dieser Stelle wird das vierte Ziel „Validierung des Konzeptes und der Werkzeuge anhand eines konkreten Anwendungsbeispiels“ beschrieben. Dies beinhaltet die konkrete Implementierung dieser Software mithilfe des DORI-Editor und des Übersetzers. Daraus lässt sich das folgende allgemeine Ziel ableiten.

Tab. 4.18.: Ziel 4: Validierung des Gesamtsystems

Ziel	Das Gesamtsystem, bestehend aus DORI-DSL, DORI-Editor und Übersetzer, muss validiert werden.
Betroffene	Teilnehmer der Projektgruppe, Betreuer
Auswirkungen	Teilnehmer der Projektgruppe: Tauglichkeit des Konzepts validieren. Betreuer: Bewertung des Konzeptes und der Arbeit der Projektgruppe
Randbedingungen	Das Gesamtsystem muss vollständig implementiert sein.
Abhängigkeiten	Ziele 1, 2 und 3 müssen erfüllt sein.

Das Arbeitsergebnis der Projektgruppe soll anhand eines konkreten Anwendungsbeispiels am Ende validiert werden. Hierzu muss zuerst die DORI-DSL entwickelt werden. Auf ihrer Basis arbeitet der DORI-Editor, damit in ihm die Gesamtinteraktion als Interaktionsmodell modelliert werden kann. Das Interaktionsmodell kann mit Hilfe des Übersetzers in eine Endanwendung auf einer Zielplattform überführt werden.

Somit sind die abzugebenden Ergebnisse neben der DORI-DSL, dem DORI-Editor und dem Übersetzer die Interaktionsmodelle und die damit beschriebenen Endanwendungen für die unterschiedlichen Plattformen.

Zur Validierung stehen der Projektgruppe verschiedene Beispielszenarien zur Verfügung. Hierzu könnte etwa das Projekt NEMo (Verbesserung der Mobilität im ländlichem Raum) oder eine selbst definierte Anwendung gehören.

Anforderungen

- **V-GSV-1:** Das Gesamtsystem muss anhand eines Anwendungsbeispiels validiert werden.
- **V-GSV-2:** Die Gesamtinteraktion des Anwendungsbeispiels muss mit dem DORI-Editor modelliert werden können.
- **V-GSV-3:** Das Anwendungsbeispiel muss mit dem Übersetzer in mindestens zwei Endanwendungen auf unterschiedlichen Plattformen übersetzt werden
- **V-GSV-4:** Das Anwendungsbeispiel muss hinreichend komplex sein.
  Beschreibung: Wenn eine gewisse Komplexität nicht gegeben ist, dann findet keine Überprüfung statt, ob das Gesamtsystem für die Software-Entwicklung geeignet ist. Beispiele für komplexe Endanwendungskomponenten sind: Client-Server Kommunikation, Navigationsleisten, geschachtelte Widgets, dynamisch reagierende Eingabefelder. Der Endnutzer muss multiple Interaktionen vornehmen können.
- **V-GSV-5:** Das Interaktionsmodell des Anwendungsbeispiels muss alle Elemente der DORI-DSL verwenden.
  Beschreibung: Das Interaktionsmodell muss die DORI-DSL ausreizen, um eine Validierung zu ermöglichen.

5. Anforderungserhebung

Nancy

Ziel des Anforderungsdokuments ist eine überprüfbare Liste von Anforderungen, gesammlet in Anhang A, die dazu dient den Funktionsumfang des von der Projektgruppe DORI erstellten Systems zu definieren und validieren. Die hier folgenden Informationen beschreiben die Quellen die der Projektgruppe bei der Anforderungserhebung zugrunde lagen.

In der Anfangsphase der Projektgruppe wurden Interviews mit den Betreuern geführt, damit eine einheitliche Meinung des Projektziels erlangt werden konnte. Weiterhin wurde Timo Schlömer befragt. Seine Masterarbeit bietet eine Machbarkeitsstudie für das Projekt. Weitere Informationen zu seiner Masterarbeit sind in dem Abschnitt 5.1 zu finden.

Die Informationen, die aus den Interviews erlangt wurden, sind in das Visionsdokument eingeflossen, welches die erste Vision der Projektziele sowie die ersten Lösungsideen für diese darlegt. Dies schafft eine gemeinsame Wissensgrundlage für die Ziele der Projektgruppe.

Teile des Visionsdokument sind Grundlage für die Kapitel 2 (Motivation) und Kapitel 4 (Beschreibung der Ziele).

Bei weiteren Recherchen zu DSLs, die eine Interaktion darstellen können ist die Projektgruppe auf die Sprache IFML (Interactionflow Modelling Language) gestoßen. Die Informationen zu dieser Sprache sind im Abschnitt 5.2.1 zu finden und bieten eine Quelle für die Anforderungen der DORI-DSL.

Um eine Vorstellung für die Umsetzung von interaktiven Anwendungen zu bekommen, wurden zwei verschiedenen Prototypen eines Taschenrechners erstellt. Die Abgrenzung der beiden Prototypen zeigt die Unterschiede zwischen den plattformspezifischen Umsetzungen. Weitere Informationen zu diesen Prototypen sind im Abschnitt 5.2 zu finden.

Um einen Überblick über den Funktionsumfang des Editors und des Übersetzers zu bekommen wurde auch für diesen ein Prototyp erstellt. Dieser findet sich im Abschnitt 5.2.4.

Um detaillierte Anforderungen erheben zu können, wurden verschiedene Use Cases definiert, die bereits in Abschnitt 4.2 zu vorgestellt wurden.

5.1. Masterarbeit von Timo Schlömer

Hauke, Stephan, Lisa

Als Machbarkeitsstudie für diese PG kann die Masterarbeit von Timo Schlömer mit dem Titel: „Modellgetriebene GUI-Erstellung für serviceorientierte Anwendungen“ [12] verwendet werden. Diese Arbeit wurde im Sommer 2017 an der Universität Oldenburg in der Abteilung Software Engineering angefertigt.

Ziel:

Zielsetzung der Masterarbeit war eine Möglichkeit zu erarbeiten mit welchem neuen Interaktionsmodelle in den SENSEI-Ansatz integriert werden können. Hierbei sollte mittels Zustandsautomaten gearbeitet werden, wobei jeder Zustand für eine Sicht steht, die ein Benutzer sehen und mit der er interagieren kann. Des Weiteren sollen Benutzeränderungen SENSEI-Services anstoßen und mit diesen Daten austauschen.

Herangehensweise:

Es wurde ein Metamodell in Anlehnung an das SENSEI-Metamodell entworfen. Hierbei werden Interaktionen und GUIs mittels Zustandsdiagrammen modelliert. Das Metamodell wird hierbei textuell umgesetzt, es existiert kein graphischer Moddellierungsansatz. Zentraler Modellierungsaspekt ist innerhalb des Interaktionsmodells das UserInteractionModel. Diese beinhaltet neben Verwaltungsinformationen wie Versionsnummer und Beschreibung, Namen, ebenfalls eine Menge an Widgets, eine Menge an WidgetTransitions, die verwendete Plattform, das initiale Widget und die initialen Daten für das jeweilige Modell. Das Modell wird in einer .json Datei gespeichert. Die Übertragung von Daten erfolgt mittels Daten-Ports, welche an Widgets gekoppelt sind, und Daten-Pipes, welche den Datentransfer während einer Transition abwickeln. Data-Ports lassen sich in Ein- und Ausgabe-Data-Ports unterschieden.

Implementierung:

Als Machbarkeitstudie wurde ein PHP-Interpreter implementiert, dessen Name Interpreter für Modellgetriebene GUI-Erstellung für serviceorientierte Anwendungen (kurz: MoGEsA) lautet. Der Interpreter ist hierbei Serverbasiert und funktioniert über einen REST Service. Hierbei erfolgt der Datenaustausch per .json- oder .xml-Datei.

5.2. Prototypen

Hannah, Thomas, Stephan

Als Sprache zum Darstellen des Inhalts, der Nutzerinteraktion und dem Kontrollverhalten des Frontends von Softwareanwendungen existiert bereits IFML (Interaction Flow Modeling Language). IFML ist Object Management Group (OMG) Standard seit März 2014 [vgl. 9].

Um Anforderungen an die DORI-DSL zu ermitteln und zu evaluieren, ob Teile von IFML in diese übernommen werden können, wurde anhand eines Beispiels die IMFL-Spezifikation mit verschieden Implementierungen verglichen. Zusätzlich wurde untersucht, wie GUI-Elemente, Interaktionen und Programmlogik in verschiedenen Frameworks realisiert und verbunden werden.

Als Implementierungen wurde das Framework Angular4 für Web-Entwicklung und Android SDK(JAVA) für native Android Entwicklung gewählt. Dazu wurde ein JAVA-REST Service implementiert, um eine gemeinsame Abstraktion für die Programmlogik zu schaffen.

Als Anwendungsbeispiel wurde ein Taschenrechner gewählt. Um die Komplexität der Umsetzung zu verringern, werden nur die Operanden '+' und '-' umgesetzt. Der Taschenrechner besteht aus zwei Bildschirmmasken. Die erste hat die Zahlen von 0-9, '+', '-' und '=' als Buttons und eine Anzeige des aktuellen Wertes. Beim Drücken eines Zahlbuttons wird die Zahl von rechts an den aktuellen Wert konkateniert. Beim Drücken der Buttons '+' oder '-' wird der aktuelle Wert als linker Operand einer Addition beziehungsweise Subtraktion gespeichert. Falls vorher eine Addition oder Subtraktion begonnen wurde, wird der aktuelle Wert mit einem zuvor gespeicherten Wert durch den zuvor gespeicherten Operator verknüpft. Beim Drücken des '=' Buttons wird auf die zweite Bildschirmmaske gewechselt, die letzte Operation ausgeführt und das Ergebnis angezeigt.

Um Anforderungen für den Editor zu formulieren wurde ein Papier-Prototyp erstellt, welcher im Folgenden näher erläutert werden soll.

5.2.1. IFML-Prototyp

Hannah

Die Modellierung dieses Taschenrechners in IFML ist in Abbildung 5.1 zu finden. Dort sind die zwei Bildschirmmasken „Calculator“ und „Result“ zu erkennen, welche als View-Container modelliert wurden. View-Container sind in IFML Elemente, die weitere View-Container oder View-Komponenten umfassen können. View-Komponenten repräsentieren dabei Elemente der Nutzerschnittstelle, welche Inhalt darstellen oder Eingaben akzeptieren. Der Calculator-Container enthält die View-Komponente „Screen“, die den aktuellen Wert des

Taschenrechners darstellt. Die View-Komponente „Result“ stellt im Result-Container das Ergebnis der Rechnung dar. Beide Komponenten enthalten ein Objekt vom Typ „Value“ und Details zu diesem werden angezeigt („Detail-Komponente“), in diesem Fall den im Value-Objekt gespeicherten Wert.

Nun können in der Screen-Komponente verschiedene Events auftreten. Wird ein Button mit einer der Zahlen 0-9 gedrückt (Event „0“. . . „9“), so bewirkt dies die Konkatenation dieser Zahl von rechts an den aktuellen Wert. Dies ist durch die Action „concat“ modelliert. Aktionen werden in IFML als Sechsecke modelliert. Diese benötigt einen Input-Parameter vom Typ „Value“. Durch ein sogenanntes Parameter-Bindung, lässt sich modellieren, dass es sich dabei gerade um das Value-Objekt handelt, welches in der Screen-Komponente angezeigt wird. Ein weiteres Parameter Binding von der Concat-Action zurück zur Screen-Komponente zeigt an, dass der Output der Action dem Value der Komponente zugeordnet wird. Schließlich ist zu bemerken, dass strenggenommen das Event „0“. . . „9“ als zehn verschiedene Events modelliert werden müsste, welche die verschiedenen Actions „concat0“ bis „concat9“ auslösen, was aus Gründen der Übersichtlichkeit jedoch zusammengefasst wurde.

Nach Drücken der Buttons „+“ bzw. „-“ (mit den gleichnamigen Events) wird die Action „add“ bzw. „sub“ ausgelöst. Der Navigationsfluss ist hier mit ähnlichen Parameter Bindings versehen wie die gerade beschriebene Konkatenierung. Dies gilt auch für den Navigationsfluss, welcher nach nach Drücken von „=“ eintritt. Nach Ausführen der Action „res“ wird hier jedoch der View-Container gewechselt und in der Result-Bildschirmmaske das Ergebnis der Rechnung angezeigt.

Der Prototyp wurde mit einem web-basierten Editor & Generator für IFML erstellt – einem Web Tool für modellbasiertes Rapid Prototyping von Web- und mobilen Anwendungen [vgl. 2]. Auch Produkte von WebRatio – einem Anbieter von Web- und mobilen Entwicklungsumgebungen – wurden im Zuge des Prototypings untersucht, da die Anwendungsentwicklungen dort ebenfalls eine Modellierung der UI mit IFML beinhaltet [vgl. 15]. Da das Ziel dieser Plattformen die Entwicklung entgültiger funktionaler und fertig gestalteter Anwendungen ist, lieferte dies für unseren Anwendungskontext zusätzliche Erkenntnisse. So stellte sich dort zum Beispiel heraus, dass eine Datenhaltung mit Datentypen sowie ein Datenfluss innerhalb des Interaktionsmodells sinnvoll ist.

5.2.2. Angular-Prototyp

Stephan

Angular4 ist eine Framework um Webanwendungen mit TypeScript beziehungsweise JavaScript zu entwickeln. Angular ist in erster Linie ein Frontend-Framework, das heißt, dass

Zugriffe auf Daten und Programmlogik durch eine separate API bereitgestellt werden sollen. In unserem Fall geschieht dies durch unseren REST Service.

Angular ist in sogenannte „Components“ aufgeteilt, im Folgenden wird dafür der Begriff Komponente verwendet. Eine Komponente hat ein Template mit einer HTML-ähnlichen Syntax und ein Modell. Das Modell und das Template werden zusammen in die Useransicht gerendert. Ein Template kann auch weitere Komponenten enthalten. Weiterhin können Elemente im Template mit Funktionsaufrufen versehen werden.

Um Funktionalität zwischen Komponenten zu teilen gibt es Services. In unserem Fall wird ein Service verwendet, der die entsprechenden REST-Aufrufe ausführt und einen Router-Service, der es erlaubt von einer Komponente zur nächsten zu wechseln. (Der Router-Service ist bereits im Framework enthalten.)

Wird Angular mit IFML verglichen, lässt sich erkennen, das die Komponenten ungefähr den View-Komponenten entsprechen, aber auch View-Container sein können. Die Funktionsaufrufe aus dem Template entsprechen den Events in den View-Komponenten. Für Data Binding, Systemevents und Events nach Actions gibt es keine Framework-Elemente. Wohl aber lassen sich Entsprechungen finden und werden im Einzelfall implementiert.

Eine wichtige Erkenntnis bei der Beispielimplementierung war, dass die Komponente nicht richtig funktioniert, wenn referenzierte Daten aus dem Modell nicht vollständig initialisiert werden. In IMFL gibt es zurzeit keine Entsprechung für initial erforderliche Daten, die zum Laden benötigt werden.

5.2.3. Android-Prototyp

Thomas

Android ist ein Betriebssystem für mobile Endgeräte. Java-basierende Anwendungen können unter Zuhilfenahme des Android SDK (Software Development Kit) entwickelt werden. Um wie beim Angular-Prototypen den Fokus nicht auf die Funktionalität zu setzen, wird auch hier der zuvor genannten REST Service verwendet.

Im Android-Framework kann eine zentrale Instanz, einer sogenannten Activity, für alle Nutzerinterkationen erstellt werden. In dieser wird dann das gesamte Verhalten der aktuellen Bildschirmmaske definiert. Jede Activity hat einen gewissen Lebenszyklus. Beim Starten der Activity wird eine Bildschirmmaske (Layout) an diese gebunden. Dieses Layout wird in einer separaten Datei definiert, wodurch eine strikte Trennung zwischen dem Verhalten und der Benutzersicht erfolgt.

Soll nun von einer Activity in die nächste gewechselt werden, so muss diese Absicht dem System anhand eines Intents mitgeteilt werden. Diesem muss zum Wechsel der Activity der

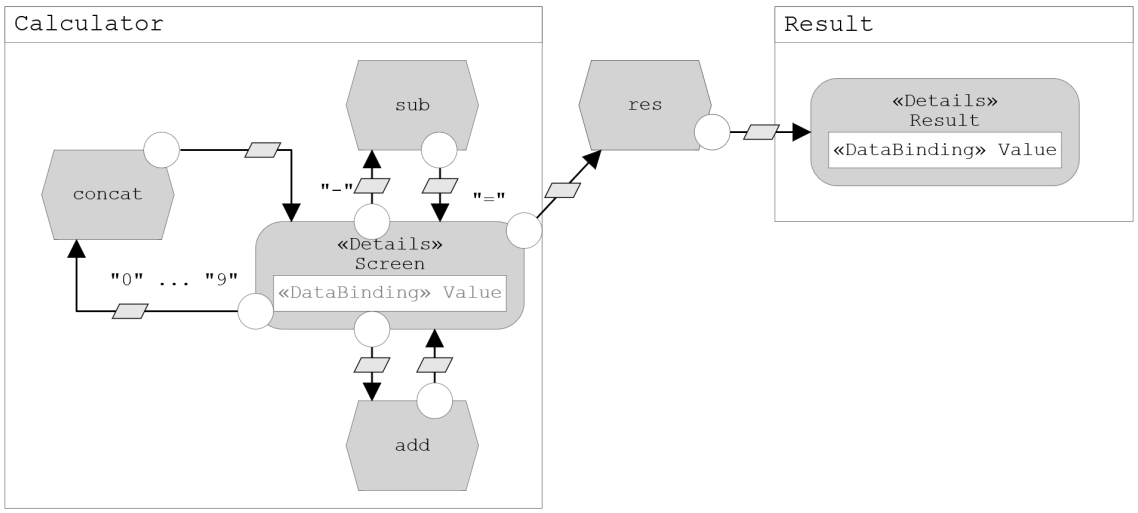


Abb. 5.1.: Modellierung des Taschenrechners in IFML

Name des Ziels mitgegeben werden. In diesem Prototyp wird der Intent nach der Antwort des REST-Servers auf die '='-Operation aufgerufen.

Beim Vergleich von Android mit IFML stellt sich heraus, dass die Grundzüge von IFML im Android-Framework abgebildet werden können. Die Activitys entsprechen den View-Containern, sind aber noch weit umfassender (zum Beispiel: Verhalten der Anwendung). Die Layouts entsprechenden den View-Components, wobei die Layouts auch Teile der View-Container übernehmen können, wie zum Beispiel das Vereinigen/Schachteln mehrerer Layouts. Alle anderen Elemente von IFML sind Teil der Activity. So kann für jede graphische Komponente (Button, Textfeld usw.) ein Event-Listener registriert werden. Dies würde jedoch das Event, die Action und eventuell sogar den Navigation bzw. Data Flow vereinen.

Die Erkenntnis des Android-Prototypen ist, dass alle Komponenten der IFML auf entsprechende Implementierung in Android abgebildet werden können, wobei die Trennung der einzelnen IFML-Komponenten aufgeweicht werden muss, da diese in der Activity (in der Programmlogik) verschmelzen.

5.2.4. Editor-Prototyp

Nancy

Abbildung 5.2 zeigt den Papier-Prototypen, der in einer Sitzung der Mitglieder der Projektgruppe entstanden ist. Dem Prototypen ist anzusehen, dass der Editor vom Aussehen her an die grafische Darstellung von Eclipse angelehnt ist..

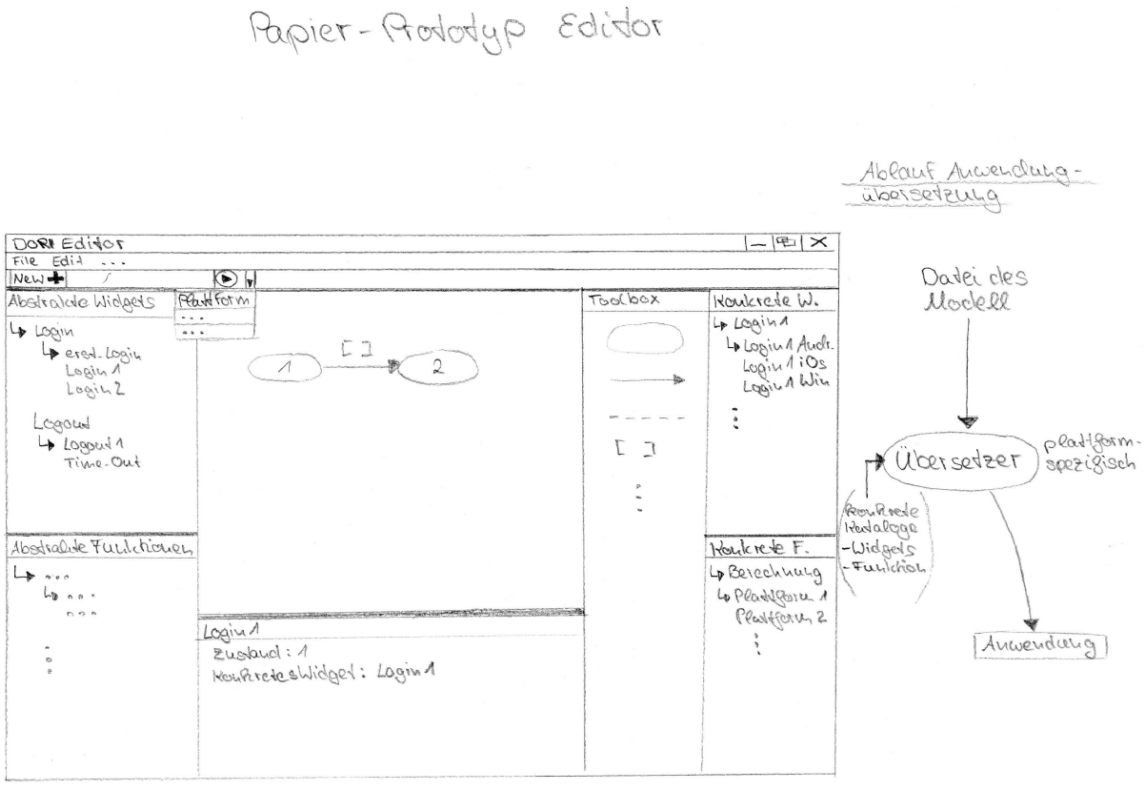


Abb. 5.2.: Papier-Prototyp des Editors

Dieser Prototyp zeigt, dass der Editor eine neue Erstellung eines Modells durch den Button „New“ ermöglicht. Diese Idee wurde aus anderen Editoren übernommen, die ebenfalls über solch einen Button verfügen.

Dieser Prototyp zeigt außerdem, dass der Editor eine Toolbox bereit stellt, welche die Elemente der DORI-DSL beinhaltet. Diese sollen in den Arbeitsbereich gezogen werden können.

Im unteren Teil des Editors sollen den Zuständen die abstrakten Widgets zugeordnet werden können, sowie den abstrakten Widget eine konkrete Implementierung.

An den Seiten sind Kataloge für die Widgets und Funktionen gegeben. Es soll jeweils ein Katalog für abstrakte und konkrete Widgets bzw. Funktionen angezeigt werden.

Über einen Button soll der Übersetzer ausgeführt werden. Der Prototyp zeigt nicht, ob der Button das Modell in eine Datei übersetzt, die an einen Interpreter weitergegeben wird oder ob das Modell direkt in Code generiert wird. Dennoch ist neben dem Übersetzer ein Drop-Down Menü gegeben, welches eine Auswahl der Plattform zum Übersetzen ermöglicht.

Dieser Prototyp zeigt vor allem die grundlegenden, oberflächlichen Funktionalitäten. Für eine Detaillierung müssen weitere Prototypen erstellt werden.

6. Projekthandbuch

Hauke, Lisa

Zusätzlich zu dem Vorgehen werden in dieser Dokumentation die beteiligten Rollen vorgestellt sowie die Tools, die von der Projektgruppe verwendet werden. Dabei geht es um jene Tools, die die Planung sowie die Dokumentation unterstützen. Zusätzlich sollen die internen Regeln der Projektgruppe vorgestellt werden.

6.1. Beteiligte Rollen

Hauke, Lisa

Projektleitung Die Projektleitung trägt die Projektleitung für:

- Verantwortung für die Projektergebnisse (Termine, Inhalt/Qualität)
- projekt-interne Kommunikation und zu Auftraggeber und Stakeholdern.
- Lösung projekt-intern eskalierter Themen (Entscheidungsfindung)

Die Projektleitung hat die Kompetenz der Entscheidungsbefugnis zu Inhalt/Termin im Rahmen der vom Auftraggeber vorgegebenen Rahmenbedingungen. Die folgenden Aufgaben gehören zu der Rolle der Projektleitung:

- Führt das Projekt mit Projektplanung und den Schlüsselrollen QA-Verantwortlicher, Anforderungsanalytiker und Systemarchitekt.
- Durchführung notwendiger Projektmanagement-Aufgaben/-Prozesse bzw.
- Delegation an weitere Rollen, Teilprojekte oder Projektmitarbeiter.
- Vorbereitung und Durchführung aller großen Meilensteintermine bzw.
- Überprüfungen („Reviews“)
- Kommunikationsmanagement
- Projekt-interne (De-)Eskalationsinstanz

- Motivation des Teams
- Erarbeitung und Etablierung von Prozessen im Umgang mit Jira
- Kontrolle der Umsetzung der Prozesse
- Einführung einer Confluence-Struktur
- Schaffung von Arbeitsanweisungen für Confluence
- Kontrolle von Confluence und der Arbeit in diesem

Diese Rolle wird von Hauke und Lisa besetzt.

Projektplanung Die Projektplanung trägt die Verantwortung für die Einhaltung der folgenden verschiedenen Ziele:

- Verantwortung für die Projektergebnisse (Termine, Inhalt/Qualität)
- Risikomanagement
- Reaktion auf Planabweichungen

Die Kompetenzen liegen in dem Aufstellen und Anpassen der Projektplanung an die gegebenen Rahmenbedingungen. Die Aufgaben umfassen das/die:

- Aufstellen der Projektplanung
- Verfeinern des Projekthandbuches
- Erstellen der Zeitplanung und Aufwandschätzung (Meilensteine)
- Risikoplanung zum Umgang mit möglichen Risiken
- Einhalten des Zeitplans überwachen
- Verwaltung der Projektteilaufgaben

Diese Rolle wird von Hauke und Lisa besetzt.

Administratoren Die Administratoren tragen die Verantwortung für die Verwaltung und den Betrieb der Tools. Dies beinhaltet:

- Verantwortung für die Rechteverteilung
- Betrieb der Software und beseitigen von Problemen
- Einführung neuer Tools

Die Kompetenzen der Administration liegen in der Verwaltung der Rechte und dem Überblick über den reibungslosen Betrieb der verwendeten Tools. Die Folgenden Punkte liegen in dem Aufgabenbereich der Administration:

- Verteilen der notwendigen Rechte an Benutzer
- Anlaufstelle bei Problemen von Benutzern
- Einrichten neuer Tools
- Vorgaben für den Umgang mit der IT-Infrastruktur
- Kontrolle der Vorgaben

Diese Rolle wird von Christopher und Felix besetzt.

Dokumentationsbeauftragte Diese Rolle trägt die Verantwortung für die Dokumentation, sprich für:

- Qualität der Dokumentation
- Einhaltung von Meilensteinen in der Erstellung

Die Kompetenzen dieser Rolle liegen in dem:

- Erstellen und Anpassen verschiedener Richtlinien für die Dokumentation
- Zurückweisen der Dokumentationen bei Nichteinhaltung der Richtlinien

Außerdem umfasst diese Rolle folgende Aufgaben:

- Erstellen von Dokumentationsleitlinien und Vorlagen
- Vorantreiben der Dokumentationserstellung
- Überwachung der Dokumentationsqualität
- Erarbeitung und Etablierung von Dokumentenstandards
- Kontrolle der Einhaltung der Standards

- Vorgaben zu Dokumentationsprozessen
- Kontrolle der Vorgaben zu Dokumentationsprozessen

Diese Rolle wird von Hannah und Nancy besetzt.

Qualitätssicherung Die Qualitätssicherungsrolle wurde am 12. Oktober offiziell eingeführt. Sie dient zum einem der Integration der Gruppen des Editors und Interpreters, zum anderen der Definition und Erstellung von Testfällen. Also insbesondere:

- der Zusammenführung der beiden Gruppen Editor und Interpreter
- den Integrationstests
- der Überwachung der Funktionalität aus Sicht des Benutzers

Die Kompetenzen dieser Rolle liegen in der:

- Integration der Gruppen des Editors und Interpreters, zum anderen der Definition und Erstellung von Testfällen.

Außerdem umfasst diese Rolle folgende Aufgaben:

- Für die Definition von Testfällen sollen von dieser Rolle verschiedene Anwendungsfälle für den Editor erstellt, dokumentiert und durchgeführt werden.
- Zudem wird festgehalten, ob das Ergebnis den vorher definierten Erwartungen entspricht.
- Um eine Brücke zwischen den beiden großen Gruppen zu schaffen, werden von den QA-Beauftragten Integrationstest erstellt.
- Das Testen des Editors aus Sicht eines Benutzers geschieht durch die Qualitätssicherung, ebenso wie die Verwendung des Interpreters und insbesondere die Ausführung der ausgewählten Anwendung.
- Außerdem ist die Entwicklung eines Konzeptes für den Informationsaustausch/Dateiaustausch zwischen den Einzelgruppen, sowie die Überprüfung der Umsetzung dieses Konzeptes Aufgabe dieser Rolle.

Diese Rolle wird von Hannah und Nancy besetzt.

6.2. Beschreibung der Tools

Hauke, Lisa

Jira

Hauke, Lisa

Jira ist eine von Atlassian entwickelte Software für operatives Projektmanagement, Fehlerverwaltung, Problembehandlungen und Aufgabenmanagement in Projekten. Jira findet seinen häufigsten Einsatz bei der Softwareentwicklung.

Des Weiteren bietet Jira die Möglichkeit über Plugins erweitert zu werden, welche unter verschiedenen Lizenzmodellen aus dem Atlassian Marketplace bezogen werden können. Generell ist Jira eine propetiäre Software, welche allerdings für NGOs und Open Source Projekte, bei der Erfüllung spezifischer Auflagen, frei zur Verfügung gestellt werden kann (Ähnliches gilt auch für weitere Atlassian-Produkte, wie Confluence).

Innerhalb dieses Softwareprojektes kommt Jira in der Version 7.2.6 zum Einsatz und soll zur zentralen Projektverwaltung dienen, die Koordination von Entwurf, Entwicklung und die Verfolgung von projektbezognenen Aufgaben.

Jira-Plugins: keine

Offizielle Adresse des Projekt-Jiras: `https://jira.swp.offis.uni-oldenburg.de/projects/PGDORI/summary`

Confluence

Hauke, Lisa

Confluence stellte eine kommerzielle Wikiplattform von Atlassian dar. Die zentrale Aufgabe von Confluence ist es eine Plattform für Wissensaustausch, Wissenskonzentration und Kommunikation für Unternehmen und Organisationen zu etablieren. Confluence bietet, neben den gängigen Wiki-Funktionalitäten, einen Rich Text-WYSIWYG-Editor, der, unter anderem, Drag and Drop und Autovervollständigung bietet. Des Weiteren lassen sich unterschiedliche Medien in Confluence einbinden, wie .pdf Dateien. Confluence lässt sich, wie JIRA über den Marketplace mit Plugins erweitern.

Bei diesem Softwareprojekt findet Confluence in der Version 6.1.2 Verwendung und soll als Plattform zum Wissensmanagement genutzt werden, um allen Projektmitgliedern den gleichen Zugang zu relevanten Information zu ermöglichen und einen Austausch über diese zu ermöglichen.

Confluence-Plugins: keine

Offizielle Adresse des Projekt-Confluences: `https://confluence.swp.offis.uni-oldenburg.de/display/PGDORI/PG_Dori+Home`

SSH

Felix

Für die Verbingunden zu den Servern wurde SSH eingesetzt. Unixoide Clients konnten so mittels der vorinstallierten SSH Clients auf Server zugreifen. Für Windows Clients wurde, wurde Putty verwendet. Eine Alternative zu SSH stand nicht zur Verfügung.

Konsolenbefehle

Felix

Für alle Aufgaben, die mittels SSH durchgeführt werden mussten, wurde eine Übersicht mit den relevanten Konsolenbefehlen erstellt.

- Durch Directories bewegen: cd
  - Aus dem aktuellen Verzeichnis in das Verzeichnis ftp wechseln: cd ftp
  - wenn das Verzeichnis nicht im gleichen Verzeichnis liegt, wie das in dem man sich gerade befindet: cd /home/tool/ftp
  - eine Ebene hoch/zurück: cd ..
- Verzeichnis erstellen:
  - mkdir VERZEICHNISNAME
- Datei umbenennen
  - mv ALTERNAME NEUERNAME
- Datei löschen
  - rm DATEI
  - Um sicher zu gehen hier mit absoluten Pfaden arbeiten, wenn möglich.
- Verzeichnis löschen
  - rm -rf VERZEICHNIS

  - Um sicher zu gehen hier mit absoluten Pfaden arbeiten, wenn möglich.
- Kopieren
  - cp -R DATEI KOPIE
  - Um sicher zu gehen auch hier mit absoluten Pfaden arbeiten, wenn möglich.
- Passwort ändern:
  - In der Konsole anmelden. Alle weiteren Punkte jeweils mit Enter bestätigen
  - passwd eingeben, dann den Anweisungen folgen, welche unten aufgeführtes besagen:
  - Einmal das aktuelle Passwort eingeben
  - Das neue Passwort eingeben
  - Nochmal das neue Passwort eingeben

FTP

Felix

FTP steht für File Transfer Protocol und ist ein Datenübertragungsprotokoll für den Austausch von Daten in einem Netzwerk, welches 1985 im RFC 959 spezifiziert wurde. Häufig wird FTP zum Hochladen von Dateien auf einen Server von einem Client oder zum Herunterladen von Dateien von einem Server auf einen Client eingesetzt [6].

Innerhalb des Projektes DORI wird ein FTP-Server als alternative und unkomplizierte Datenaustauschmöglichkeit verwendet. Es wird sFTP (secure File Transfer Protocol) genutzt, welche eine sichere Datenübertragung gewährleistet [6].

Git

Felix

Git ist eine Open Source Software zur dezentralen Verwaltung von Dateien mit Versionskontrolle. Git verzichtet auf einen zentralen Server. Dies bedeutet, dass die Repositorys als lokale Kopien bei den lokalen Benutzern liegen – inklusive der Versionshistorie. Git lässt sich mittels Befehlszeile oder GUI-basierten Programmen nutzen und verwalten.[14]

Innerhalb des Projektes wird Git verwendet und dient zur Verwaltung von projektbezogenen-Dateien und dem damit verbundenen dezentralen Datenaustausch. Zur Auswahl stand neben Git noch SVN. Durch eine Abstimmung innerhalb der Projektgruppe wurde die Verwendung von Git festgelegt.

6.3. Regeln

Hauke, Lisa

In diesem Abschnitt sind die gesammelten Regeln, welche während der Umsetzung des Projektes für die gemeinsame Teamarbeit vereinbart und festgelegt wurden. Dies erstreckt sich über Arbeitsregeln mit Tools bis zu Umgangsregeln auf zwischenmenschlicher Ebene, sowie Urlaubsregelungen.

Krankheit/Abwesenheit:

- Absagen von offiziellen Treffen bitte an den allgemeinen DORI-Mailverteiler.
- Absagen – wenn möglich – bitte frühzeitig.
- Bei längerer Krankheit/Abwesenheit bitte rechtzeitig längeren Ausfall anmerken.
- Auch bei Arbeitsgruppentreffen etc. immer abmelden.

Urlaub:

- Urlaub kann beliebig genommen werden (6 Wochen)
- dies frühzeitig mitteilen und in den Kalender und die Liste im Confluence eintragen
- vor dem Urlaub Arbeitsplanung anpassen und mit der derzeitigen Arbeitsgruppe abklären
- Niemand wird aus dem Urlaub zurückgeholt, Verantwortlichkeiten und Vertreter vorher festlegen

JIRA:

- JIRA aktiv benutzen
- Aufgabenerstellung nur nach dem Vier-Augen-Prinzip, keine Alleingänge.
- Verwendung der Statusverwaltung von Aufgaben etc.: Aufgabe -> In Arbeit -> Zur Überprüfung -> In Überprüfung -> Fertig
- Bei Problemen oder Zeitmangel bitte rechtzeitig Bescheid geben

- Aufgaben immer zuordnen, Aufgaben ohne Bearbeiter und Aufgaben an denen nicht weiter gearbeitet wird vermeiden
- Unteraufgaben in der Obe-Aufgabe über den Menüpunkt „Unteraufgabe“, nicht über den Erstell-Button eingefügen
- Bearbeiter der Aufgabe ist nicht Tester, dieser wird vorab in den Kommentaren festgelegt und übernimmt

Confluence:

- Keine leeren Seiten erstellen.
- Neue Seiten bitte auf der überliegenden Seite verlinken, die Struktur im Wiki folgt aus den Links - Seiten können beliebig verschoben werden.
- Gründliches und gewissenhaftes Erstellen von Inhalten.
- Bei Verweisen auf externe Seiten nur verkürzte Links (.z.B.: goo.gl) nutzen
- nach dem Hochladen von PDFs die kleinste Vorschau wählen
- Diagramme können mit externen Tools erstellt werden, Einbinden dann über JPG-Datei
- alle Dokumente müssen einem Review unterliegen

Git:

- Es sollen keine Kompilate, sondern nur Quellcode ins Git kommen (also z.B. nur die .tex, nicht aber die .pdf, ebensowenig die Hilfsdateien die z.B. LaTeX anlegt)
  - Ins Git gestellte Dateien werden automatisch kompiliert und sind, z.B. im Falle von LaTeX-Dokumenten, danach als PDF im FTP zu finden
- Wird die Ordnerstrucktur im Git geändert, so ist darauf zu achten, dass auch in alten Dokumenten die Pfade aktuell bleiben, da die Continous Integration sonst fehlschlägt. Es sollen also immer nur kompilierbare Dateien auf dem Master-Branch liegen.
  - Alternativ können alte/sonstige Dateien auch in einen anderen Branch im Git geschoben werden. Bei Bedarf (größere und wichtige Dokumente) kann ein eigener angelegt werden.

6.4. Vorgehensmodell

Hauke

Die Projektgruppe DORI wird die Anforderungen nach einem bestimmten Vorgehensmodell umsetzen. Das Vorgehen der Projektgruppe DORI orientiert sich an dem Vorgehensmodell Scrum.

Scrum-Grundlagen Scrum ist ein iteratives und inkrementelles Vorgehensmodell. In Scrum wird neben dem Produkt auch die Planung iterativ und inkrementell entwickelt. Der langfristige Plan (Product Backlog) wird kontinuierlich verfeinert und verbessert. Der Detailplan (Sprint Backlog) wird nur für den jeweils nächsten Zyklus (Sprint) erstellt. Damit wird die Projektplanung auf das Wesentliche fokussiert [13].

Scrum setzt auf drei Säulen:

1. Transparenz: Fortschritt und Hindernisse eines Projektes werden regelmäßig und für alle sichtbar festgehalten.
2. Überprüfung: In regelmäßigen Abständen werden Produktfunktionalitäten geliefert und sowohl das Produkt als auch das Vorgehen beurteilt.
3. Anpassung: Anforderungen an das Produkt, Pläne und Vorgehen werden nicht ein für alle Mal festgelegt, sondern kontinuierlich detailliert und angepasst. Scrum reduziert die Komplexität der Aufgabe nicht, strukturiert sie aber in kleinere und weniger komplexe Bestandteile, die Inkremente.

Scrum-Aspekte: Sprint: Ein Sprint ist ein geplanter Arbeitsabschnitt, in dem die Umsetzung einer festgelegten Produktfunktionalität durchgeführt wird. Zu Beginn eines Sprints erfolgt ein Sprintplanning und zum Ende eines Sprints ein Sprintreview.

Sprint Planning: Im Sprint Planning werden zwei Fragen beantwortet: Was kann im kommenden Sprint entwickelt werden? Wie wird die Arbeit im kommenden Sprint erledigt? In der Planung wird zuerst das „Was“ und anschließend das „Wie“ geplant.

Daily Scrum: Zu Beginn eines jeden Arbeitstages trifft sich das Entwicklerteam zu einem kurzen Daily Scrum, bei dem Scrum Master und Product Owner häufig anwesend, jedoch nicht aktiv beteiligt sind, falls sie nicht selbst Backlog-Elemente bearbeiten. Zweck des Daily Scrum ist der Informationsaustausch. Im Daily Scrum werden keine Probleme gelöst – vielmehr geht es darum, sich einen Überblick über den aktuellen Stand der Arbeit zu verschaffen.

Sprint Review: Das Sprint Review steht am Ende des Sprints. Hier überprüft das Scrum Team das Ergebnis, um den Produkt Backlog bei Bedarf anzupassen. Das Entwicklungsteam präsentiert seine Ergebnisse und es wird überprüft, ob das zu Beginn gesteckte Ziel

erreicht wurde. Das Scrum Team und die Stakeholder besprechen die Ergebnisse und was als Nächstes zu tun ist.

Product Backlog Refinement: Das Product Backlog Refinement, welches auch Backlog Grooming genannt wird, ist ein fortlaufender Prozess, bei dem der Product Owner und das Entwicklungsteam gemeinsam den Product Backlog weiterentwickeln. Hierzu gehören: Ordnen der Einträge, Löschen von Einträgen die nicht mehr wichtig sind, Hinzufügen von neuen Einträgen, Detaillieren von Einträgen, Zusammenfassen von Einträgen, Schätzen von Einträgen, Planung von Releases.

SCRUM-Artefakte: Product Backlog: Das Product Backlog spiegelt sämtliche Anforderungen an ein Produkt wider und hat aufgrund des inkrementellen und iterativen Charakters keinen Anspruch auf Vollständigkeit.

Sprint Backlog: Bildet die Planung für den jeweiligen Sprint mit ausgewählten Anforderungen aus dem Product Backlog.

Anpassungen des Vorgehensmodells an die Bedürfnisse der PG Für die PG und ihr individuelles Vorgehensmodell wurden einige Anpassungen an Scrum vorgenommen:

- Es findet jeweils ein Sprintplanung und ein Sprintreview statt.
  - Nachpflege von Tasks im laufenden Sprint sind möglich.
- Klassische Scrum Rollen wie Scrum Master oder Productowner werden nicht gesondert besetzt sondern von anderen Rollen teilweise wahrgenommen.
- Anstatt Daily Scrum erfolgt ein Weekly Scrum in dem die Mitglieder innerhalb etwa einer Minute berichten was sie erledigt haben und was noch zu erledigen ist bzw. wo es Probleme gibt oder zu erwarten sind.
  - Zusätzlich sollen die Verantwortlichen der Teilgruppen sich einmal die Woche (Sonntags) zurück melden (Mailverteiler).
- Es erfolgt keine Aufwandsplanung mittels Storypoints oder Planning Poker.
  - Aufwand wird über Mann-Stunden geschätzt.
- Die Projektplanung erfolgt über Confluence und JIRA.
- Abbildung 6.1 zeigt das grobe Muster eines Sprints:

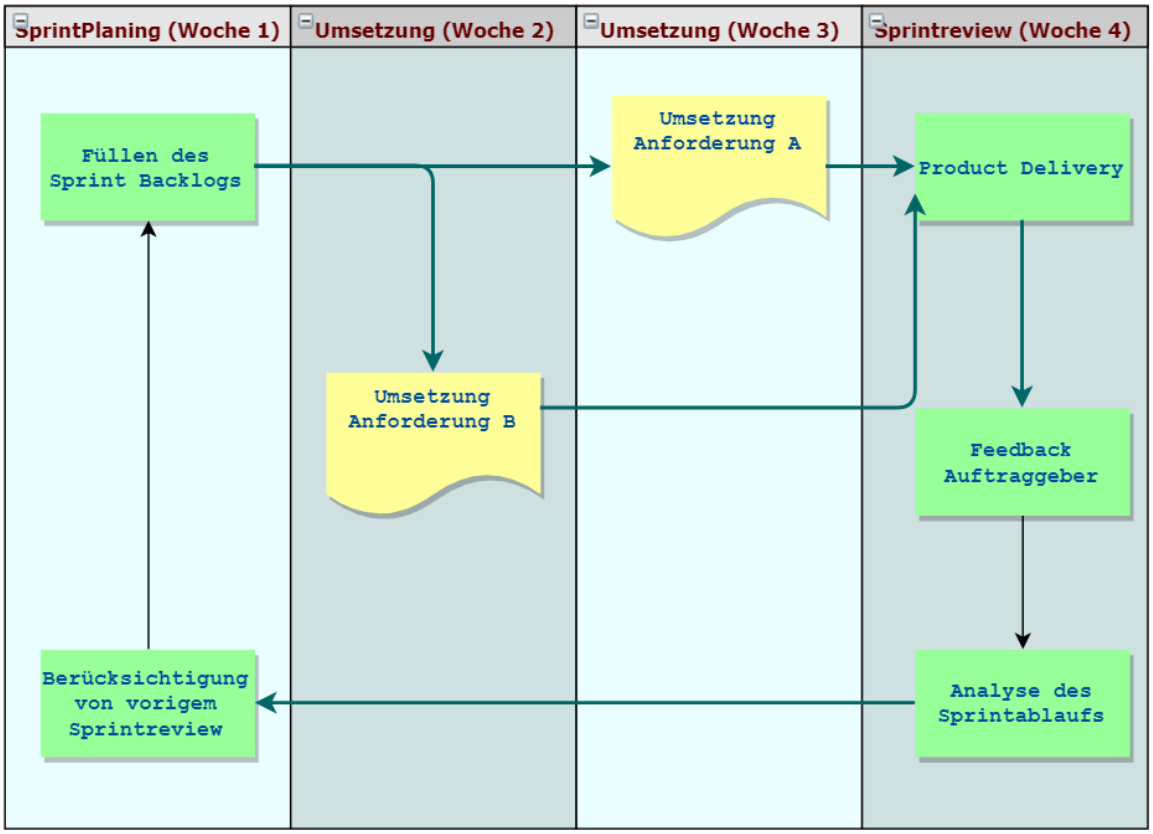


Abb. 6.1.: Vorgehen (eigene Anfertigung)

Teil II.

Systemdokumentation

7. Globale Analyse

Felix, Thomas, Jan (ehemaliges Projektgruppenmitglied), Nancy

Die globale Analyse soll die Faktoren herausstellen, die Einfluss auf Architekturentscheidungen, Architekturentwurf oder organisatorische Verwaltung der Projektgruppe haben. Dabei werden diese Faktoren in organisatorische, technische und produkt Faktoren aufgegliedert.

7.1. Organisatorische Faktoren

Felix, Thomas, Jan (ehemaliges Projektgruppenmitglied), Nancy

Die organisatorischen Faktoren beschreiben die Faktoren, welche Einfluss auf die gesamte Projektgruppe haben. Diese werden in der Tabelle 7.1 dargestellt.

Tab. 7.1.: Organisatorische Faktoren

Faktor	Flexibilität/Variabilität	Wirkung
O1: Managment		
O1.1: Entwickeln oder Kaufen		
Die Entwicklung des Editors wird durch vorhandene, anzupassende Software gestützt. Je nach Wahl der Editorumgebung würden Nutzungskosten entstehen. Die Entwicklung der DSL und des Übersetzers erfolgt durch die Projektgruppe selbst, somit kostenfrei.	Mit einer Entscheidung zu MetaEdit+, als Entwicklungstool würden Nutzungskosten enstehen. Der Kauf eines bestehenden Gesamtsystem kommt nicht in Frage.	Entsprechende Anpassungen und Funktionalitäten müssen selbst entwickelt werden, bzw. muss zur Erstellung des Editors ein Framework ausgewählt werden.
O1.2: Zeitplan oder Funktionalitäten		
Die Projektdauer ist auf zwei Semester beschränkt.	Die Projektdauer kann nicht verlängert werden.	Der Projektleitung wird ein klar definierter Zeitrahmen vorgeben. Eventuell werden aufgrund der beschränkten Projektdauer Abstriche bei der Funktionalität vorgenommen.
O1.3: Unternehmensziele		
Das Ziel ist die Erstellung einer Software für die modellgetriebenen Entwicklung von Applikationen.	Der grundlegende Funktionsumfang wird bei Projektstart festgelegt.	Die Projektgruppe muss die vereinbarten Ziele des Projekts umsetzten.
O2: Mitarbeiterinteressen und -kenntnisse		
O2.1: Analyse und Entwurf		

Fortsetzung auf der nächsten Seite

Tab. 7.1.: *Fortsetzung der* Organisatorischen Faktoren

Faktor	Flexibilität/Variabilität	Wirkung
Der Entwurf für die Erstellung der DSL wird ein UML-Metamodell sein. Das Metamodell für die abstrakte Syntax wird in Form eines Ecore-Metamodells erstellt. Die konkrete Syntax beschreibt ein Editor.	Nur durch einen hohen Aufwand wäre eine Änderung des Entwurfskonzepts möglich.	Die Analyse und der Entwurf werden mithilfe der Unterstützung von UML-Technologien gestaltet.
O2.2: Implementierung		
Durch das Modulangebot der Universität Oldenburg fokussiert sich die Implementierung auf Java.	Nur durch einen hohen Aufwand wäre eine Änderung der Implementierung möglich.	Aufgrund der Nutzung von Java haben alle Mitglieder Kenntnisse der verwendeten Programmiersprache. Des Weiteren hat Java im Allgemeinen den Vorteil einer großen Plattformunterstützung. Allerdings können natürlich mit jeder Designentscheidung auch bestimmte Einschränkungen entstehen, so muss auf allen Plattformen Java lauffähig sein.
O3: Prozess- und Entwicklungsumgebung		
O3.1: Prozessmodell		

Fortsetzung auf der nächsten Seite

Tab. 7.1.: *Fortsetzung der* Organisatorischen Faktoren

Faktor	Flexibilität/Variabilität	Wirkung
Als Prozessmodell wird ein von der Projektgruppe angepasstes Scrum verwendet.	Ein unangepasstes, vollständiges Scrum Prozessmodell wäre für die Projektgruppe zu komplex. Der Scrum Prozess umfasst für die Projektgruppe zu starre Strukturen, so wurden etwa keine konkreten Rollen, wie „Product Owner“ eingeführt oder die zeitlichen Vorgaben wurden nicht direkt übernommen. Es wurden allerdings Kernelemente wie Sprintplanung, Sprints mit einem Informationsaustausch der Mitglieder und ein Sprint Review behalten.	Durch eine Anpassung des Prozessmodells wird der Arbeitsprozess agiler. Die Projektgruppe nutzt nur die Aspekte die als nützlich betrachtet werden.
O3.2: Entwicklungsplattform		
Die gängigen Plattformen (Linux, OSX, Windows) müssen von dem Editortool unterstützt werden. Nur so, kann plattformunabhängig eine gewünschte Interaktion modelliert werden.	Flexibel	Nur mit der entsprechenden Plattformunterstützung kann auf der jeweiligen Plattform modelliert werden.
O3.3: Entwicklungswerkzeuge		
MetaEdit+ und Sirius stehen zur Auswahl.	Flexibel	Bei MetaEdit+ treten Kosten auf und die Einarbeitungszeit ist höher. Alle Entwicklerwerkzeuge müssen die verwendeten Plattformen der Entwickler unterstützen.
O3.4: Testprozess und -werkzeuge		

Fortsetzung auf der nächsten Seite

Tab. 7.1.: *Fortsetzung der* Organisatorischen Faktoren

Faktor	Flexibilität/Variabilität	Wirkung
Der Projektgruppe stehen mobile Endgeräte, sowie private Endgeräte der Mitglieder und die ARBI-Ressourcen zur Verfügung.	Flexibel, es können Änderungen auftreten.	Die entwickelten Produkte können nur auf den Geräten getestet werden, die zur Verfügung stehen.
O4: Zeitplan		
O4.1: Auslieferungsplan		
Die zu erstellende Funktionalität ist durch feste Meilensteine seitens der Projektgruppe festgelegt.	Die Meilensteine sind bedingt verschiebbar.	Die Pufferzeit verändert sich durch die Verschiebung von Meilensteinen, dies erfordert eine Anpassung der Planung.
O5: Budget		
O5.1: Mitarbeiter		
Die Projektgruppe besteht aus neun Mitgliedern. Die Projektgruppe wird von 3 Lehrenden betreut.	Ein Verlassen der Projektgruppe ist möglich.	Durch weniger Projektmitgliedern verändert sich die Ressourcenplanung. In diesem Fall kann eine neue Arbeitseinteilung der Mitglieder vorgenommen werden.
O5.2: Werkzeuge		
Der Projektgruppe steht kein Budget für Werkzeuge zur Verfügung.	Geringfügig flexibel.	Anschaffungen müssen im einzelnen mit den Betreuenden abgestimmt werden.

7.2. Technische Faktoren

Felix, Thomas, Jan (ehemaliges Projektgruppenmitglied), Nancy

Die technischen Faktoren beschreiben die Faktoren, die Einfluss auf die technische Umsetzung des Gesamtsystems haben. Diese werden in der Tabelle 7.2 dargestellt.

Tab. 7.2.: Technische Faktoren

Faktor	Flexibilität/Variabilität	Wirkung
T3: Software		
T3.1: Benutzerschnittstelle		
Der DORI-Editor ist die Schnittstelle zum Interaktionsentwickler, der den Editor verwendet. Zusätzlich bildet der DORI-Editor die Schnittstelle zum Interpreter oder Generator, der den nächsten Arbeitsschritt bildet.	Nicht flexibel	Im Modellierungsprozess wird eine Toolchain verwendet, in der die verschiedenen Arbeitsschritte durchlaufen werden.
T3.2: Software-Fremdkomponenten		
Sollte die Wahl auf MetaEdit+ fallen, muss für eventuelle Kataloge eine Fremdkomponente verwendet werden.	Es besteht eine Wahl zwischen Sirius und MetaEdit+	Die Entscheidung über die Software-Fremdkomponente stellt die Grundlage für die weitere Entwicklung.
T4: Architektur		
T4.1: Architekturstile		
Der Architekturstil der Projektgruppe folgt dem Architekturstil "QUADRAT", welche sich in Analyse-, Modellierungs- und Evaluierungsphase unterteilt.	Ein anderer Architekturstil wäre auch möglich.	QADRAT ist bekannt, ein anderer Architekturstil würde Einarbeitungszeit benötigen.
T4.2: Produktlinien-Architektur		

Fortsetzung auf der nächsten Seite

Tab. 7.2.: *Fortsetzung der* Technischen Faktoren

Faktor	Flexibilität/Variabilität	Wirkung
Der Editor stellt einen festen Anteil der Produktlinien-Architektur dar. Die austauschbaren Komponenten werden durch die plattformenspeziefischen Übersetzer dargestellt.	Es können mehrere Plattformen in einem Übersetzer realisiert werden.	Dadurch, dass der Editor plattformspezifische Komponenten erzeugen kann, ist die Benutzung des Editors auf nur einer Plattform notwendig.

7.3. Produkt Faktoren

Felix, Thomas, Jan (ehemaliges Projektgruppenmitglied), Nancy

Die Produktfaktoren sind die Faktoren, die Einfluss auf die Benutzerfreundlichkeit sowie den Nutzen des Gesamtsystems haben. Diese sind in der Tabelle 7.3 dargestellt.

Tab. 7.3.: Produkt Faktoren

Faktor	Flexibilität/Variabilität	Wirkung
P1: Funktionalität		
Die gewünschte Funktionalität wird u.a. über eine Domänenspezifische Sprache, einen Editor und einen Interpreter umgesetzt.	Flexibel	Mit der DSL können Elemente der konkreten Syntax dargestellt werden. Mit dem Editor kann eine Modellierung umgesetzt werden und mit dem Interpreter die Übersetzung in die Anwendung.
P4: Zuverlässigkeit		
P4.1: Verfügbarkeit		

Fortsetzung auf der nächsten Seite

Tab. 7.3.: *Fortsetzung der* Produkt Faktoren

Faktor	Flexibilität/Variabilität	Wirkung
Durch die Installation des DORI-Editors ist dieser auf dem jeweiligen System verfügbar.	Flexibel	Die Architektur muss dementsprechend gestaltet werden.
P5: Fehlerbehandlung		
P5.1: Erkennung		
Eine Übersetzung in eine Endanwendung soll nur im fehlerfreien Zustand möglich sein. Es soll vor der Übersetzung eine Konsistenzprüfung im Editor stattfinden.	Nicht flexibel, da feste Vorgabe.	Geringerer Aufwand für die Erstellung eines Übersetzers.
P7: Produktkosten		
P7.1 Lizenzkosten		
Bei einer Verwendung von MetaEdit+ würden entsprechende Lizenzkosten entstehen.	Nicht flexibel.	Der DORI-Editor könnte weder kostenfrei, noch Open Source angeboten werden.

8. Entscheidungsfindung

Nancy

Wie bereits aus den technischen - und den Produktfaktoren hervor geht musste sich die Projektgruppe für verschiedene Technologien entscheiden. Dazu zählt etwa die Festlegung der Betriebsplattformen und des einzusetzenden Tools für den DORI-Editor sowie die Entscheidung bezüglich der Art des Übersetzers und der Zielplattform für die erste Iteration. Auch das Vorgehen für den Entwurf des Meta-Modells der DORI-DSL musste festgelegt werden. Diese Entscheidungen dienen dazu einen Architekturentwurf zu erstellen und die Schnittstellen der einzelnen Komponenten festzulegen. Im Folgenden werden diese Entscheidungsfindungen näher erläutert.

8.1. Vorgehen beim Entwurf der DORI-DSL

Hannah

Als Inspiration und Grundlage für die Erfüllung des ersten Zieles der Projektgruppe, der „Entwicklung eines plattformunabhängigen Konzepts zur graphischen Darstellung einer Gesamtinteraktion und hierfür eine plattformunabhängige Beschreibungsmöglichkeit für Widgets und Funktionalitäten“, wurden zwei Modellierungssprachen identifiziert – UML und IFML. Hierbei wurde sich an den bestehenden Modellierungssprachen orientiert. Beide Sprachen realisieren das in Kapitel 2 im Lösungsansatz 1 gewählte Konzept von Zustandsautomaten auf leicht unterschiedliche Weise.

Bei UML (Unified Modelling Language) [10] handelt es sich um eine standardisierte Modellierungssprache der OMG (Object Management Group) zur Spezifikation, Visualisierung und Dokumentation von Software-Systemen. Die Sprache enthält verschiedene Diagrammtypen zur Darstellung unterschiedlicher Aspekte des Gesamtsystems. Im Speziellen existiert der Diagrammtyp „Zustandsdiagramm“. Da UML jedoch bewusst allgemein gehalten ist, fehlen innerhalb dieses Diagrammtyps für den gegebenen Anwendungsfall benötigte Konzepte, wie abstrakte oder konkrete Inhalte. Spezifiziert sind lediglich generische Zustände sowie Ereignisse, welche Zustandsübergänge auslösen. Die Übergänge können an Bedingungen gekoppelt werden und beim Auslösen einer Transition können Aktionen durchgeführt werden.

Eine hierarchische Schachtelung von Zuständen sowie orthogonale Bereiche sind modellierbar. Den Zuständen und Zustandsübergängen können jedoch keine Daten zugeordnet werden.

Bei IFML handelt es sich – wie bereits in Abschnitt 5.2.1 erwähnt – um eine Sprache zum Darstellen des Inhalts, der Nutzerinteraktion und des Kontrollverhaltens des Frontends von Softwaresystemen. Die Sprache liegt damit nahe am gegebenen Anwendungsfall der Projektgruppe. Das Konzept der abstrakten Inhalte ist hier in Ansätzen durch View Components (für abstrakte Widgets) und Actions (für abstrakte Funktionen) abgedeckt. Eine Schachtelung der View Components ist jedoch nicht möglich. Geschachtelt werden können die sogenannten View Container, welche jedoch lediglich View Components strukturieren und zusammenfassen. View Components und Container (kurz: View Elemente) können mit Events verknüpft werden. Treten diese ein, so wird der Zustand der Nutzerschnittstelle entsprechend eines modellierten Interaktionsflusses verändert. Dabei können Actions ausgelöst werden. In dieser Hinsicht sind daher Ähnlichkeiten zu UML-Zustandsdiagrammen vorhanden. In Bezug auf Datenhaltung und Datenflüsse liefert IFML gute zusätzliche Ansätze: So besitzen sowohl View Components als auch Actions Eingangs- und Ausgangsparameter. Abhängigkeiten zwischen den Ausgangs- und Eingangsdaten von View Elementen und Actions können mittels sogenannter Parameter Bindings annotiert werden.

Aus der obigen Analyse geht hervor, dass weder UML noch IFML die gesetzten Anforderungen zur Gänze erfüllen. Das Meta-Modell der DORI-DSL wird daher neu entwickelt, wobei sich an beiden Sprachen orientiert wird. Die konkrete Syntax wird soweit möglich aus UML Zustandsdiagrammen übernommen, da diese einen höheren Bekanntheitsgrad im Kontext der Softwareentwicklung besitzen. In begründeten Fällen werden hier aber auch Abweichungen von dieser Syntax zugelassen.

8.2. Festlegung der Betriebsplattformen des Editors

Hauke

In diesem Abschnitt werden die unterstützten Betriebsplattformen für den Betrieb des Editors vorgestellt beziehungsweise diese abgegrenzt. Eine wichtige Limitierung der Plattformen ist die grundlegende Aufgabe des DORI-Editors, die Modellierung von Interaktionen und die Erzeugung von plattformspezifischen Endanwendungen benötigt einen Leistungsbedarf, der mobile Betriebsplattformen von vorne herein ausschließt. Dies lässt als Auswahlobjekt vorrangig die gängigen PC-Plattformen übrig - Windows, MacOS und Linux. Die Projektgruppe hat sich, aufgrund der Systemverteilung unter den PG-Teilnehmern, für die gezielte Unterstützung von Windows 10 64 Bit und Linux Ubuntu 16.04 LTS und Debian ab der Version

8.0 entschieden. Die zur Auswahl stehenden Editor-Basistechnologien bieten zwar grundsätzlich das Potential zur Lauffähigkeit unter MacOS, allerdings wird es für diese Plattform keinen Support oder gesonderten Customizing-Aufwand geben.

8.3. Festlegung des Editor Tools

Nancy und Hauke

Eine wichtige Komponente des DORI-Projektes ist der Editor. Dieser Editor wird aufgrund der Wiederverwendbarkeit und der damit verbundenen Aufwandsreduzierung, nicht von grundauf selber von der Projektgruppe programmiert, sondern es wird eine Basistechnologie zur Erstellung eines Editors für unsere DORI-DSL verwendet. Die Basistechnologien wurden in einer Vorauswahl auf die zwei Tools MetaEdit+ und Sirius beschränkt. Eine weitere Technologie wurde in einer ersten Evaluierungsiteration mittels Prototypen bereits ausgeschlossen, dieses Tool war Meta Programming System (MPS), welches Limitierungen in der Umsetzung von grafischen Editoren aufwies und aus diesem Grund früh als Auswahl ausschied. Im Folgenden sollen die beiden Tools MetaEdit+ und Sirius beschrieben und anhand der Anforderungen evaluiert werden. Ebenfalls erfolgt eine Evaluierung der Benutzerfreundlichkeit, diese bezieht sich auf die Benutzerfreundlichkeit zur Erstellung des Editors. Anschließend soll eine begründete Entscheidung für eines dieser Tools getroffen werden. Diese Evaluierung fand auf Basis der Anforderungen vom 01.08.2017 statt.

8.3.1. MetaEdit+

Nancy

Im Folgenden soll das Tool MetaEdit+ vorgestellt werden. Dieses Tool bietet viele Funktionalitäten für die Erstellung einer eigenen DSL und die folgende Benutzung dieser in einem Editor. Die genauen Funktionalitäten werden in dem Kapitel 8.3.1 beschrieben. Anschließend erfolgt eine Evaluation dieses Tools gegen die im vorherigen beschriebenen Anforderungen. Diese Anforderungen sind vor allem funktionale Anforderungen.

Beschreibung des Tools

Seit 1991 gibt es die Firma MetaCase. Das Aushängeschild dieses Unternehmens ist das Tool MetaEdit+. Dieses Tool bietet die Möglichkeit, eine vordefinierte DSL einzubinden oder eine gewünschte DSL zu entwerfen und daraus einen Editor zu generieren. Außerdem stellt MetaEdit+ verschiedene Generatoren zum Erzeugen der gezeichneten Modelle in eine Form

wie zum Beispiel XML. Die Generatoren von MetaEdit+ sind hierbei nicht zu verwechseln mit dem DORI-Übersetzer.

MetaEdit+ lässt sich in zwei unterschiedliche Tools unterteilen. In der MetaEdit+ Workbench kann über die Definition verschiedener Objekte und die Beziehungen derer eine DSL definiert werden. MetaEdit+ Modeler ist das Tool, welches den aus der DSL generierten Editor darstellt. Im Folgenden sollen dieses Tool anhand eines konkreten Beispiels, welches auch für Sirius verwendet wurde, näher erläutert werden.

Für die Beschreibung der Tools wird das Meta-Modell einer Familie verwendet. Die Abbildung 8.1 zeigt dieses Meta-Modell.

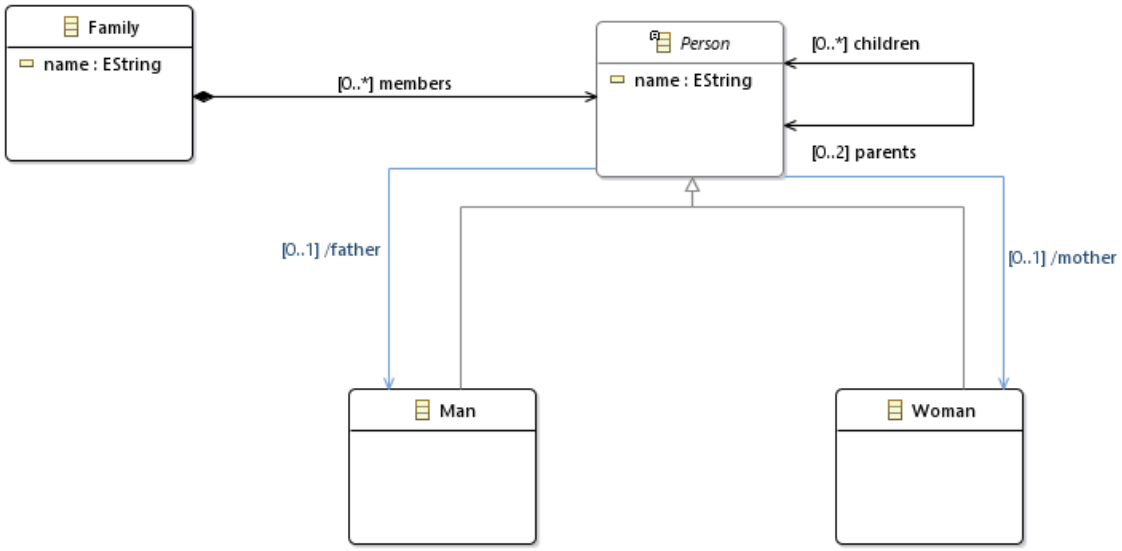


Abb. 8.1.: Meta Modell Familie

In diesem Modell sind verschiedene Elemente gegeben, wie Klassen, Assoziationen, Generalisierung, Multiplizitäten und abgeleitete Features. MetaEdit+ arbeitet allerdings nicht auf der Basis von UML, sondern auf auf der Basis von GOPRR. GOPRR steht für: Graph, Object, Property, Relationship, Role. Das bedeutet, dass für eine Eingabe des Meta-Modells die verschiedenen Elemente zerlegt und auf die Elemente von MetaEdit+ verteilt werden müssen. Somit stellen die Klassen in MetaEdit+ die Objekte dar. Wie die Abbildung 8.2 zeigt, kann zu jedem Objekt ein Name und ein Property angelegt werden. Die Properties sind in dem Beispiel die Attribute der Klasse oder des Objekts.

Über ein eingebautes Zeichen-Tool kann der Benutzer die konkrete Syntax seines Objektes angeben. Dies veranschaulicht die Abbildung 8.3

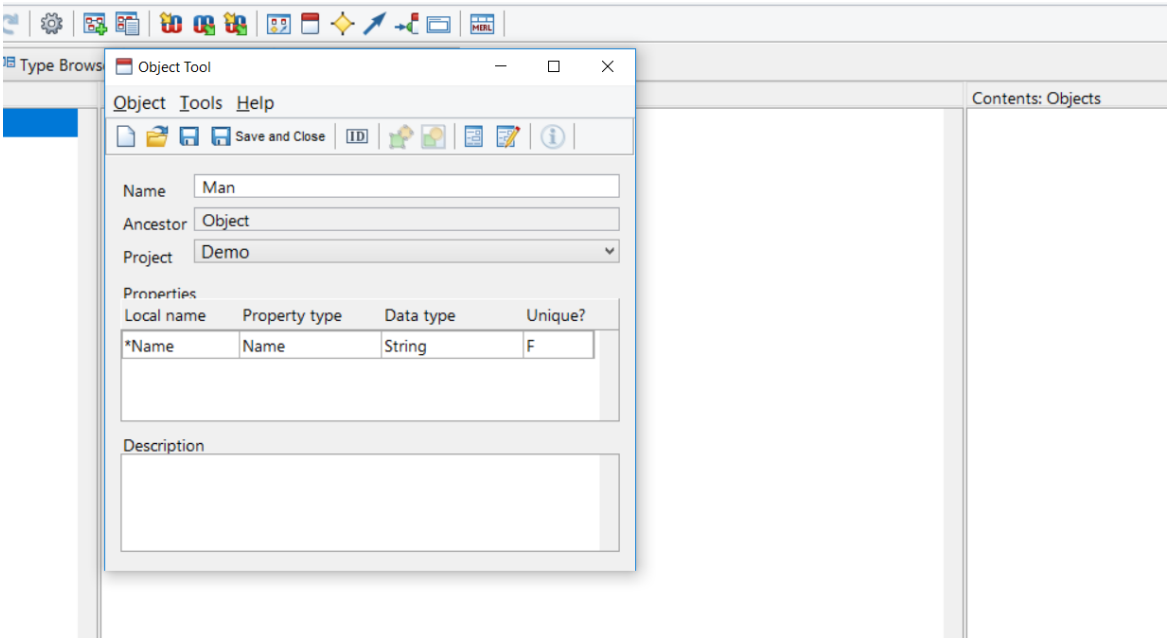


Abb. 8.2.: Objekt anlegen

Über das Definieren von Typen können die verschiedenen Assoziationen angelegt werden. Die konkrete Syntax von diesen wird allerdings in den Rollen festgelegt. Die Rollen sind wichtig für das Festlegen der Bindings. Die Bindings sind dafür da, um die Beziehungen zwischen den Objekten zu definieren. Wenn eine Beziehung zwischen zwei Objekten nicht definiert ist, können diese auch in dem Editor nicht in einem Modell festgelegt werden. Zu jedem Objekt muss ein Typ - sprich eine Assoziation - angegeben werden, sowie zwei Rollen. Sollte wie in dem Beispiel eine Generalisierung bestehen, muss auch diese über die Bindings mit definiert werden. Dieser Teil sorgte für die größte Umsetzungsdauer bei dem Tool, da die Angabe der Bindings, je größer das Modell wird, sehr lang werden kann.

Ist die DSL fertig definiert, kann der Editor generiert werden. Dieser stellt eine Toolbox zur Verfügung, welche die verschiedenen Elemente anzeigt, die verwendet werden können, wie zum Beispiel alle Objekte und Assoziationen. Diese können in dem Arbeitsbereich angelegt werden. Das Objekt wird in der zuvor angegebenen Syntax dargestellt. Dies bedeutet, dass der generierte Editor ein grafischer Editor ist. Durch die Generierung des fertigen Modells kann dieses in ein schriftliches Dateiformat gebracht und somit Berichte erstellt werden. Außerdem kann kein fehlerhaftes Modell erstellt werden, da durch die Bindings fehlerhafte Verbindungen von dem Tool nicht zugelassen werden.

MetaEdit+ stellt somit die Möglichkeit, eine eigenen DSL anzugeben und daraus einen Editor zu bauen, um Modelle nach der DSL zu erstellen. Allerdings ist die Verwendung von diesem Tool nicht kostenfrei. Eine Starter Lizenz für dieses Tool kostet einmalig 3.450 Euro.

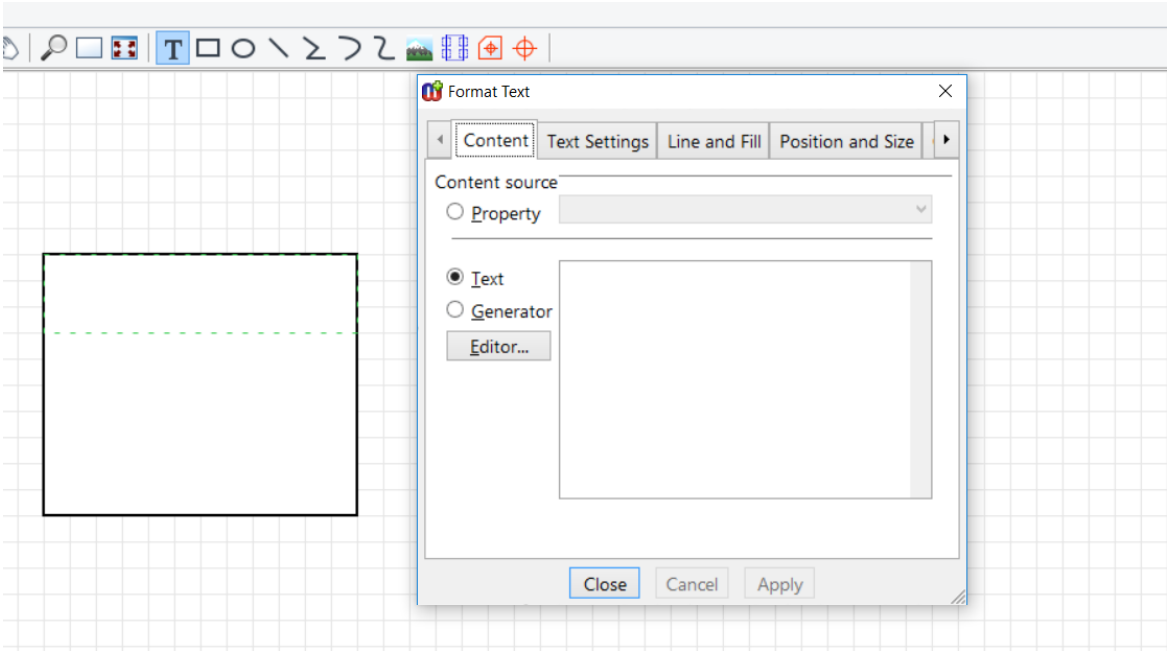


Abb. 8.3.: Festlegen konkrete Syntax

Eine Verwendung dieses Tools muss genau anhand der Anforderungen evaluiert werden. Diese Evaluation wird im Folgenden beschrieben.

Evaluation gegen die Anforderungen

Für die Evaluation gegen die Anforderungen werden nur die Anforderungen an den Editor betrachtet. Diese teilen sich in die folgenden Anforderungsgruppen auf:

- E-MDG: Modellierung der Gesamtinteraktion
- E-MSP: Modell speichern
- E-GIB: Gesamtinteraktion bearbeiten
- E-IKO: Inhalte in Katalogen organisieren
- E-KMP: Konsistenz des Modells prüfen
- E-MUU: Modell an Übersetzer übergeben
- E-SON: Sonstige Anforderungen

Diese Anforderungsgruppen werden einzeln betrachtet.

E-MDG: MetaEdit+ ermöglicht es ein Modell graphisch zu erstellen, wobei die Modellierungsmittel aus der definierten DSL entstammen. Hierbei kann sowohl ein neues Modell erstellt werden als auch ein bereits vorhandenes Modell geladen und weiter bearbeitet werden. Somit deckt der Editor die Anforderungen dieser Anforderungsgruppe komplett ab.

E-MSP: MetaEdit+ ermöglicht es ein Modell manuell zu speichern. Dabei kann diese Datei an einem neuen Ort und unter einem neuen Namen gespeichert werden oder ein bereits gespeichertes Modell ersetzen. Das Dateiformat ist, mittels einem integriertem Generators, der selber definiert werden muss, frei wählbar. Somit ist eine Versionierung des Modells, wenn ein textuelles Dateiformat gewählt wurde, möglich. Beim Schließen des Editors wird der Nutzer gefragt, ob er das Modell speichern möchte sofern er dies vorher nicht manuell gemacht hat. Möchte der Nutzer dies nicht schließt sich der Editor ohne zu speichern. MetaEdit+ erfüllt diese Anforderungen dementsprechend.

E-GIB: Für die abstrakte Syntax der DORI-DSL wird von der Projektgruppe ein Meta-Modell erstellt. Dieses Modell lässt sich mittels verschiedenen Bindings und Erstellung von Objekten in das Tool einpflegen. Zu jedem Objekt kann die konkrete Syntax angegeben werden. Sind diese beiden Sachen definiert, kann das Graphen-Tool verwendet werden, welches zur Modellierung eines Modells, auf Basis der DSL, dient. Somit werden alle Anforderungen der Anforderungsgruppe von MetaEdit+ abgedeckt.

E-IKO: In MetaEdit+ ist es möglich verschiedene Sichten der DSL anzeigen zu lassen. Somit können abstrakte und konkrete Inhalte zusätzlich zu den Modellierungsmitteln angezeigt werden. Die Kataloge sind nicht in der DORI-DSL definiert. Somit ist es auch nicht möglich diese in dem Editor anzuzeigen, dies gilt jedoch ebenso für Sirius. Mittels eines externen Tools müssen somit die Kataloge umgesetzt und verwaltet werden. Des Weiteren müssen Schnittstellen für dieses externe Tool geschaffen werden. Es bildet sich eine Tool-Chain. Dieses ist von der Projektgruppe nicht präferiert.

E-KMP: MetaEdit+ bietet eine Echtzeit-Konsistenzprüfung. In dem Arbeitsbereich können keine Modelle modelliert werden, die nicht der Semantik der DSL entsprechen. Sollte eine Modellierung nicht möglich sein zeigt MetaEdit+ dies an. Somit wird es ausgeschlossen, dass ein fehlerhaftes Modell erstellt werden kann. Allerdings prüft das Tool nicht, ob es die angegebenen konkreten Inhalte auch gibt, da diese sich in einem anderen Tool befinden würden. Somit werden nicht alle Anforderungen dieser Anforderungsgruppe abgedeckt.

E-MUU: MetaEdit+ bietet bereits Generatoren für verschiedene Zielplattformen. Weiterhin können eigens definierte Generatoren hinzugefügt werden. Der Export des Modells mittels Generatoren bildet dabei die gewünschte Schnittstelle in Basisform.

E-SON: Das Installationsanleitung und das Benutzerhandbuch wird von der Projektgruppe erstellt und ist somit losgelöst von dem Tool. Die gängigen Designrichtlinien und Richtlinien

der Software-Egonomie werden in einer Form von MetaEdit+ eingehalten, können aber nicht von der Projektgruppe geändert werden, da der Quellcode des Tools nicht frei zugänglich ist. Die Berichte können durch einen Generator erstellt werden, indem die Zielplattform zum Beispiel ein Word-Dokument ist. Diese Berichte beziehen sich vor allem auf die abstrakten Inhalte, um den GUI- und Programmlogik-Entwickler über die zu implementierenden Widgets und Funktionen zu berichten.

Fazit: Die Evaluation zeigt, dass MetaEdit+ viele der Anforderungen umsetzen kann. Dennoch werden nicht alle in der Form umgesetzt, wie die Projektgruppe es sich wünscht. Eine Entscheidung für dieses Tool bezieht somit auch die nicht-funktionalen Anforderungen mit ein, die im Folgenden betrachtet werden.

Evaluation der Benutzerfreundlichkeit

Die Benutzerfreundlichkeit bezieht sich bei dieser Evaluation auf die Anwendung des Tools zur Erstellung des Editors. Hierbei wird Benutzerfreundlichkeit des Endbenutzers des Editors ausgeklammert, da dies durch die Erfüllung der sonstigen Anforderungen bereits gegeben sein soll. Die sonstigen Anforderungen wurden bei der Evaluation hinsichtlich der gestellten Umsetzungsmöglichkeiten der Tools überprüft.

Ein wichtiger Bestandteil der Benutzerfreundlichkeit ist die Lernkurve für das Tool. Die Erfahrungen der Projektgruppenmitglieder bezieht sich auf UML Modellierung. Bei MetaEdit+ ist eine Einarbeitung in die Modellierungssprache GOPRR nötig. Dies erhöht den Aufwand für die Erarbeitung der benötigten Kompetenzen und ist eine Bedingung für die Benutzung des Tools.

Um alle Anforderungen zu erfüllen, müssen außerdem weitere Tools neben MetaEdit+ verwendet werden. Eine Einarbeitung in die anderen Tools erhöht auch den Lernaufwand. Außerdem wird der Aufwand durch die Implementierung der Schnittstellen erhöht.

Ein Vorteil von MetaEdit+ ist die Definition der konkreten Syntax der DSL. Diese kann entweder selber gezeichnet oder durch eine importierte Grafik erstellt werden. Bei Sirius hingegen ist ein Import von Grafiken möglich, außerdem ist ein Editor mit vordefinierten Formen vorhanden. Ein freihändiges Zeichnen von eigener Syntax ist bei Sirius nicht möglich.

Die Erstellung der DSL selber erfolgt bei MetaEdit+ in tabellarischer Form. Durch die Bindings können die Beziehungen der verschiedenen Objekte dargestellt werden. Je komplexer das Meta-Modell der DSL wird, desto schwieriger wird die Definition der Bindungs. Die Lernkurve für diese Erstellung ist somit sehr hoch. Die Darstellung ist außerdem nicht trivial. Dieses zeigt auch das Paper „Evaluation of Modellierung Tools Adaption“ [7]. Dort testen Experten dieses Tool und grenzen dieses gegen andere Tools ab. Die Benutzerfreundlichkeit von MetaEdit+ schneidet dabei schlechter ab als die anderen Tools. Das Kompetenzlevel der

Mitglieder der Projektgruppe ist geringer, als das der Wissenschaftler und Experten aus dem Paper. Der Support für Fragen im Zusammenhang mit MetaEdit+ ist dabei gebündelt auf die Angebote von Meta Case. Dabei ist der Support gut und schnell und dennoch limitiert in der Quantität. Somit ist die Benutzerfreundlichkeit aus unserer Sicht deutlich eingeschränkt.

8.3.2. Sirius

Hauke und Nancy

Im Folgenden soll das Tools Sirius vorgestellt werden. Dieses Tool basiert auf der IDE Eclipse und ist als Open Source Projekt ausgelegt. Somit ist es frei zugänglich und bietet viele Anpassungsmöglichkeiten für die Erstellung eines Editor-Tools auf Basis der DORI-DSL. Im Folgenden werden die Funktionalitäten des Tools erläutert und anschließend gegen die Anforderungen evaluiert. Die Benutzerfreundlichkeit wird, wie bereits bei MetaEdit+, auch für Sirius betrachtet.

Beschreibung des Tools

Im diesem Abschnitt wird das Tool Sirius näher vorgestellt. Sirius ist ein Eclipse-Projekt zur Erstellung von grafischen Modellierungswerkzeugen. Es ist anzumerken, dass mit Eclipse-Projekt nicht die Projekte in der Eclipse IDE, sondern Entwicklungsprojekte gemeint sind. Sirius basiert auf Entwicklungen der Firmen Obeo und Thales, hierbei stellt Sirius die Open Source-Variante von Obeos Obeo Designer dar, welcher kommerziell erworben werden kann. Sirius setzt auf verschiedene Elemente, initial werden dem Entwickler Editoren für Diagramme, Tabellen und Bäume bereit gestellt. Diese Elemente werden auf Basis der Technologien des Eclipse Modelling Frameworks (EMF) bereit gestellt. Das EMF ist ein weiteres Eclipse-Projekt zur Datenmodellierung und Erstellung von Editoren und wird von xText verwendet. Die Abbildung 8.1 zeigt das für eine erste Vorauswahl verwendete Meta-Modell einer einfachen Familie als EMF Ecore-Modell.

Im nächsten Schritt lässt sich ein zuvor erstelltes EMF Ecore-Modell zur Definierung eines Editortools verwenden - hierbei ist ein Metamodell einer DSL dementsprechend das strukturierende Element eines mit Sirius umgesetzten Editors. Im weiteren Verlauf lässt sich nun der grafische Editor und seine abzubildenden Elemente genauestens anpassen und umsetzen. Für die Gestaltung eines Editors werden in einem separatem Projekt Sichten auf das Meta-Modell definiert. In dieser Sicht wird die grundsätzliche Visualisierungstechnik für eine Sicht auf ein Modell festgelegt, hier lassen sich zum Beispiel Diagramme oder Tabellen als Darstellung festlegen. Sirius bietet mehrere Möglichkeiten für die Anpassung des Editors an die eigenen Wünsche. So bekommen die Elemente eines Metamodells die konkrete Syntax des Metamodells zugewiesen, wie auch die Beziehungen zwischen den Elementen. Für die

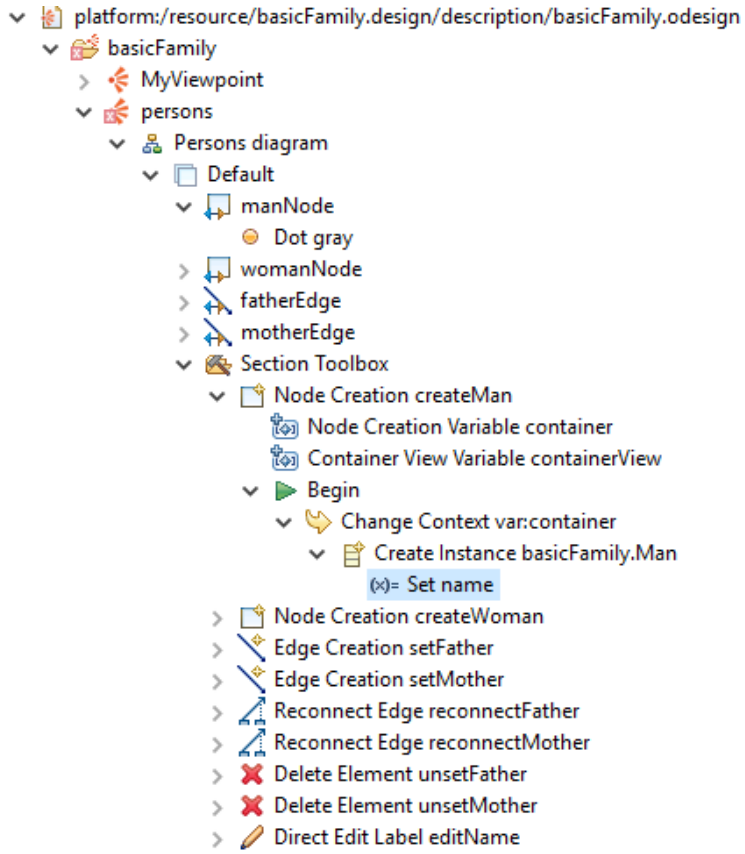


Abb. 8.4.: Sichtendefinition in Sirius

spätere Benutzung des Editors müssen und können alle Funktionalitäten, welche bei der Interaktion mit dem Editor und zur Erstellung eines Modells von Nöten sind, definiert werden. Dies geht von der Erzeugung von Elementen, im Arbeitsbereich, über das Entfernen von Elementen, aus dem Arbeitsbereich, bis zur Umsetzung einer Konsistenzprüfung. Abbildung 8.4 zeigt die Sichtendefinition zum verwendeten Metamodell.

In einem neuen Modellierungsprojekt lässt sich diese Sicht mit dem zugrundeliegenden Metamodell nun einbinden und verwenden. Anschließend lässt sich eine gewünschte Repräsentation eines Modells erstellen. Abbildung 8.5 zeigt exemplarisch den Arbeitsbereich, welcher das erstellte Modell abbilden kann. Es ist ein Arbeitsbereich zu sehen, mit einer kleinen Modellierung. Diese folgt der im Viewpoint festgelegten konkreten Syntax. Am rechten Rand ist die Toolbox mit den instanzierbaren Elementen des Metamodells.

Sirius bietet sowohl die Möglichkeit den erstellten Editor als Standalone-Tool als auch als Ecplise Plugin umzusetzen. Die vorgestellten Eigenschaften müssen allerdings gegen die Anforderungen der Projektgruppe evaluiert werden, dies erfolgt im nächsten Abschnitt.

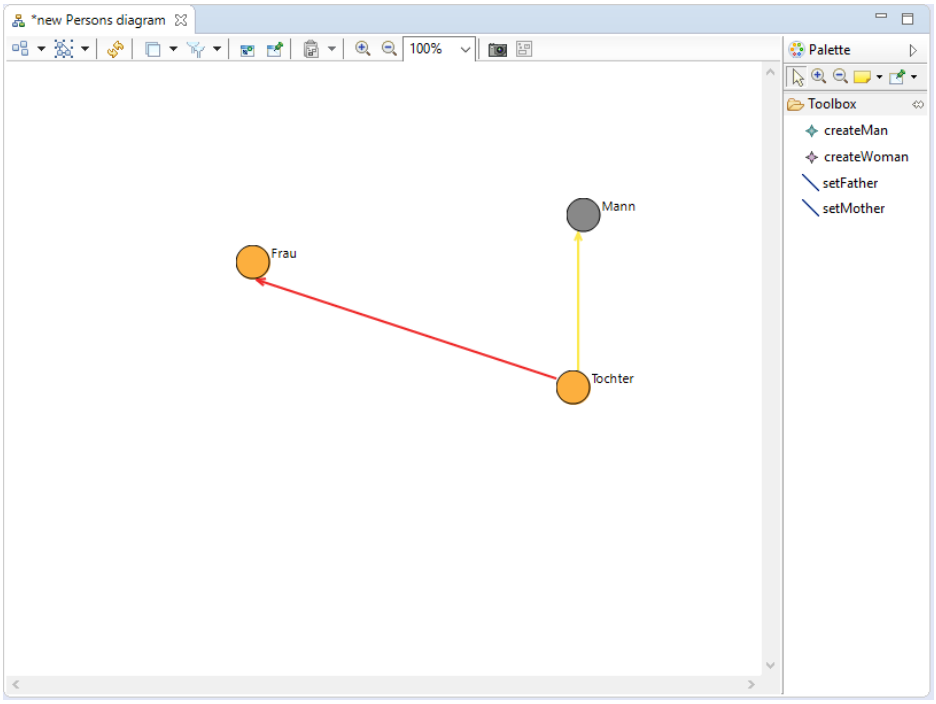


Abb. 8.5.: Beispiel eines Editors in Sirius

Evaluation gegen die Anforderungen

Für die Evaluation gegen die Anforderungen werden nur die Anforderungen an den Editor betrachtet. Diese teilen sich in die gleichen Anforderungsgruppen wie bei MetaEdit+ auf:

- E-MDG: Modellierung der Gesamtinteraktion
- E-MSP: Modell speichern
- E-GIB: Gesamtinteraktion bearbeiten
- E-IKO: Inhalte in Katalogen organisieren
- E-KMP: Konsistenz des Modells prüfen
- E-MUU: Modell an Übersetzer übergeben
- E-SON: Sonstige Anforderungen

Diese Anforderungsgruppen werden auch hier wieder einzeln betrachtet.

E-MDG: In dem Sirius Editor werden die Sprachmittel der definierten DSL angezeigt und können für die Modellierung eines Modells verwendet werden. Dabei kann ein neues Modell

erstellt werden sowie ein Erstelltes geöffnet und bearbeitet werden. Sirius erfüllt somit alle Anforderungen dieser Anforderungsgruppe.

E-MSP: Sirius bietet einen Button, mit diesen ein manuelles Speichern möglich ist. Dabei wird das Modell in dem Ordner Workbench abgelegt. Der Name der Datei entspricht dabei dem Namen des Modells. Bei einer Änderung des Modells und anschließender Speicherung kann das Modell umbenannt und in einer neuen Datei gespeichert werden oder aber die vorhandene Datei ersetzen. Somit ist eine Versionierung des Modells in einem textuellen Format möglich. Andere Speichermöglichkeiten können durch eigene Implementationen hinzugefügt werden. Der Editor fragt außerdem beim Schließen ob das nicht gespeicherte Modell gespeichert werden soll. Auch ist ein Schließen ohne Speichern möglich. Somit werden die Anforderungen alle erfüllt.

E-GIB: Die DORI-DSL kann mittels eines EMF Ecore Modells als Basis eines Editor-Tools umgesetzt werden. Dies bedeutet, dass das mit dem RSA erstellte Metamodell in Form eines Ecore Modells abgespeichert und in Sirius geladen werden kann. Alternativ kann grafisch das Metamodell modelliert werden. Die konkrete Syntax kann dabei frei definiert werden. Es existieren Standardformen und es können eigenen Grafiken eingebunden werden. Eine Bearbeitung der Grafiken in dem Tool selber ist nicht möglich. Die Verwendung der konkreten und abstrakten Syntax ist durch die Abdeckung der Anforderungen der Anforderungsgruppe E-MDG gegeben.

E-IKO: Dadurch, dass Sirius auf Eclipse basiert können verschiedene Tools eingebunden werden. Außerdem ist der Quellcode frei verfügbar und somit eine Einbindung von Katalogen, die nicht in der DORI-DSL stehen, möglich. Die Verwaltung der Kataloge kann außerdem auch selber definiert werden. Es bildet sich keine Tool-Chain mit separaten, externen Tools, da alle Tools direkt, als eigenständiges Plugin, in die Eclipse Basis eingebunden werden können.

E-KMP: Eine Konsistenzprüfung ist Standardmäßig nicht in Sirius implementiert. Dennoch ist ein Hinzufügen einer Prüfung durch die vielfältigen Anpassungsmöglichkeiten möglich. Dazu wird ein Validierungssystem geboten, welches nur noch eingebunden werden muss. Das Grundgerüst dieses Validierungssystems ist bereits gegeben und müsste von der Projektgruppe nur noch mit Bedingungen ausformuliert werden.

E-MUU: Es besteht die Möglichkeit mittels einer Schnittstelle, die modellierte Gesamtinteraktion an den Übersetzer zu übergeben. Das Beziehungsgeflecht ist Standardmäßig in XML gespeichert. Die Schnittstelle kann direkt in dem Tool umgesetzt werden, kann aber auch zu ein externen Tool exportiert werden.

E-SON: Wie bereits bei MetaEdit+ beschrieben ist die Erstellung eines Installations- und Benutzerhandbuches unabhängig von dem Tool. Die gängigen Richtlinien für Design und Software-Ergonomie werden von dem Tool umgesetzt. Eine Änderung dieser ist bei diesem

Tool möglich. Eine Berichterstellung (wie in MetaEdit+ beschrieben) müsste eigenständig implementiert werden.

Fazit: Während Sirius nicht standardmäßig alle Anforderungen erfüllt, können diese jedoch über eigene Implementierungen erfüllt werden. Es ist eine genaue Anpassung der Tools an die Vorstellungen der Projektgruppe möglich und somit erfüllt dieses Tool alle Anforderungen an den Editor.

Evaluation der Benutzerfreundlichkeit

Wie bereits bei MetaEdit+ beschrieben wird hier die Benutzerfreundlichkeit auf die Entwicklung des Editor beschränkt. Dabei spielt der Lernaufwand sowie Anpassbarkeit des Tools eine wichtige Rolle. Die Benutzerfreundlichkeit für den potentiellen Endbenutzer wird durch die Erfüllung der sonstigen Anforderungen grundlegend abgedeckt.

Die Basis von Sirius sind Klassendiagramme, die dem Ecore-Standard folgen. Mit diesen Diagrammen ist die Projektgruppe bereits in Kontakt gekommen und somit ist die Lernkurve sehr gering. Ein Vorteil dieses Tools ist der leichte Datenaustausch von verschiedenen Modellen auf Basis von UML-Klassendiagrammen. Dieses geschieht über XML und XMI.

Wenn noch kein Modell in einem anderen UML-Tool erstellt wurde, ist das auch in Sirius möglich. Dies geschieht grafisch und die Oberfläche gleicht die von anderen Editoren. Somit funktioniert die Erstellung des Editors ähnlich wie die Verwendung eines Editors und dies senkt den Aufwand für das Lernen.

Für die Erstellung der konkreten Funktionalitäten für das Erzeugen und in Beziehung setzen von DORI-DSL-Elementen, muss sich in die Herangehensweise eingearbeitet werden. Dieses Prinzip zieht sich durch das ganze Modell. Der Umgang mit dem Editor erfordert wie in jedem anderen Tool einen Einarbeitungsaufwand.

Die einzige neu zu lernende Komponente stellt die Objekt Constraint Language (OCL) dar, die dazu benötigt wird, durch das Modell zu iterieren. Diese Informationen sind jedoch leicht für die Projektgruppe zugänglich, da diese Themen in den Seminaren der Softwaretechnik der Universität Oldenburg behandelt werden.

In dem zuvor bereits angesprochenen „Paper Evaluation of Modeling Tools Adaptation“ [7] wird das Sirius Tool nicht direkt angesprochen. Allerdings wird der OBEO Designer als Tool betrachtet. Dieser stellt ein freies „All-in-One“-Paket mit Sirius dar, welches von den Firmen Obeo und Thales angeboten wird. Zusätzlich ist zu erwähnen, dass die Benutzerfreundlichkeit auf Basis der zugrundeliegenden Technologien GOPRR und EMF bewertet wurde. Dabei schneidet der Obeo Designer mit Sirius und EMF in der Benutzerfreundlichkeit besser als MetaEdit+ mit GOPRR ab. Der Support bietet ebenfalls professionelle Hilfsmittel, wobei

die Community im Bereich Eclipse Plugin Entwicklung sehr groß ist. Als negativ anzumerken bleibt nur die Erstellung des Editors selber. Während bei MetaEdit+ ein Generator diesen erstellt, muss bei Sirius der Editor selber als Eclipse Plugin oder Standalone-Tool umgesetzt werden, was eine bestimmte Einarbeitungszeit benötigt.

Festzuhalten bleibt, dass Sirius viele Funktionalitäten bietet, die eine geringe Einarbeitung benötigen. Zusätzlich ist es möglich für fehlende Funktionalitäten, durch den weitreichenden Support, eine schnelle Einarbeitung für Lösungsansätze zu finden.

8.3.3. Gegenüberstellung der Tools

Hauke und Nancy

Grundsätzlich lassen sich die beiden Tools zum einen an der Erfüllung der Anforderungen und zum anderen an der Bedienung an sich vergleichen.

MetaEdit+ erfüllt die Anforderungsgruppe E-IKO nicht nach den Wünschen der Projektgruppenmitglieder. Außerdem wird die Anforderungsgruppe E-KMP nur zum Teil abgedeckt. Für die Erstellung und Verwaltung von Katalogen muss bei MetaEdit+ ein separates Tool angebunden werden. Auch Sirius bietet nicht standardmäßig diese Funktionalität, diese lässt sich aber durch verschiedene Plugins einfügen oder gegebenenfalls implementieren.

Aus Sicht der aufgestellten Anforderungen kann bei MetaEdit+ nicht sicher gestellt werden, dass die Eindung und Verwaltung von Katalogen in dem Editor möglich ist. Bei Sirius jedoch lassen sich diese leichter einbinden, da es sich mittels Ecplipse Plugin umsetzen lassen würde. Diese Technologie beziehungsweise dessen Umsetzung ist im Internet relativ ausführlich dokumentiert. Somit erfüllt Sirius alle Anforderungsgruppen.

Neben der Erfüllung der Anforderungen bieten beide Tools ihre Vor- und Nachteile in der Benutzung. Ein großer Vorteil von Sirius ist die direkte Importierung des im RSA erstellten Metamodells der DORI-DSL und die Verwendung derselben Modellierungssprache. Die Anpassung des Modells selber kann mittels einer graphischen Modellierungsoberfläche erfolgen, während die von MetaEdit+ textuell ist. Ein Import von einem Metamodell in MetaEdit+ ist nur durch einen Generator möglich, welcher von der Projektgruppe selbst geschrieben werden müsste. Für diesen muss die Programmiersprache MERL, zusätzlich zu der Modellierungssprache GOPRR, gelernt werden. Grundlegend ist für Sirius der Lernaufwand von, benötigten, UML-Sprachen und OCL auch gegeben - nur existieren hier bereits grundlegende Vorkenntnis in der Projektgruppe.

Ein Vorteil von MetaEdit+ ist dafür die direkte Bearbeitung der konkreten Syntax in dem Tool selber, während bei Sirius diese extern erfolgt.

Zur Berücksichtigung bei der Gegenüberstellung sind auch Gegebenheiten aus der Arbeit innerhalb der Projektgruppe. Das Metamodell wird aktuell mittels UML-Klassendiagrammen erstellt. So entsteht für MetaEdit+ definitiv ein Transferaufwand für die Überführung in GOPRR mittels Importierung oder händischem Transfer. Des Weiteren dürfte sich die Handhabung von Constraits in MetaEdit+ schwieriger gestalten als in Sirius, wo eine Umsetzung mittels OCL-Constraits in einem Ecore-Modell verwendet werden könnte. Abschließend ist die Anpassungsfähigkeit einer Konsistenzprüfung für die Projektgruppe wichtig, dies lässt sich bei MetaEdit+ nicht gewährleisten, da sie standardmäßig umgesetzt und verwendet wird.

Ein Nachteil zeigt sich in dem Lizenzmodell von MetaEdit+. Für jedes Metacase-Tool muss eine teure Lizenz beschafft werden. Dies stellt eine Hürde für die Verwendung innerhalb und außerhalb der Projektgruppe dar. Sirius hingegen ist ein Open Source Projekt und somit für jeden frei zugänglich. Das Sirius Plugin ist über den Eclipse Martkplatz beziehbar und auf der Projektseite auch als gezippte Datei abrufbar.[1] Zusätzlich lässt sich das Open Source Projekt per Git clonen.[2] Wie in Kapitel 7.2 anhand der Faktoren zu sehen ist stellen Punkte, wie die Weiterarbeit am Produkt und Kosten einen wichtigen Punkt in der Entscheidungsfindung dar.

Begründung der Entscheidung

Die Projektgruppe hat sich am 10.08.2017, nach einer ausführlichen Diskussion, für das Tool Sirius entschieden. Die Entscheidung wurde mit fünf Stimmen dafür und zwei Enthaltungen getroffen. Gegenstimmen gab es nicht und zwei Mitglieder waren im Urlaub. Der Grund für die Entscheidung war die des Aufbaus von Sirius auf UML-Klassendiagrammen. In diesem Bereich liegen die größten Grundkenntnisse der Projektgruppenmitglieder. Außerdem wird die DORI-DSL in einem UML Metamodell beschrieben. Dieses kann somit direkt in dieses Editor-Tool importiert werden. Dies wäre mit MetaEdit+ nicht möglich. Ein weiteres wichtiges Argument für Sirius ist die flexible Anpassungsmöglichkeit des Tools. Die Umsetzung der DORI-DSL in diesem Tool erfolgt in dem Sprint zwei.

8.4. Auswahl der Zielplattform

Hauke und Nancy

Nachdem in den vorherigen Kapiteln die Anforderungen an das DORI-System aufgestellt wurden, behandelt dieser Abschnitt die Auswahl einer geeigneten Zielplattform für die erste

[1] http://www.eclipse.org/sirius/download.html
[2] git://git.eclipse.org/gitroot/sirius/org.eclipse.sirius.git

prototypische Implementierung des DORI-Übersetzers. In dieser Auswahl wird nur über die Plattform selbst und nicht über ein konkretes Framework entschieden.

Im Allgemeinen können alle Plattformen in drei Kategorien eingeordnet werden. Die erste Kategorie besteht aus dem klassischen PC beziehungsweise Laptop. Die zweite Kategorie ist die der mobilen Endgeräte. Hierunter fallen zum Beispiel alle Smartphones, Tablets und Smartwatches. Wird der Grenze dieser Kategorie ein wenig aufgeweicht, so können auch die Smart-TVs und Infotainment-Systeme mit zu dieser Kategorie gezählt werden, da diese von der Struktur große Ähnlichkeiten aufweisen. Die dritte Kategorie umfasst alle Web-Anwendungen, somit all diejenigen, die von einem Browser aufgerufen werden.

Für die Umsetzung eines DORI-Übersetzers für den klassischen PC spricht die vorhandene Wissensbasis innerhalb der Projektgruppe, denn jeder Student der Informatik hatte, hat bereits eine fundierte Ausbildung in der Softwareentwicklung mit Java. Wird der Ansatz einer Java-Anwendung weiter verfolgt, so ergibt sich auch ein weiterer Vorteil, die Reichweite solcher Anwendungen ist fast uneingeschränkt, da diese unter vielen Betriebssystemen eingesetzt werden können.

Soll der DORI-Übersetzer für mobile Plattformen entwickelt werden, so ist hier der Vorteil, dass sich aus diversen Gruppengesprächen schon Experten herauskristallisiert haben. Diese Experten haben bereits Erfahrungen in der Android-Entwicklung. Hierbei haben die einzelnen Experten jedoch mit unterschiedlichen Frameworks gearbeitet, was dann wieder zu einer erneuten Einarbeitung in das gewählte Framework erfordert. Bei der Entwicklung für mobile Systeme werden die Betriebssysteme iOS und Windows Phone sowie kleine Betriebssysteme ausgeschlossen, da diese zum Teil mit Kosten verbunden sind oder deren Verbreitung zu gering für ein beachtenswertes System ist.

Die zuvor genannten Vorteile passen alle auf die dritte Kategorie der Plattformen, den Web-Anwendungen. Hinzu kommt noch, dass in der Masterarbeit von Timo [12] die selbe Plattform gewählt wurde und auch schon erste Erfolge verzeichnet wurden. Weiterhin unterliegt diese Kategorien keinen technischen Einschränkungen für den späteren Endanwender, wie zum Beispiel der Installation der Endanwendung. Hinzu kommt noch, dass die Rechenleistung beim Endnutzer sehr gering ist, da dessen Gerät lediglich als Anzeigemedium dient.

Zusammengefasst wurde die Entscheidung getroffen, dass die erste prototypische Implementierung des DORI-Übersetzers eine für Web-Anwendungen ist. Diese Entscheidung wurde getroffen, da die vorhandene Masterarbeit den gleichen Ansatz verfolgt. Angelehnt an dieser Masterarbeit kann auch für diese Projektgruppe ein Java-basiertes Hintergrundsystem entwickelt werden, welches die einzelnen Endgeräte nur noch ansprechen müssen, um die Anwendung dann darzustellen.

In dem weiteren Verlauf der Projektgruppe sollte eine zweite Plattform umgesetzt werden. Die Gruppe hat sich für Android entschieden, da Teile der Übersetzer-Technologie wieder-

verwendet werden konnten. Außerdem konnte ein Android-Übersetzer ebenfalls mit Java umgesetzt werden. Die genaue Beschreibung der Entscheidung für einen konkreten Übersetzer ist im Folgenden zu finden.

8.5. Festlegungen Übersetzer

Felix, Thomas, Stephan

In diesem Kapitel werden die Möglichkeiten vorgestellt, auf welche Arten der Übersetzer umgesetzt werden kann. Wie bereits im Abschnitt 8.4 erwähnt, wird als erstes ein Übersetzer entwickelt, der eine Interaktion mit entsprechenden GUI-Masken und Programmlogik in eine Webanwendung übersetzt. Dafür werden die drei untersuchten Möglichkeiten vorgestellt und jeweils die notwendigen Arbeiten aufgelistet und die Vor- und Nachtteile der Lösung vorgestellt.

8.5.1. Generator

Felix, Thomas, Stephan

Die Idee des Generators ist es, dass ein Programm alle Dateien schreibt, die notwendige sind, um eine Endanwendung zu kompilieren.

Da sich bereits in der ersten Phototypenphase mit Angular beschäftigt wurde, wurde dieses als mögliches Zielframework untersucht. Weiterhin ist Angular insofern interessant, dass es mit der NativeScript Erweiterung nicht nur als Webanwendung, sondern auch auf iOS und Android läuft.

Eine Tool Chain für einen Angular Generator würde wie folgt aussehen: Das Modell einer Gesamtinteraktion wird in ein Java-Programm eingelesen. Dieses Java-Programm generiert alle Type-Script Dateien und danach werden diese Dateien und alle Abhängigkeiten mit NPM(Node-Package Manager) zu der Endanwendung kompiliert.

Notwendige Arbeiten

- Das DORI-Metamodell muss verstanden und ein entsprechendes Java Handling implementiert werden.
- Angular und TypeScript(TS) müssen erlernt werden
- Schnittstelle zum Einladen des Modells muss entwickelt werden

- Angular ist ein komplexes Framework bei dem Generieren von Code gibt es folgende Unterpunkte zu beachten:
  - Für alle Datentypen müssen TS-Klassen erstellen werden.
  - Services für Programmlogik Calls müssen erstellt werden.
  - Für jedes abstrakte Widget muss eine Komponente erstellt werden
  - Komponenten, Klassen, Services etc. müssen verbunden werden
  - Export/Import Pfade und Dependency Injection in allen Dateien müssen vollständig und korrekt aufgelöst werden.
  - Programmlogikaufrufe in den unterschiedlichen Komponenten müssen richtig zugeordnet werden.

Vorteile

- Theoretisch ließe sich eine gleich gute Performance im Vergleich zu einer von Hand entwickelten Anwendung erreichen.
- Mit NativeScript ist ein Übersetzung zu iOS und Android möglich
- GUI Definition in Angular ist extrem gut kapselbar.

Nachteile

- Das Auflösen von vielen Dependencies, Imports und Exports führt zu einer hohen Komplexität
- Es muss die für alle PG-Mitglieder eher unbekannte Programmiersprache Typescript gelernt werden.

8.5.2. Interpreter JAVA

Felix, Thomas, Stephan

Der Grundgedanke hinter dieser Art des Übersetzers ist es, dass ein Programm entwickelt wird, welches ein Dori-Modell entgegen nimmt und dann alle notwendigen Entscheidungen anhand dieses trifft. Im Gegensatz zum zuvor genannten Generator wird das Dori-Modell zur Laufzeit interpretiert und nicht nur einmalig.

In der Prototypenphase wurde bereits mit JSF (Java Server Faces) experimentiert. Dieses ist ein Framework zur Entwicklung von Java-Backends für Webanwendungen. Somit ist dieses eine erste Wahl für die zuvor festgelegte Zielplattform. Hingegen zum Angular-Framework ist jedoch keine Portierung zu anderen Plattformen möglich.

In diesem Ansatz wird eine zentrale Serverinstanz entwickelt, welche dann von einem Web-Browser angesprochen werden kann. Dieser erhält dann, nach den einzelnen Anfragen, die üblichen fertig gerenderten HTML-Seiten, welche nur noch angezeigt werden müssen.

Notwendige Arbeiten

- Das DORI-Metamodell muss verstanden und ein entsprechendes Java Handling implementiert werden.
- JSF muss gelernt werden
- Entsprechender Webserver muss aufgesetzt werden.
- Schnittstelle zum Einladen des Modells muss programmiert werden.
- Auswahl der Aktionen auf Grund des Modells muss implementiert werden.
- Eine Schnittstelle zur Programmlogik muss implementiert werden.
- Da Java eine kompilierte Sprache ist aber trotzdem jedes DORI-Modell neue Klassen für die Datenhaltung braucht, müssen dynamische Klassen nachgeladen werden. Dies geschieht, indem aus composite data types Java Klassen generiert werden, um anschließend Instanzen von diesen erstellen zu können.

Vorteile

- Eher geringe Komplexität
- 'Reines' Java
- GUI Definition extrem gut kapselbar

Nachteile

- Es steht zu erwarten, dass die Performance im Vergleich zu Generator schlechter ist. Dies liegt daran, dass jeweils zur Laufzeit berechnet werden muss, wie sich die Interaktion verhält.
- Eher schlecht mögliche Anpassung an andere Plattformen. JSF bietet nur eine Webanwendung.
- Kompilierte Sprache wird 'dynamisch' verwendet.

8.5.3. Timos Interpreter

Felix, Thomas, Stephan

Diese Idee des Interpreters ist von der „Masterarbeit von Timo“ inspiriert. Dieser Interpreter besteht aus zwei Teilen. Der eine ist, wie auch beim einfachen Interpreter, ein zentrales Backend. Von diesem können alle Informationen wie der Interaktions- und Datenfluss sowie die Daten selbst angefragt werden. Der zweite Teil ist ein clientseitiges Programm. Dieses fragt die soeben genannten Informationen beim dem Backend ab und reichert diese Informationen mit plattformspezifischen Inhalten an. Somit gibt es etwa einen Webanwendungsclient, der dann die Daten vom Backend in eine Web-Anwendung übersetzt. Außerdem könnte es nötig sein, einen weiteren Server zu haben, wo die Plattformspezifischen Dateien liegen, die das clientseitige Programm zur Laufzeit anfordert.

Diese Aufteilung in Microservices führt zu einem erhöhten Kommunikationsaufwand zwischen den Programmen aber ermöglicht eine sehr schnelle Portierung auf andere Plattformen. Es muss nicht der gesamte Interpreter neu entwickelt werden, sondern nur das clientseitige Programm.

Notwendige Arbeiten

- Das DORI-Metamodell muss verstanden und ein entsprechendes Java Handling implementiert werden.
- Entsprechender Webserver muss aufgesetzt werden.
- Schnittstelle zum Einladen des Modells muss programmiert werden.
- Auswahl der Aktionen auf Grund des Modells muss implementiert werden.
- Eine Schnittstelle zur Programmlogik muss implementiert werden.
- Ein extra Client für jede Plattform muss entwickelt werden.
- Die Kommunikation zwischen den Programmen muss implementiert werden.

Vorteile

- Die Aufteilung der einzelnen Funktionen in seperate Programme führt zu einer sehr zukunftssicheren, modularen erweiterbaren Architektur.
- Neue Plattformen brauchen ’nur’ ein Clientprogramm.

Nachteile

- Eher hohe Komplexität durch die Aufteilung in verschiedene Programme und der Kommunikation dieser untereinander.

- Die Umsetzung des Interpreters als Server könnte zu Depolymentproblemen führen, wenn die Anwendung offline verwendet werden soll.
- Wie schon beim einfachen Interpreter ist zu erwarten, dass die Performance im Vergleich zum Generator schlechter ist. Weiterhin kommt noch der Kommunikationsaufwand zwischen dem Client und Server hinzu.

8.5.4. Entscheidung für erste Iteration

Felix, Thomas, Stephan

Nachstehend werden die Ergebnisse des vorherigen Abschnitts nochmal tabellarisch zusammen gefasst.

Tab. 8.1.: Gegenüberstellung Übersetzer

Lösung	**Komplexität**	**Neue Technologien**	**Transportierbarkeit Plattformen**	**Performance Endanwendung**
Generator	hoch	Programiersprache (JS/TS), Framework (Angular)	sehr gut (iOS, Android, Desktop)	eher hoch
Interpreter (einfach)	gering	Framework (JSF)	nicht gegeben	eher niedrig
Interpreter (Timo)	hoch	Framework (Webserver)	möglich durch neuen Client, aber offline Probleme	eher niedrig

Für die erste Iteration fällt die Entscheidung auf den einfachen Interpreter. Der Grund hierfür ist vor allem die geringe Komplexität und die eher kleine Einarbeitung in ein neues Framework und keine Einarbeitung in neue Programmiersprachen. Erste sichtbare Ergebnisse können somit schnell erreicht werden.

Da die meisten Mitglieder der Projektgruppe JAVA Programmiererfahrung haben, bleibt einstehender Sourcecode für alle nachvollziehbar.

Die Nachteile sind besonders für einen ersten technischen Durchstich zu vernachlässigen.

9. Architekturbeschreibung

Felix, Thomas, Nancy, Jan (ehemaliges Projektgruppenmitglied)

Aus den Entscheidungen geht heraus, dass es eine DORI-DSL gibt, die mittels eines Meta-Modells beschrieben wird. Auf dieses Modell baut der Editor auf, der mit Sirius gebaut wird. Die fertig modellierte Interaktion wird an einen Interpreter übergeben, der die Interaktion in eine Applikation übersetzt.

Im Folgenden wird die grundlegende Architektur des Gesamtsystems DORI in seine einzelnen Teilkomponenten zerlegt und beschrieben. Angelehnt an die Siemens Sichten werden im Folgenden die Entwurfskomponenten und deren Beziehung zueinander, in Form einer Konzeptsicht, beschrieben. Die Modul-Sicht schlüsselt diese Komponenten in ihre Module und Teilsysteme auf. Auf eine Beschreibung der Ausführungs- und Quelltext-Sicht wird an dieser Stelle verzichtet. Eine genauere Beschreibung der Komponenten, wie diese ausgeführt werden und wie diese im Detail erstellt wurden, kann aus der Benutzerdokumentation bzw. der Entwicklerdokumentation entnommen werden.

9.1. Konzeptsicht

Felix, Thomas, Nancy, Jan (ehemaliges Projektgruppenmitglied)

In der Konzeptsicht werden aus den vorhergehenden Faktoren, sowie aus den Anforderungen, die Hauptfunktionalitäten abgeleitet und beschrieben. Die Hauptfunktionalitäten sollen auf die Komponenten des Zielsystems abgebildet werden. Die Komponenten wurden durch die Entscheidungsfindungen festgelegt.

Das DORI-Gesamtsystem gliedert sich in zwei Hauptfunktionalitäten. Diese ergeben sich aus den Zielen der Projektgruppe, welche in der Abbildung 9.1 dargestellt sind.

Modellierung der Gesamtinteraktion Diese Hauptfunktionalität beinhaltet die Eingabe des Modells einer Gesamtinteraktion mittels einer definierten graphischen Sprache in den Editor. Die Basis der graphischen Sprache bildet die DORI-DSL. Diese ist eine wichtige Komponente des Zielsystems und wird im Kapitel beschrieben. Für den Editor wird Sirius als Tool verwendet.

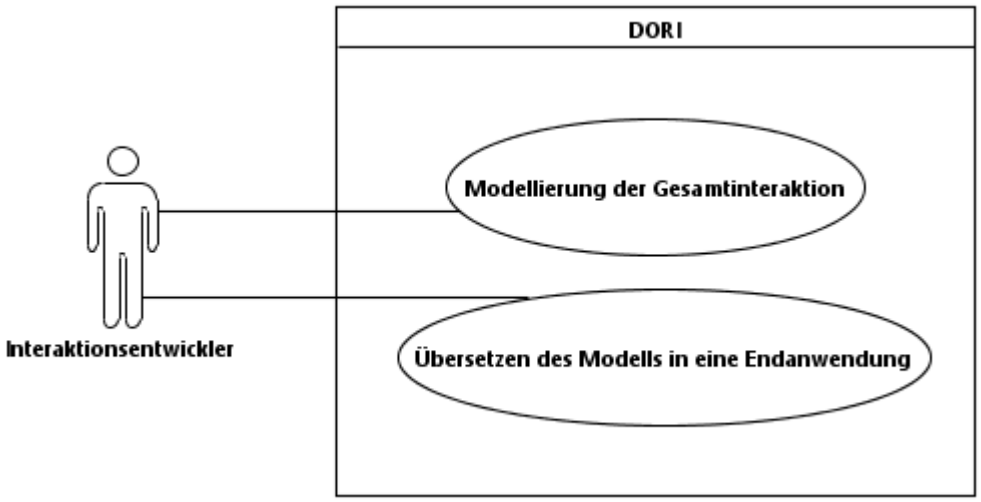


Abb. 9.1.: DORI-Hauptfunktionalitäten

Übersetzen des Modells in eine Endanwendung Diese Hauptfunktionalität beinhaltet die Überführung des Modells in eine Datei, die von einem Übersetzter verwendet wird für die Übersetzung in eine ausführbare Endanwendung. Bei diesem Übersetzter handelt es sich um einen Java-Interpreter. Die Plattform, in die die Anwendung übersetzt wird wurde für den ersten technischen Durchstich auf „Web“ festgelegt. Im späteren Verlauf des Projektes wurde auch die Plattform „Android“ umgesetzt.

Für diese Hauptfunktionalitäten (Modellierung der Gesamtinteraktion und Übersetzen des Modells in eine Endanwendung) sind die drei Komponenten Editor, Interpreter und DORI-DSL nötig. Die DORI-DSL stellt dabei die Basis für den Editor. Der Editor stellt eine Schnittstelle für den Interpreter. Sobald in dem Editor eine, der DORI-DSL konformen, Gesamtinteraktion modelliert wurde, wird dieses Modell in Form einer XML-Datei an den Interpreter übergeben.

Die Hauptfunktionalität Modellierung der Gesamtinteraktion erfolgt in dem Editor. Die Hauptfunktionalität Übersetzung in eine lauffähige Endanwendung wird von dem Interpreter übernommen. Der DORI-DSL ist keine der Hauptfunktionalitäten direkt zugeordnet. Doch ohne diese könnte der Editor die gewünschten Funktionalitäten nicht umsetzen. Die Abbildung 9.2 zeigt den Zusammenhang dieser Komponenten.

Neben dem Zusammenhang der Komponenten, sollen hier auch die einzelnen Funktionsweisen derer erläutert werden.

Mit Sirius ist es möglich einen Editor für seine eigene DSL zu erstellen. Als Grundlage dient die DORI-DSL. Diese bietet die Möglichkeiten, abstrakte GUI-Komponenten zu verwenden und zu definieren. Ebenso müssen diese Funktionalitäten für die Programmlogik möglich

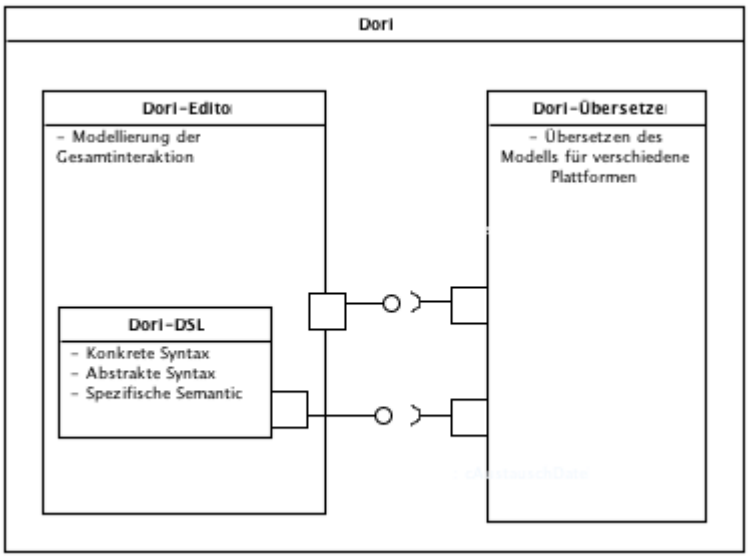


Abb. 9.2.: Komponentendiagramm System

sein. Für eine Interaktion ist es nötig, einen Wechsel zwischen GUI-Komponenten zu haben. Diese Wechsel werden von der DORI-DSL bereit gestellt, ebenso wie die Modellierung von Datenflüssen. Zu jedem abstrakten Inhalt muss eine konkrete Implementierung vorhanden sein. Die Verknüpfung von dem abstrakten und konkreten Inhalt wird von der DORI-DSL bereit gestellt und von dem Editor visualisiert. Der Editor selber soll außerdem die Elemente der DORI-DSL visualisieren können. Dabei soll es möglich sein, eine angefangene Interaktion zu bearbeiten und zu speichern. Um eine Wiederverwendung zu schaffen, sollen für abstrakte und konkrete Inhalte Kataloge durch den Editor zur Verfügung gestellt werden.

Um eine funktionierende Übersetzung zu garantieren muss dem Interpreter ein DSL konformes Modell übergeben werden. Eine Teilaufgabe des Editors ist somit die Konsistenzprüfung, sowie das Exportieren des Modells in ein geeignetes Format, welches der Interpreter einlesen kann.

Der Interpreter muss nicht nur die Übersetzung des Modells in eine lauffähige Endanwendung ermöglichen, sondern auch die Übersetzung auf verschiedene Plattformen ermöglichen. Dazu verwendet dieser ebenfalls die DORI-DSL.

Die drei Komponenten DORI-DSL, Editor und Interpreter stellen die Hauptkomponenten des DORI-Zielsystems dar. Diese lassen sich noch in verschiedene Teilkomponenten und -Systeme unterteilen.

9.2. Modulsicht

Felix, Thomas, Nancy, Jan (ehemaliges Projektgruppenmitglied)

Die Modulsicht verfeinert die Konzeptsicht. In dieser Sicht sollen die Entwurfskomponenten Editor, DORI-DSL und Interpreter in deren Teilsysteme und Teilkomponenten zerlegt und definiert werden.

Der Editor basiert auf Sirius. Sirius ist ein Tool, welches auf das Eclipse Modelling Framework (EMF) aufbaut. Dabei ist es mit Sirius möglich, sowohl die abstrakte als auch die konkrete Syntax der DORI-DSL einzubinden und daraus einen Editor zu generieren, der es zulässt verschiedene Instanzen von dem Metamodell der DORI-DSL zu modellieren. Das Metamodell ist dabei in Form eines Ecore-Modells, welches von Sirius verwendet wird. Eine genauere Beschreibung von dem Aufbau der verschiedenen Technologien und deren Verbindungen ist in der Entwicklerdokumentation vorzufinden.

Für die Kataloge stellt der Editor verschiedene Sichten in Form von Tabellen, Baumstrukturen und Diagrammflächen bereit. Die Konsistenzprüfung wird ebenfalls durch Sirius bereitgestellt.

Ein wichtiges Element der DORI-DSL sind die Guards. Diese konnten nicht direkt in Sirius realisiert werden. Dazu wurde als externes Tool XText eingesetzt. Dabei stellt XText sowohl die Überprüfung der Guards als auch deren Syntax bereit. XText wird dabei über einen Rest-Service eingebunden.

Der Editor teilt sich somit in die Teilkomponenten Sirius, EMF, XText und der DORI-DSL selber auf. Die DORI-DSL als eigenständige Komponenten teilt sich in die einzelnen Klassen auf.

Auch im Interpreter wird XText über einen Rest-Service eingebunden. Dieser dient dazu die Guards in dem Modell nochmal vor dem Übersetzten zu überprüfen und diese logisch aufzulösen. Der Interpreter selber bekommt das Modell in Form einer XML-Datei. Dieser teilt sich in verschiedene Teilkomponenten auf. Der Core-Interpreter stellt alle Funktionalitäten zum Übersetzten des Modells. Für jede Plattform wird jeweils ein neuer Interpreter geschrieben, der speziell für die Plattform noch zusätzliche Funktionalitäten bereit stellt, um die Applikation zu erstellen und zu verwenden. Die einzelnen Interpreter teilen sich noch weiter in Teilkomponenten auf. Die Beschreibung hierzu kann in der Entwicklerdokumentation nachgelesen werden.

Die Abbildung 9.3 zeigt nicht nur die Aufteilung der Komponenten in deren Teilkomponenten, sondern auch den bereits in der Konzeptsicht angesprochenen Zusammenhang dieser.

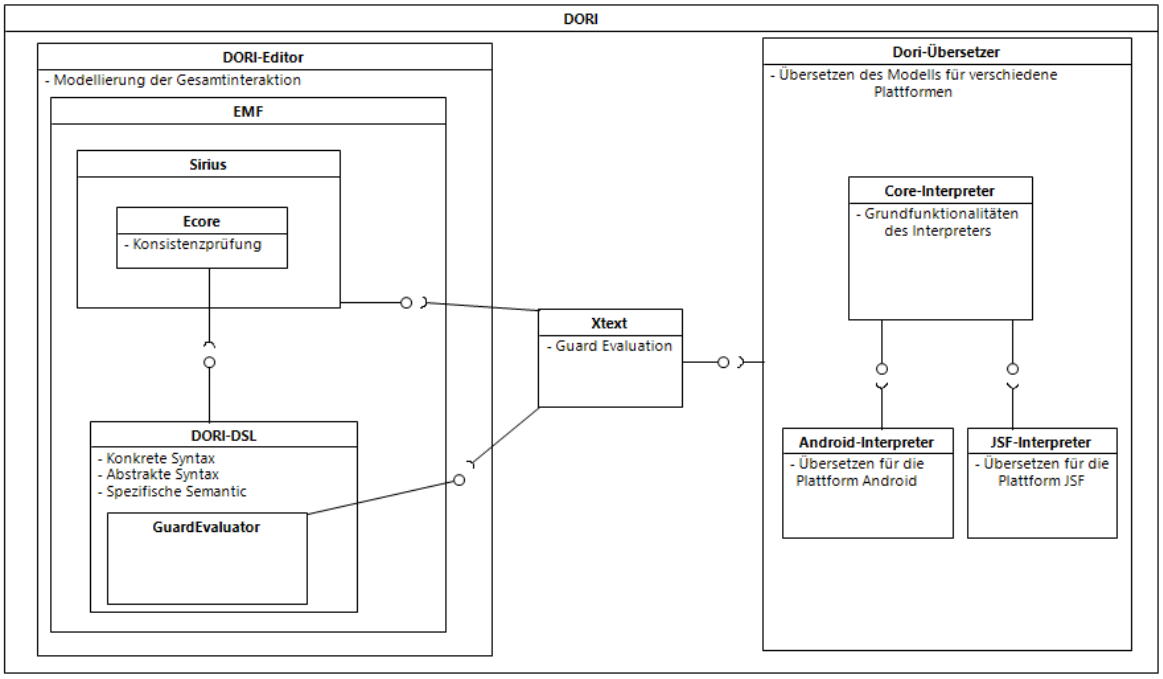


Abb. 9.3.: Komponentendiagramm System

Teil III.

Projekttagebuch

10. Sprint 0

Hannah

Planung

Nachdem am 6. April 2017 das Einführungstreffen der Projektgruppe stattgefunden hatte, welches vorrangig informativer Natur war, nahm diese zum 12. April 2017 ihre Arbeit auf. Es galt zunächst, organisatorische Themen zu klären, Richtlinien zum gemeinsamen Arbeiten aufzustellen und den einzelnen Mitgliedern der Projektgruppe Rollen zuzuweisen. Zudem wurde das Projektziel motiviert und definiert sowie erste Anforderungen erhoben, um selbiges zu erreichen. Sprint 0 wurde daher – im Gegensatz zu den folgenden Sprints – nicht als formeller Sprint inklusive Sprintzielen, Review und Retrospektive durchgeführt, sondern kann als Findungsphase angesehen werden. Beteiligt waren an diesem Sprint zu jeder Zeit sämtliche Mitglieder der Projektgruppe bis auf die in Tabelle 10.1 aufgeführten Mitglieder, welche eine Zeit lang Urlaub genommen hatten. Der Sprint endete am 31. Juli 2017, da anschließend der erste offizielle Sprint eingeleitet wurde.

Tab. 10.1.: Urlaubszeiten in Sprint 0

PG-Mitglied	Urlaubsbeginn	Urlaubsende
Hauke Fischer	17. Juli 2017	31. Juli 2017
Lisa Ripke	29. Juli 2017	20. August 2017

Durchführung

Die Projektgruppe begann die Projektarbeit damit, den organisatorischen Rahmen für die Zusammenarbeit der Mitarbeiter abzustecken. So wurden etwa feste wöchentliche Besprechungstermine etabliert, welche durch eine Sitzungsleitung geleitet und von einem Protokol-

lanten dokumentiert wurden. Zudem wurden Kommunikationskanäle innerhalb der Gruppenmitglieder sowie zwischen der Gruppe und den Betreuern eingerichtet. Anschließend wurden verschiedene Rollen definiert und mit Mitgliedern besetzt. Zum Aufgabenmanagement wurde JIRA, zum Wissensaustausch Confluence eingerichtet. Auch wurde die Basis für eine Projektdokumentation in Form von strukturierten LaTeX-Dokumenten geschaffen. Zusätzlich wurde ein Git sowie ein FTP-Server eingerichtet. Schließlich einigte sich die Projektgruppe für die Projektarbeit auf ein Scrum-ähnliches Vorgehensmodell. Sämtliche obig beschriebene Regeln und Prozesse wurden zudem in einem Projekthandbuch niedergeschrieben (siehe Kapitel 6).

Die Projektgruppe teilte sich während der ersten Hälfte des nullten Sprints in drei Arbeitsgruppen auf. Während die Arbeitsgruppen „Projekthandbuch“ und „Administration“ mit der Einrichtung und Niederschrift des oben beschriebenen organisatorischen Rahmens betraut waren, beschäftigte sich die Arbeitsgruppe „Visionsdokument“ damit, die Ziele der Projektgruppe innerhalb eines solchen Dokuments zu spezifizieren und zu verschriftlichen. Als Basis dienten ihnen hierfür sowohl ausgedehnte Gespräche zwischen der Projektgruppe und den Betreuern als auch Interviews, welche mit den einzelnen Betreuern durchgeführt wurden. Anschließend wurden in der zweiten Hälfte von der ganzen Projektgruppe Anforderungen zum Erreichen der Ziele formuliert und stetig angepasst. Im Rahmen der Anforderungserhebung wurde Prototyping auf dem Gebiet der Softwareentwicklung für Web und Android sowie zur Verwendung von IFML betrieben (siehe Kapitel 5).

Gegen Ende des Sprints wurde sich verstärkt der längerfristigen Projektplanung gewidmet. Statt etwa den Projektplan weiterhin in MS-Project darzustellen, wurde dieser ins Confluence verlegt und feiner gegliedert, um ihn präsent und nachvollziehbar zu halten. Zudem wurde der Umfang des Systementwurfs umrissen, da dieser Hauptinhalt von Sprint 1 werden sollte. Nach vorhergehender Recherche auf dem Gebiet von DSLs wurde sich auf ein Vorgehen bezüglich der Sprachentwicklung geeinigt. Außerdem wurde Web als Zielplattform für den ersten technischen Durchstich (Sprint 2) festgelegt. Zum Ausbau des Wissens der Projektgruppenmitglieder wurden Seminarvorträge vergeben und gehalten, die Vortragenden und ihre Themen können Tabelle 10.2 entnommen werden.

Tab. 10.2.: Seminarvorträge in Sprint 0

PG-Mitglied	Vortragsthema	Gehalten am
Lisa Ripke	JIRA und Confluence	03. Mai 2017
Felix Kempa	Generator versus Interpreter	24. Mai 2017
Christopher Bischopink	GUI-Builder, GUI-Prototyping	24. Mai 2017
Hauke Fischer	DSLs, EMF und Sirius	31. Mai 2017
Stephan Bogs	MPS – Meta Programming System	31. Mai 2017
Nancy Kramer	MetaEdit+	08. Juni 2017
Hannah Meyer	IFML	14. Juni 2017

11. Sprint 1

Hannah

Planung

Der erste offizielle Sprint der Projektgruppe startete am 01. August 2017 und sollte dazu dienen, den System- und Sprachentwurf als Basis für den im nächsten Sprint folgenden ersten technischen Durchstich (siehe Kapitel 12) fertigzustellen. Beteiligt waren an diesem Sprint zu jeder Zeit sämtliche Mitglieder der Projektgruppe bis auf die in Tabelle 11.1 aufgeführten Mitglieder, welche eine Zeit lang Urlaub genommen hatten. Sprint 1 wurde für einen Monat angesetzt und am 31. August 2017 beendet.

Tab. 11.1.: Urlaubszeiten in Sprint 1

PG-Mitglied	Urlaubsbeginn	Urlaubsende
Lisa Ripke	29. Juli 2017	20. August 2017
Jan Philipp Hörding	07. August 2017	22. August 2017
Stephan Bogs	24. August 2017	04. September 2017

Ziele

- Systementwurf
  - Festlegung Editorplattform
  - Festlegung Editortool
  - Festlegung Übersetzer (Generator, Interpreter)
  - Komponentenbeschreibung

- Metamodell der Sprache
- Systementwurfsdokument
- Definition Anwendungsbeispiel

Durchführung

Innerhalb dieses Sprints wurden die Anforderungen, welche im vorherigen Sprint aufgestellt worden waren, weiter überarbeitet. Zudem wurde ein erstes Metamodell für die DORI-DSL entworfen und laufend weiterentwickelt. Im Rahmen des Systementwurfs wurden verschiedene Editortools gegeneinander und gegen die Anforderungen evaluiert sowie die Zielplattform des Editors definiert. Die Entscheidung, ob innerhalb des ersten technischen Durchstiches ein Interpreter oder ein Generator entwickelt werden soll, wurde zugunsten des Interpreters getroffen. Es wurden Strategien zur übersichtlichen Versionierung der entstandenen Projektdokumentation festgelegt. Schließlich ist ein Anwendungsbeispiel konzipiert worden, welches im ersten technischen Durchstich umgesetzt werden soll.

Review

Es existiert eine vorläufige Version des Metamodells, mit der weitergearbeitet werden kann. Hier sind jedoch einige Constraints noch nicht in OCL (Object Constraint Language) umgesetzt. Die Ziele des Systementwurfs wurden sämtlich erreicht. Die erste Version des Systementwurfsdokuments konnte jedoch nicht gänzlich fertiggestellt werden, siehe Retrospektive.

Retrospektive

Als Probleme während des ersten Sprints und deren Lösungsansätze wurden die folgenden Punkte vermerkt:

- Die geführte Recherche bezüglich Inhalt und Gliederung eines Systementwurfsdokuments war nahezu überflüssig, da sie zu abstrakt und zu wenig auf die Projektgruppe

abgestimmt vorgenommen wurde. Dadurch entstanden zum Ende des Sprints Zeitprobleme bei der Gliederung und Erstellung der ersten Version des Systementwurfsdokuments. Recherchen werden künftig in direktem Bezug auf die Projektgruppe durchgeführt. Zudem kann sich bei Dokumentgliederungen an alten Projektgruppen orientiert werden.

- Es gab Schwierigkeiten bei der Erstellung des Metamodells, da auf sehr abstrakter Basis gearbeitet wurde. Das fehlende Expertenwissen verursachte auch Probleme bei der Formulierung der Anforderungen. In Zukunft soll früheres Prototyping die Entwicklung erleichtern.
- JIRA erwies sich für kleine Arbeitsgruppen als zu arbeitsintensiv. Zudem wurde JIRA als überflüssig eingestuft, wenn – wie etwa in der vorlesungsfreien Zeit – die Möglichkeit gegeben ist, dass sich alle Gruppenmitglieder verbal abstimmen. Eine leichtgewichtigere Nutzung von JIRA innerhalb von Kleingruppen und während der Semesterferien stellte die Lösung für diese Punkte dar. Innerhalb des Semesters soll JIRA jedoch wieder verstärkt genutzt werden.
- Der RSA (Rational Software Architect) erwies sich als Blocker, er wurde überwiegend als wenig performant sowie schwierig in der Bedienung beurteilt. Trotzdem entschloss sich die Projektgruppe, mit diesem beim Entwurf von UML-Diagrammen weiterzuarbeiten, um die optische Einheitlichkeit in der Abgabe zu wahren.
- Die Mitglieder der Projektgruppe haben sich zum Teil nicht an Regeln und Deadlines gehalten. Um Regeln und Richtlinien präsenter zu halten, wurden diese an einer zentraleren Stelle im Confluence vermerkt. Für den direkten Überblick wurde der Projektplan innerhalb des PG-Raums auf dem Whiteboard nieder geschrieben. Zudem sind die Mitglieder angehalten, frühestmöglich zu kommunizieren, wenn eine Deadline nicht eingehalten werden kann.

Als positiv empfunden wurden folgende Aspekte:

- Zielorientierteres, synchronisierteres Arbeiten der Projektgruppe.
- Regelmäßiges verpflichtendes Donnerstagstreffen zum gemeinsamen Arbeiten inklusive zusätzlichem Standup. Aus diesem Grund soll ein gemeinsames Arbeitstreffen auch während des Semesters etabliert werden.
- Projektplan am Whiteboard für direkten Überblick über Ziele, Daten und Verantwortlichkeiten.
- Kleine, fokussiert arbeitende Arbeitsgruppen sowie die Kommunikation innerhalb dieser.

12. Sprint 2

Hannah

Planung

Der zweite offizielle Sprint der Projektgruppe startete am 01. September 2017 und trug den Beinamen „Erster technischer Durchstich“, da in diesem Sprint ein Teil des Gesamtsystems durch alle Ebenen hindurch – insbesondere sowohl im Editor als auch im Interpreter – implementiert werden sollte. Beteiligt waren an diesem Sprint zu jeder Zeit sämtliche Mitglieder der Projektgruppe bis auf die in Tabelle 12.1 aufgeführten Mitglieder, welche eine Zeit lang Urlaub genommen hatten. Sprint 2 wurde für einen Monat angesetzt und am 30. September 2017 beendet.

Tab. 12.1.: Urlaubszeiten in Sprint 2

PG-Mitglied	Urlaubsbeginn	Urlaubsende
Thomas Sprock	08. September 2017	15. September 2017
Christopher Bischopink	10. September 2017	24. September 2017
Felix Kempa	18. September 2017	01. Oktober 2017
Nancy Kramer	18. September 2017	01. Oktober 2017

Ziele

Die folgenden Aspekte der DORI-DSL sollten nach dem ersten technischen Durchstich im Editor modellierbar sowie vom Interpreter verarbeitbar sein:

- Geschachtelte Widgets
- Transitionen

- Events
- Daten / Variablen
- Funktionsaufrufe (optional)
- Datenübergabe zwischen Widgets (optional)

Insbesondere wurden Guards, Kataloge, parallele Widgets und Konsistenzprüfungen aus dem ersten technischen Durchstich ausgeschlossen. Im Verlauf des Sprints wurde zudem die Abbildung der konkreten Syntax im Editor aus den Zielen gestrichen.

Durchführung

Während des zweiten Sprints teilten sich die Mitglieder der Projektgruppe etwa zu gleichen Teilen auf die Arbeitsgruppen „Editor“ und „Interpreter“ auf. Sowohl vom Metamodell als auch vom Anwendungsbeispiel wurden gekürzte Versionen erstellt, welche die Ziele des Sprints widerspiegeln. Das vereinfachte Metamodell wurde als Basis für den Editor nach Ecore übertragen. Als Schnittstelle zwischen Interpreter und Editor wurde sich nach einem Seminarvortrag über Austauschformate (siehe Tabelle 12.2) auf JSON geeinigt. Diese Entscheidung wurde jedoch wieder verworfen, als sich herausstellte, dass beide Gruppen (nahezu) direkt mit dem Ecore Metamodell und dessen .xmi Datei arbeiten können. Somit wurde dieses als Single Point of Truth etabliert. Der Editor wurde mittels Eclipse Sirius, der Interpreter als JSF-Projekt realisiert. Zudem wurden innerhalb des Sprints tiefergehende semantische Festlegungen insbesondere bei der Abarbeitung von Events und Transitions (inklusive deren Actions und ParameterBindings) getätigt. Schließlich wurden Methoden zur Qualitätssicherung etabliert.

Tab. 12.2.: Seminarvorträge in Sprint 2

PG-Mitglied	Vortragsthema	Gehalten am
Thomas Sprock	Austauschformate, Techniken	06. September 2017
Jan Philipp Hörding	Qualitätssicherung	13. September 2017

Review

Die Ziele des Sprints sind bis auf einige nachfolgend umrissenen Einschränkungen vollständig erreicht worden. Das Team konnte dadurch die Betreuer beeindrucken. Als Abstriche verbleiben lediglich, dass im Editor nur globale Daten definiert werden können und im Interpreter das Datenhandling nicht komplett umgesetzt werden konnte. Es stellte sich heraus, dass beim Scope der Variablen generell noch Diskussionsbedarf besteht. Dies gilt auch für das Transition-Handling, insbesondere für die Semantik von Transitionen zwischen Widgets unterschiedlicher Ebenen. Von Seiten der Betreuung wurde angemerkt, dass zu jedem folgenden Release installationsfähige Plugins erstellt werden sollen, um anderen Personen das Produkt zugänglich zu machen. Zudem wird empfohlen, die Usability des Editors und der konkreten Syntax zu überdenken, welche in diesem Rahmen kein Fokus des Sprints war.

Retrospektive

Als Probleme während des zweiten Sprints und deren Lösungsansätze wurden die folgenden Punkte vermerkt:

- Sirius stach negativ durch schlechte Dokumentation und mangelnde Fehlerauswertung hervor. Dies wurde nicht frühzeitig erkannt, da bei der Toolevaluation nur ein Tutorial nachgebaut wurde. In Zukunft müssen Toolevaluationen über das Tutorial hinaus durchgeführt werden.
- Die späte Einigung bezüglich des zugrundeliegenden Metamodells sowie des Austauschformats der Modelle hat zu leichter Verzögerung geführt. In Zukunft soll derartigen Verzögerungen durch umfangreichere Recherche im Vorfeld entgegengewirkt werden.
- JIRA wurde erneut relativ wenig genutzt. Die Projektgruppe beschließt, verstärkt mit Epics zu arbeiten und einen detaillierteren Backlog vor Sprintbeginn anzulegen.
- Änderungen am Anwendungsbeispiel wurden nicht zentral abgesprochen. Die aktuelle Version soll künftig zentral im Confluence gepflegt werden.
- Es muss an der Synchronisation der Arbeitsgruppen sowie an der Qualitätssicherung gearbeitet werden. Dazu soll in Sprint 3 eine neue Rolle geschaffen werden. Zudem nimmt sich die Projektgruppe vor, eine bessere Impact-Analyse für Tasks zu betreiben.

Als positiv empfunden wurden folgende Aspekte:

- Umfang der Ergebnisse des ersten technischen Durchstichs.

- Gutes Teamklima, motivierte Mitglieder (z.T. trotz Krankheit).
- Mob-Programming in der Interpretergruppe.

Schließlich wurde diskutiert, in welchem Umfang die Projektgruppe Scrum einsetzen möchte. Als Ergebnis lässt sich festhalten, dass Scrum mit Kleingruppen inklusive Gruppenleitern kombiniert wird.

13. Sprint 3

Jan (ehemaliges Gruppenmitglied), Hannah

Planung

Der dritte offizielle Sprint der Projektgruppe startete am 30. September 2017 und trug den Beinamen „Reviewsprint“, da in diesem Sprint die Arbeitsergebnisse des vorangegangenen Sprints aufgearbeitet und ausgewertet wurden. Dies betraf vor allem die Dokumentation der implementierten Features sowie Erläuterungen im Code. Zudem sollten so einheitliche Sprintlängen von einem Monat für die Sprints vor den Weihnachtsfeiertagen ermöglicht werden. Beteiligt waren an diesem Sprint zu jeder Zeit sämtliche Mitglieder der Projektgruppe bis auf die in Tabelle 13.1 aufgeführten Mitglieder, welche eine Zeit lang Urlaub genommen hatten. Sprint 3 wurde für zwei Wochen angesetzt und am 16.10.2017 beendet. Aufgrund der Kürze des Sprints wurde hier kein ausführliches Review mit Retrospektive durchgeführt, jedoch fand ein informeller Rückblick statt.

Tab. 13.1.: Urlaubszeiten in Sprint 3

PG-Mitglied	Urlaubsbeginn	Urlaubsende
Stephan Bogs	6. Oktober 2017	15. Oktober 2017
Hannah Meyer	7. Oktober 2017	15. Oktober 2017

Ziele

Die folgenden Ziele wurden in Sprint 3 festgelegt.

- Review, Retrospektive von Sprint 2
- Präsentation und Demonstration der Deliverables von Sprint 2

Das Hauptaugenmerk lag bei diesem Sprint auf der Dokumentation der Projektgruppe.

Durchführung

Die in Sprint 2 erfolgte Aufteilung der Projektgruppe etwa zu gleichen Teilen in die Arbeitsgruppen „Editor“ und „Interpreter“ wurde in Sprint 3 weitergeführt. Es wurde daran gearbeitet, die Arbeitsumgebung des Interpreters auf allen Rechnern der Interpretergruppe lauffähig zu gestalten, da hier insbesondere bei Windows-Rechnern bisher Probleme auftraten. Zudem wurde angefangen, eine Testerrolle (QA) zu definieren. Diese soll in Zukunft die Qualitätssicherung gewährleisten, indem Test erstellt werden, die vom Interpreter durchlaufen werden. Im Editor wurde mit der Erstellung eines Entwicklerhandbuchs begonnen. Fehler im Modell des Anwendungsbeispiels wurden behoben und das Metamodell wurde angepasst.

Rückblick

Es wird nun ein eingebetteter Jetty-Server benutzt, der über Maven gestartet wird. Die bisherige Lösung mit einem Tomcat führte zu Problemen bei der Verwendung unter Windows. Die Dokumentationen wurden weitergeführt und in diesem Rahmen die Urlauber der vorherigen Phase eingearbeitet.

14. Sprint 4

Hannah

Planung

Der vierte Sprint startete am 16. Oktober 2017 und war der Ergänzung der Editor- und Interpreterfunktionalität um Guards und DecisionStates gewidmet. Beteiligt waren an diesem Sprint zu jeder Zeit sämtliche Mitglieder der Projektgruppe bis auf die in Tabelle 14.1 aufgeführten Mitglieder, welche eine Zeit lang Urlaub genommen hatten. Sprint 4 wurde für etwa einen Monat angesetzt und am 12. November 2017 beendet.

Tab. 14.1.: Urlaubszeiten in Sprint 4

PG-Mitglied	Urlaubsbeginn	Urlaubsende
Thomas Sprock	23. Oktober 2017	30. Oktober 2017
Christopher Bischopink	07. November 2017	16. November 2017

Ziele

Die für Sprint 4 gesetzten Ziele lauteten wie folgt:

- Ergänzung des Metamodells um Guards und Decision-States sowie Einbindung von Constraints in selbiges.
- Abbildung des erweiterten Metamodells sowohl im Editor als auch im Interpreter.
- Deployment des Editors sowie Testen des Editorcodes.
- Implementierung des Layerkonzepts und des Compiler/Classloader Features im Interpreter.

- Gewährleistung der Wiederverwendbarkeit der xhtml-Files.
- Arbiträre Parameter für Rest-Calls.
- Deployment des Interpreters.
- Erstellen von Testmodellen im Editor und einfachen Zugriff auf diese im Interpreter, außerdem Definition eines Änderungsmanagements dieser Modelle.
- Durchführung und Dokumentation von explorativen Testruns.
- Integrationstests

Durchführung

Während des Sprints setze die Projektgruppe erneut auf die etablierte Aufteilung in die Untergruppen „Editor“, „Interpreter“ sowie die neu geschaffene „QA“. Trotz Vorlesungsbeginns konnte als Kernarbeitszeit Dienstag von 10 bis 16 Uhr etabliert werden. Im Verlauf des Sprints traten einige Probleme mit XText auf, womit Guards realisiert werden sollten. So gelang es lange nicht, die Guards hiermit in der gewünschten Granularität zu parsen. Zudem bekam der Interpreter XText lange nicht integriert und die Dokumentation für XText im Editor erwies sich als dürftig. Die genauere Untersuchung dieses Problems ist im Sprintreview zu finden. In diesem Sprint wurde ein genauer Ablauf für Sprintenden und -übergänge etabliert und hier insbesondere die sogenannte „Review-Woche“ eingeführt, die nach jedem zukünftigen Sprint stattfinden sollte und in der etwa die Dokumentation und Javadoc aufgearbeitet werden wird. Schließlich wurde eine Urlaubsregelung – insbesondere über die Weihnachtsfeiertage – definiert.

Review

Nach Abschluss von Sprint 4 sind fast alle Elemente der DSL – bis auf Guards und einige Pseudostates – im Editor abbildbar. Als neue Features lassen sich hier die Einführung einer Datensicht sowie von Datentypen, die Ermöglichung einer Druckansicht, nutzerfreundliches Löschen von Elementen und die Bereitstellung einer Deployed Zip, welche bereits Sirius enthält, nennen. Die Usability und konkrete Syntax sind noch verbesserungswürdig und werden als Ziele für einen eigenen Sprint angesetzt.

Der Interpreter kann noch keine Guards verarbeiten, was im Wesentlichen auf Unklarheiten bezüglich XText (siehe auch „Retrospektive“) zurückzuführen ist. Die Bereitstellung der Guards erhält oberste Priorität im kommenden Sprint.

Von Seiten der Qualitätssicherung sind mithilfe des Editors Testmodelle erstellt worden, welche zum Testen des Interpreters eingesetzt werden. In Zukunft sollen auch die Funktionalitäten des Editors kritischer getestet werden.

Schließlich ist nun eine Uploadpage unter `pg-dori-web.informatik.uni-oldenburg.de:9090/dori` verfügbar, auf welcher deployte Teilprodukte von DORI herunter- sowie eigene Modelle hochgeladen werden können. Auch können von hier aus Testmodelle gestartet werden. Allerdings fehlt an dieser Stelle noch eine Benutzungsanleitung.

Retrospektive

Als Probleme während des vierten Sprints und deren Lösungsansätze wurden die folgenden Punkte vermerkt:

- Die Technologie-Entscheidung für XText zur Umsetzung von den Guards war nicht fundiert genug, XText stellt sich in Verbindung mit EMF als schwergängig heraus. In Zukunft müssen Toolevaluationen gerichteter und insbesondere mit Fokus auf den genauen Anwendungspunkt und die Weiterverwendbarkeit durchgeführt werden.
- Im Zusammenhang mit XText trat als weiteres Problem auf, dass nur eine einzelne Person mit dieser (für die Guards kritischen) Recherche- und Implementierungsaufgabe betraut war, welche zudem am Ende des Sprints ohne ausreichende Aufgabenübertragung auf andere Gruppenmitglieder den Urlaub antrat. In zukünftigen Sprints sollen kritische Aufgaben vorher identifiziert werden und Abhängigkeiten im Vorfeld aufgedeckt werden. Zudem hat jedes Gruppenmitglied für die Übergabe seiner Aufgaben und Kompetenzen vor Urlaubsantritt Sorge zu tragen. Gleichzeitig soll weniger auf Einzelkompetenz gebaut werden, indem eine kritische Aufgabe mit mehr Personen besetzt wird. Der Urlaub für die restliche Zeit der Projektgruppe sowie eventuell nötige Urlaubssperren sollen zeitnah abgeklärt werden.
- Das Feedback der QA an die Editor-Gruppe war zu dünn, da der Fokus zunächst auf die Erstellung von im Interpreter ausführbaren Testmodellen gelegt wurde. Dies soll ausgeglichen werden, indem direkt zu Beginn des nächsten Sprints explorative Testruns im Editor definiert und durchgeführt werden. Zudem gingen die Erwartungen an die QA innerhalb der Projektgruppe auseinander. In Zukunft sollen derartiges mit verbesserter Kommunikation über die Priorisierung von Tests vermieden werden.
- Die Umgangsformen in der Projektgruppe laufen beizeiten ins Unhöfliche. Es wird vereinbart, dass persönliche Probleme initial nicht in der ganzen Gruppe sondern persönlich bzw. über den „Hassbeauftragten“ als Mittelsmann geklärt werden. Insbesondere

soll hier das persönliche Gespräch der Kommunikation über E-Mail vorgezogen werden.

- Die Möglichkeiten von JIRA zur Planung und Untergliederung von Aufgaben wurden nicht voll ausgeschöpft. Dies soll ab sofort verbessert werden, um den aktuellen Arbeitsstand ersichtlicher zu machen. Zudem werden die Gruppenmitglieder angehalten, die Tasks nicht ohne einen Bearbeiter in „To Review“ zu verschieben. Außerdem soll Zeit für die Überprüfung der Tasks eingeplant werden.
- Es wurde der Wunsch nach mehr Anwesenheit der Mitglieder geäußert, auch damit der Austausch zwischen den Arbeitsgruppen verbessert werden kann. In Zukunft sollen daher Arbeitszeiten vermehrt in den Arbeitsgruppen abgesprochen werden. Zudem gab es in diesem Rahmen Bedenken über die gleichmäßige Arbeitbelastung innerhalb der Gruppe. Falls sich dies in Zukunft wiederholt, soll eine Klärung über die Projektleitung stattfinden. Außerdem wird darauf hingewiesen, dass Arbeit in Abwesenheit durchgeführt werden kann.
- Es muss besser darüber kommuniziert werden, wann welche Bestandteile der Sprintplanung umgesetzt werden. Priorisierungen müssen – in JIRA oder im Gespräch – abgeklärt werden.
- Schließlich wurde angemerkt, dass die Projektgruppendokumentation während der Sprints oft schleifen gelassen wurde. Diesem Problem soll mit der Einführung einer Review-Woche nach jedem Sprintende, welche unter anderem zur Aufarbeitung der Dokumentation dienen soll, entgegengewirkt werden.

Als positiv empfunden wurden folgende Aspekte:

- Umfassende Dokumentationstätigkeit der QA
- Fortschritt bezüglich Unit-Tests im Interpreter
- Gemeinsamer Arbeitstag „trotz“ Semesters
- Guter Termin für das Arbeitstreffen *vor* der Sitzung
- Einführung einer Review-Woche nach jedem Sprint
- Gutes Gruppenklima
- tolle Zusammenarbeit in der Arbeitsgruppe „Editor”

15. Sprint 5

Jan (ehemaliges Gruppenmitglied)

Planung

Der fünfte Sprint startete am 20. November 2017 und war der Ergänzung der Editor- und Interpreterfunktionalität um Guards und Decision-States gewidmet, da in diesem Bereich im letzten Sprint Funktionalitäten unvollständig geblieben waren. Beteiligt waren an diesem Sprint zu jeder Zeit sämtliche Mitglieder der Projektgruppe. Sprint 5 wurde für etwa einen Monat angesetzt und am 17. Dezember 2017 beendet. Die Woche zwischen Weihnachten und Silvester wurde als Betriebsferien definiert.

Ziele

Die für Sprint 5 gesetzten Ziele lauteten wie folgt:

- Metamodell
  - Guards
  - Constraints
  - fehlende States
- Editor
  - Guards
  - Decision States
  - Katalogsichten
  - Usability, IDs
- Interpreter

  - Guards
  - Decision States
  - Constraints prüfen
  - Testen
- QA
  - Automatisierte Testruns
  - Explorative Testruns Editor
  - Konzept für Testmodelländerungen
  - Konkretes Anwendungsbeispiel
- Konzept und Prototypen für 2. Plattform

Durchführung

Auch in diesem Sprint wurde auf die etablierte Aufteilung der Projektgruppe in die Untergruppen „Editor“, „Interpreter“ sowie die neu geschaffene „QA“ gesetzt.

Da im letzten Sprint Problemen mit XText und Guards auftraten, wurde hierauf ein Hauptaugenmerk gesetzt. Diese Probleme konnten letztlich in diesem Sprint gelöst werden. Sowohl das Parsen der Guards, als auch die Einbindung von XText gelang schließlich, sodass nun auch die Werte bei Guards ausgelesen werden können. Für die Umsetzung der Guards wurde ein gemeinsames Projekt erstellt, das vom Interpreter als auch vom Editor benutzt werden kann. Das durch XText generierte Ecore-Klassenmodell wird vorerst nicht in das Metamodell eingebunden. Dies geschieht aufgrund einer Komplexitätsreduzierung für die Editor- und Interpretereinbindung aufgrund dessen, dass Grammatikanpassungen leichter durchgeführt werden können.

Es wurde eine Validierung des Interaktionsmodells auf der Startseite des Interpreters hinzugefügt. Diese validiert das hochgeladene Interaktionsmodell mit EMF-Boardmitteln. Zusätzlich zur Log-Datei gibt es im Interpreter nun noch einen Debug-Modus, welcher alternativ zum normalen Start des Modells verwendet werden kann. Hier werden etwa Informationen angezeigt, welche Transition genommen wurde oder welche Guards ausgewertet wurden.

Beim Editor gibt es Probleme mit der Einbindung externer Funktionalität in Form von JARs. Hierzu wurde eine Frage in einem öffentlichen Forum erstellt, jedoch gab es hierauf keine Antwort. Als Alternative wurde eine Einbindung als REST-Service überlegt. Im Editor wurden

Kataloge als komplett neue Sichten eingeführt, die abstrakte auf konkrete Widgets, Funktionen usw. zuordnen. Die Kataloge waren ein erste Ansatz die Usability zu verbessern und konnten im Gegensatz zur ersten Annahme ohne Änderungen am Metamodell umgesetzt werden. Als Vorbereitung für den folgenden Sprint wurden Interviews zur Optik des Editors und der derzeitigen konkreten Syntax durchgeführt.

Die neu implementierten Features wurden von der QA überprüft. Durch Testmodelle konnten Fehler in Editor und Interpreter gefunden und die entsprechenden Arbeitsgruppen informiert werden. Auch wurden automatisierte GUI-Tests mit Selenium eingeführt.

Des Weiteren wurde mit der Abteilung Softwaretechnik ein Anwendungsbeispiel für diese PG Besprochen.

Zum Abschluss des Sprints wurde eine einwöchige Reviewwoche durchgeführt. In dieser wurde die Dokumentation gepflegt und die implementierten Features dokumentiert und Javadoc gepflegt.

Review

Die gesetzten Ziele des Sprints wurden erreicht. Lediglich der Androidübersetzer, der von Anfang an als Sollbruchstelle geplant war, wurde auf den nächsten Sprint verschoben.

Im Editor wurden die folgenden Aspekte hinzugefügt: Decision States, Guards, Kataloge, Mapping und Literale. Außerdem wurde ein Dropdown Menü für die jeweilige Plattform ergänzt. Im Interpreter wurden folgende Punkte ergänzt: Guards, Decision States, Literale und Mapping. Wesentlichen Punkte der QA: Die Testruns wurden mit Selenium automatisiert. Es wurden explorative Testruns im Editor durchgeführt. Ein konkretes Anwendungsbeispiel wurde umgesetzt und es wurden neue Testcases erstellt.

Der Guardevaluator in Sirius ist durch einen Restservice umgesetzt worden. Auf dem Server läuft nun die aktuelle Version. Der Core-interpreter wurde aus dem Dori-Projekt in ein eigenes Projekt extrahiert, mit dem Ziel eine Grundlage für die zweite Plattform Android zu bekommen.

Retrospektive

Als Probleme während des fünften Sprints und deren Lösungsansätze wurden die folgenden Punkte vermerkt:

- Der Prozess in Bezug auf die zu etablierende Ordnerstruktur der Testcases im Interpreter wurde nicht eingehalten und verursachte unnötige Arbeit.
- Features im Editor und in der Guardevaluierung waren nicht bis zum besprochenen Dienstag fertig. Hier wird auf eine verbesserte Planung geachtet.
- Es werden viele verschiedene Betriebssysteme und Toolversionen eingesetzt, die Probleme verursachen. So läuft der Interpreter lokal nicht unter Windows. Auch wurden zu viele unbekannte und mächtige Frameworks eingesetzt, die schlecht eingeschätzt werden konnten, z.B. OCL und XText. Die Anwendungen sind generell zu schwergewichtig für eine Virtual Box. Dennoch wird als Notfalllösung eine virtuelle Maschine mit dem Interpreter gebaut, um der QA im Ernstfall einen Zugang zu ermöglichen.
- Eclipse Sirius ist schlecht dokumentiert.
- JIRA wurde in den Gruppen selbst etwas zu selten eingesetzt. Hier wird versucht, mehr JIRA Oberaufgaben in der Sprintplanung zu identifizieren.
- Die Kommunikationsprobleme innerhalb der Abteilungen SWT/VLBA in Bezug auf Anwendungsbeispiele haben die QA aufgehalten.
- Es gab Probleme mit Git, insbesondere bei der Nutzung von Branches oder fehlerhaft durchgeführten Merges, bei denen Daten gelöscht wurden und somit fehlten.
- Die QA hatte sehr viele Fehler und Benutzungsprobleme mit der Live-Version des Editors. Um dies zu vermeiden, bekommt die QA, nur noch deployte Versionen zur Verfügung gestellt, auch wenn dadurch auf neue Features seitens der QA nicht direkt zugegriffen werden kann.

Als positiv empfunden wurden folgende Aspekte:

- Die QA Debug Session mit lief gut, die Testcases wurden konzentriert durchgearbeitet
- So gut wie alle Sprintziele erreicht
- Die Arbeit in Kleingruppen (bis zu 3 Personen)
- JIRA wurde mehr genutzt
- Die Hilfsbereitschaft in der Gruppe
- Guter Fortschritt innerhalb des Sprints
- Gute und realistische Planung

16. Sprint 6

Hannah

Planung

Der sechste offizielle Sprint der Projektgruppe startete am 02. Januar 2018 und war der Entwicklung der zweiten Plattform Android gewidmet. Hier sollte ebenfalls eine automatisierte Durchführung der TestCases ermöglicht werden. Im JSF-Interpreter sollte das Error-Handling und im Editor die konkrete Syntax und Usability verbessert werden. Beteiligt waren an diesem Sprint zu jeder Zeit sämtliche Mitglieder der Projektgruppe bis auf die in Tabelle 16.1 aufgeführten Mitglieder, welche eine Zeit lang Urlaub genommen hatten. Sprint 6 wurde für einen Monat angesetzt und am 28. Januar 2018 beendet.

Tab. 16.1.: Urlaubszeiten in Sprint 6

PG-Mitglied	Urlaubsbeginn	Urlaubsende
Hannah Meyer	01. Januar 2018	07. Januar 2018
Jan Philipp Hörding	01. Januar 2018	07. Januar 2018
Christopher Bischopink	01. Januar 2018	07. Januar 2018

Ziele

- Android als zweite Plattform
- Nachziehen und Automatisieren der TestCases auf Android
- Usability im Editor
  - Anpassen der Poperty Views

  - Automatische Erstellung von Events und Variablen beim Mapping einer AbstractWidgetInstance auf ein AbstractWidget
  - Automatische ID-Erstellung
- Konkrete Syntax im Editor
- Error-Handling im JSF-Interpreter

Durchführung

Die Plattform Android wurde um REST und Collections erweitert, wobei während der Weihnachtferien schon grundlegende Funktionalitäten wie etwa das Wechseln eines (beliebig geschachtelten) Widgets in ein anderes via Event umgesetzt worden war. Im Bereich der Guards wurden Fehler behoben und diese um beliebig tief verschachtelte CompositeDataTypes ergänzt. Der JSF-Interpreter wurde um eine aussagekräftige Error-Page erweitert, auf der im Falle einer Exception zum Beispiel Variablen und deren Werte sowie die aktuellen States und die Eventqueue angezeigt werden. Auch kann in diesem Interpreter nun das hochgeladene Archiv vor Ausführung auf gewisse Fehler untersucht werden. Das Editorhandling und -deployment wurden verbessert, ersteres etwa durch Einführung von Custom Property Views und der Möglichkeit, Property Views durch Doppelklick im Diagramm zu öffnen. Die konkrete Syntax wurde auf Grundlage der Interviews des vorherigen Sprints überarbeitet und entsprechend der Wünsche und Möglichkeiten angepasst. Für die Android-TestCases wurden GUIs entwickelt und die Testmodelle entsprechend ergänzt. Auch in Android wurde eine Möglichkeit für automatisiertes Testen geschaffen und die funktionierenden TestCases automatisiert. Die Konzeption des Anwendungsbeispiels wurde finalisiert und AbstractWidgets dafür erstellt.

Review

- Da es Teile im Metamodell gibt, welche vom Interpreter/Editor nicht unterstützt werden, soll es nun eine weitere abgespeckte Version von diesem geben. Zudem ist darauf zu achten, das Metamodell im Hinblick auf Erweiterung zu dokumentieren.
- Es sind bereits einige TestCases vorhanden, die sowohl auf Web als auch Android durchlaufen.
- Im Core-Interpreter wurden krankheitsbedingt die HistoryStates gestrichen.

- Über drei Interfaces können die plattformspezifischen Interpreter der Core-Interpreter verwenden, die Guards werden vom Core-Interpreter nativ und vom Editor über REST beherrscht. Hierzu ist eine Architekturübersicht gewünscht.
- Die konkrete Syntax und die Properties Views wurden im Editor überarbeitet, zudem ist dort nun eine Filterung einzelner Diagrammelemente möglich. Beim Erstellen einer Transition wird automatisch eine PBG erstellt, welche nun Container für PB, Action und Trigger darstellt. Zusätzlich wurde ein Automatismus eingefügt, der auf Basis eines Mappings direkt die benötigten Elemente erstellt.

Retrospektive

Als Probleme während des sechsten Sprints und deren Lösungsansätze wurden die folgenden Punkte vermerkt:

- Nach Änderungen im Interpreter wurden trotz vorhandener Selenium-Test diese nicht durchgeführt, ab jetzt soll dies bei Änderungen am Core-Interpreter immer geschehen.
- Thomas hat in Bezug auf Android viel geleistet, jedoch ist eine Einarbeitung nun für den Rest der Gruppe schwierig. In Zukunft soll mehr im Team vorangeschritten werden.
- Eine Syntax-Änderung wurde übereilt getroffen. Diese stellte sich als redundant heraus, zog aber trotzdem Komplikationen nach sich. In Zukunft müssen derartige Änderungen überlegter umgesetzt werden.
- Es wurde viel über NEMo diskutiert, obwohl früh klar war, dass damit nicht das Anwendungsbeispiel umsetzt wird. Die Projektleitung kümmerte sich darum.
- Dokumente im Confluence müssen ab sofort aktualisiert werden oder mit einer Warnung versehen werden, dass sie veraltet sind.

Als positive Aspekte wurden folgende Punkte vermerkt:

- Modularer Interpreter ermöglicht leichte Weiterentwicklung
- „Erste Hilfe“ unter Gruppenmitgliedern, etwa zum Interpreter und den Guards, bei der Bereitstellung eines aktuellen Interpreter-Deployments für die QA und bei der Modellierung des Anwendungsbeispiels
- Spontane Android-Präsentation vor einem Gast aus Kapstadt
- Kommunikation des Projektfortschritts, insbesondere sonntägliche Rückmeldung
- Verbesserte Grafik und Bedienung des Editors

17. Sprint 7

Hannah

Planung

Der siebte offizielle Sprint der Projektgruppe startete am 05. Februar 2018 und war dem Nachziehen der Funktionalität des Android-Übersetzers, der Umsetzung des Anwendungsbeispiels, der finalen Guard-Einbindung und dem Usability-Feinschliff gewidmet. Zudem sollte – wann immer möglich – der Fokus auf die Dokumentation der Projektgruppe gelegt werden. Beteiligt waren an diesem Sprint zu jeder Zeit sämtliche Mitglieder der Projektgruppe bis auf die in Tabelle 17.1 aufgeführten Mitglieder, welche eine Zeit lang Urlaub genommen hatten. Sprint 7 wurde für einen knappen Monat angesetzt und am 25. Februar 2018 beendet. Da von der Projektgruppe ein nahtloser Übergang in den letzten Sprint gewünscht wurde, entfällt in diesem Sprint die formelle Durchführung von Review und Retrospektive.

Ziele

- Einbinden fehlender Constraints ins Metamodell
- Dokumentation der konkreten Syntax

Tab. 17.1.: Urlaubszeiten in Sprint 7

PG-Mitglied	Urlaubsbeginn	Urlaubsende
Hannah Meyer	28. Januar 2018	19. Februar 2018
Jan Philipp Hörding	28. Januar 2018	11. Februar 2018
Nancy Kramer	20. Februar 2018	11. März 2018
Felix Kempa	20. Februar 2018	11. März 2018

- Tooleinbindung der Guards
- Benutzer- und Entwicklerdokumentation des Editors
- Usability-Feinschliff im Editor
- Überarbeiten der Code-Qualität im Core-Interpreter
- Testen und Dokumentation des Core-Interpreters
- Dialoge für den JSF-Interpreter
- Deployment des Android-Interpreters und Nachziehen auf Testcase-Basis
- Umsetzung des Anwendungsbeispiels
- Nachziehen der automatisierten TestCases

Durchführung

Das Guard-Handling wurde für Android überarbeitet, da es Probleme mit XText auf Android gab. Das Anwendungsbeispiel wurde im Editor modelliert und für Web, jedoch noch nicht für Android umgesetzt. Die Dokumentation wurde vorangetrieben. Zudem wurde zusammengetragen, welche Komponenten und Thematiken bei einer Weiterführung von DORI aufgegriffen werden müssten. Zusammenfassend lässt sich festhalten, dass für den letzten Sprint die letzten beiden obig genannten Ziele – das Anwendungsbeispiel in Teilen sowie alle ausstehenden automatisierten TestCases – sowie eine Weiterführung der Dokumentation allgemein ausstehen.

Sonstiges

Jan Philipp Hörding verließ im Sprint die Projektgruppe auf eigenen Wunsch. Dem vorangegangen waren Differenzen bezüglich geleisteter Arbeit und Erwartungen. Diese wurden innerhalb der Gruppe versucht zu klären, auch wurde die Unterstützung durch die Betreuer in dieser Thematik in Anspruch genommen.

18. Sprint 8

Hannah

Planung

Der achte offizielle Sprint der Projektgruppe startete am 26. Februar 2018. Da dieser der letzte Sprint dieser Projektgruppe war und im Lauf dessen dieses Abgabedokument fertiggestellt und in Druck gegeben wurde, ist an dieser Stelle kein(e) Review/Retrospektive aufgeführt. Der Fokus dieses Sprints lag auf der Fertigstellung und Korrektur der schriftlichen Abgabedokumente sowie der Vorbereitung auf die Abschlusspräsentation. Beteiligt waren an diesem Sprint zu jeder Zeit sämtliche Mitglieder der Projektgruppe bis auf die in Tabelle 12.1 aufgeführten Mitglieder, welche eine Zeit lang Urlaub genommen hatten. Sprint 8 wurde für einen Monat angesetzt und am 28. März 2018 beendet. Am 28. März fand die Abschlusspräsentation der Projektgruppe statt.

Tab. 18.1.: Urlaubszeiten in Sprint 8

PG-Mitglied	Urlaubsbeginn	Urlaubsende
Stephan Bogs	05. März 2018	11. März 2018

Ziele

- Finale Fassung der Projektdokumentation
- Finale Fassung der Benutzerdokumentation
- Nachziehen des Anwendungsbeispiels auf Android
- Vorbereitung und Halten der Abschlusspräsentation

Durchführung

Im letzten Sprint der Projektgruppe wurden in den Arbeitsgruppen „Interpreter“ und „Editor“ vor allem die Benutzer- und Entwicklerdokumentationen fertiggestellt. Auch wurde das Testkonzept durch die Arbeitsgruppe „QA” schriftlich festgehalten. Das Anwendungsbeispiel wurde auf Android nachgezogen und dokumentiert. Zudem wurden einzelne Teile der Dokumentation überarbeitet und an den aktuellen Stand angepasst.

Teil IV.

Testdokumentation

19. Testkonzept

Hannah

Im Folgenden wird beschrieben, wie das Testen der Systemkomponenten und des Gesamtsystems konzeptioniert und durchgeführt wurde.

Am 13.09.2017 wurde ein Seminarvortrag zum Thema „Qualitätssicherung“ angesetzt, um Möglichkeiten vor allem im Bezug auf die Entwicklung des Interpreters vorzustellen. Auf Basis dieses Vortrags wurde für den Interpreter festgelegt, dass jede public-Methode mit einem Unittest versehen werden soll. Zur besseren Nachvollziehbarkeit der Funktionalität sollte jede Klasse sowie innerhalb dieser jede public-Methode mit Kommentaren versehen werden. Zur Sicherung der Codequalität sollten zudem Code Conventions eingehalten werden. Eine einheitliche Code Formatierung konnte leicht, durch Installation eines CodeFormaters erreicht werden.

Im Laufe der Projektgruppe wurde von dem Konzept „Unittest für jede public-Methode“ abgewichen. Einerseits stellte sich heraus, dass das dafür benötigte Mocking Modell, auf dessen Grundlage die Unittests liefen und welches in der Erstellung bereits unverhältnismäßig aufwändig war, nicht mit vertretbarem Aufwand an Aktualisierungen im Metamodell angepasst werden konnte. Zusätzlich wären die zu testenden Vor- und Nachbedingungen der Unittests außerordentlich komplex gewesen, da eine Methode Auswirkungen auf einen Großteil des Zustands des Modells haben kann. Aus diesen Gründen entwickelte sich das Testen des Interpreters in die Richtung von Highlevel-Integrationstests. Dies wurde durch die Einführung der Arbeitsgruppe „QA“ bewusst fokussiert, welche Testmodelle für diesen Zweck erstellte. Händisches Testen auf Codebasis wurde während des Entwickelns ebenfalls eingesetzt.

Bei der Entwicklung der Guards wurde zunächst zum Testen mit der manuellen Übergabe von GuardStrings an die XText-Grammatik gearbeitet. Während der Integration in den Core-Interpreter wurde zu den von der QA entwickelten Testmodellen gewechselt.

Während der Implementierung des DORI-Editors konnte häufig nur an einem einzelnen Rechner gearbeitet werden, da die Konfiguration des Editors über eine einzige Datei stattfand. Daher wurden die von einer Person der Editor-Gruppe entwickelten Funktionalitäten und Services von anderen Editor Gruppenmitgliedern verifiziert. Dazu wurden Wegwerfmodelle erstellt. Das im Laufe der Projektgruppe erstelle Anwendungsbeispiel *Telefonbuch*

diente als Basis. Anschließend wurden abgeschlossene Features an die QA zum Review übergeben.

Die Testmodelle der QA wurden auf Basis der Anforderungen, Wünschen der einzelnen Arbeitsgruppen sowie aufgetretener Fehler entwickelt. Dadurch, dass die Testmodelle im Editor entwickelt und mit dem Interpreter ausgeführt wurden, konnte gleichzeitig die Funktionalität und Usability beider Komponenten inspiziert werden. Darüber hinaus konnte die QA dadurch als Bindeglied und Kommunikationsmedium zwischen den Arbeitsgruppen fungieren. Da den Testmodellen eine herausragende Bedeutung im Testprozess der Projektgruppe zukommt, sind diese im Detail im Kapitel 20 beschreiben.

20. Testmodelle

Nancy, Hannah

Um die Funktionalitäten des Editors und der Interpreter zu testen, wurden verschiedene Testmodelle anhand der Anforderungen und der aufgestellten Use-Cases entworfen. Diese wurden im Arbeitsbereich des DORI-Editors erstellt. Zusätzlich wurden für diese Modelle die benötigten .xhtml-Dateien und REST Services für die Plattform JSF erstellt. Für die Plattform Android wurden ebenfalls entsprechende GUIs angelegt. Das Anlegen von neuen REST Services war für diese Plattform nicht nötigt, da die selben Services verwendet werden konnten. Eine Beschreibung der nötigen Arbeitsschritte für die Erstellung der GUIs und REST Services befindet sich in der Benutzerdokumentation in Kapitel 6. Vor der Auseinandersetzung mit den Testmodellen empfiehlt sich eine genaue Beschäftigung mit der DORI-DSL, welche zum Beispiel in der Benutzerdokumentation in Kapitel 9.1 zu finden ist.

20.1. Beschreibung der Modelle

Nancy, Hannah

Im Folgenden werden die Modelle beschrieben. Dazu gehört eine genauere Erläuterung, welche Funktionalität dieser Testfall abdecken soll. Eine Abbildung zu jedem Testfall zeigt die syntaktische Darstellung des Modells im Editor.

Testcase001 Dieser Testfall enthält zwei einfache Widgets. Von dem ersten Widget kann mittels einer Transition in das andere Widget gewechselt werden. Der Wechsel wird durch einen Trigger ausgelöst. Ein Wechsel zurück in das erste Widget ist bei diesem Testfall nicht gegeben. Ebenfalls wurden für diesen Testfall keine Funktionen benötigt. Abbildung 20.1 zeigt die syntaktische Darstellung von Testcase001.

Abb. 20.1.: Interaktionsdiagramm von Testcase001

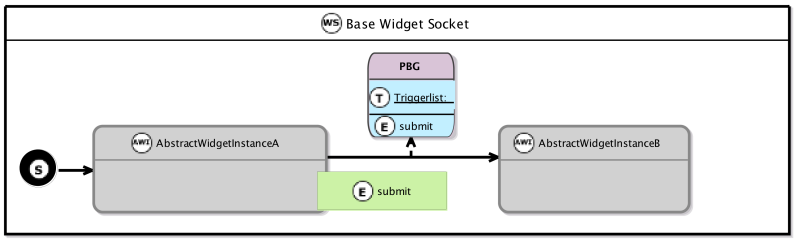
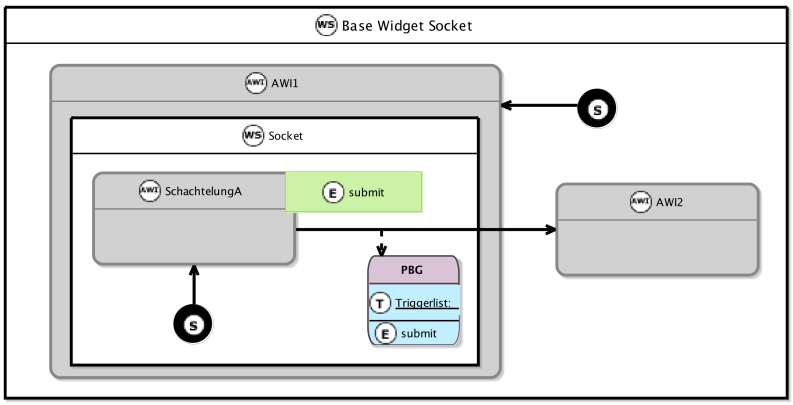


Abb. 20.2.: Interaktionsdiagramm von Testcase002

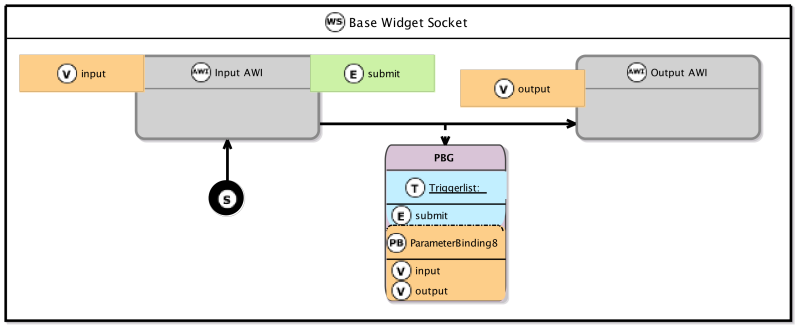


Testcase002 Dieser Testfall verfeinert den Testcase001. Es gibt ebenfalls einen Übergang von einem Widget zu einem anderen. Das erste Widget besitzt hier eine Schachtelung. Die Transition führt von dem inneren geschachtelten Widget zu einem nicht geschachtelten Widget. Wie beim ersten Testfall wurden hier keine Funktionen verwendet. Abbildung 20.2 zeigt die syntaktische Darstellung von Testcase002.

Testcase003 In diesem Testfall wurde ein einfaches ParameterBinding getestet. Die Widgets sind nicht geschachtelt. Es gibt eine Eingabe (String), die an das zweite Widget als Ausgabe übergeben wird. Es wurden keine Funktionen verwendet. Abbildung 20.3 zeigt die syntaktische Darstellung von Testcase003.

Testcase004 Dieser Testfall soll das Layerkonzept testen. Dazu gibt es in diesem Testcase zwei Widgets zwischen denen gewechselt werden kann. Beide besitzen eine tiefer gehende Schachtelung. In dem ersten Widget ist ein weiteres Widget drin geschachtelt. In diesem geschachtelten Widget befindet sich eine komplexere Interaktion. Es befinden sich in diesem Widget zwei parallele WidgetSockets. In dem einem der beiden parallelen WidgetSockets

Abb. 20.3.: Interaktionsdiagramm von Testcase003



(hier einmal WidgetSocket 1 genannt) befindet sich eine weitere Schachtelung. In dieser Schachtelung befindet sich ein Widget, welches eine Selbsttransition besitzt mit einer Action, die ein Event wirft. In dem anderen parallenen WidgetSocket (hier einmal WidgetSocket 2 genannt) befinden sich zwei Widgets zwischen denen gewechselt werden kann. Dabei kann von dem einen Widget nur in das andere Widget gewechselt werden, wenn die Action in WidgetSocket 1 aktiviert wird. Ein Wechsel zurück geschieht über ein normales Event. Es kann aber auch ein zweites Event geworfen werden, was einen Wechsel in das zweite große Widget auslöst. Dieses Widget ist 3-Fach geschachtelt und es wird durch die Transition in das innerste Widget gewechselt. Die Abbildung 20.4 zeigt die syntaktische Darstellung von Testcase004.

Testcase005 Dieser Testfall zeigt den Übergang von einem Widget ins nächste mittels eines Events. Es gibt sowohl einen Übergang von einem inneren als auch von einem übergeordneten Widget in jeweils ein anderes, äußeres Widget. Beide Übergänge hören dabei auf ein Event mit gleichem Namen. Dieser Testfall soll zeigen, dass der Interpreter zwischen Events gleichem Namens differenzieren kann, sofern diese aus unterschiedlichen Widgets stammen. Abbildung 20.5 zeigt die syntaktische Darstellung von Testcase005.

Testcase006 Dieser Testfall ähnelt dem Testcase002. Hier soll die Transition nun von einem ungeschachtelten Widget in ein anderes inneres geschachteltes Widget erfolgen. Mit diesem Testfall wird das Layer-Konzept getestet. Das überliegende Widget muss von dem Interpreter erkannt und zusätzlich aufgebaut werden. Abbildung 20.6 zeigt die syntaktische Darstellung von Testcase006.

Testcase007 Bei diesem Testfall sollte das ParameterBinding genauer getestet werden. Dazu gibt es zwei Widgets. Das erste Widget besitzt eine Schachtelung. Die Schachtelung

Abb. 20.4.: Interaktionsdiagramm von Testcase004

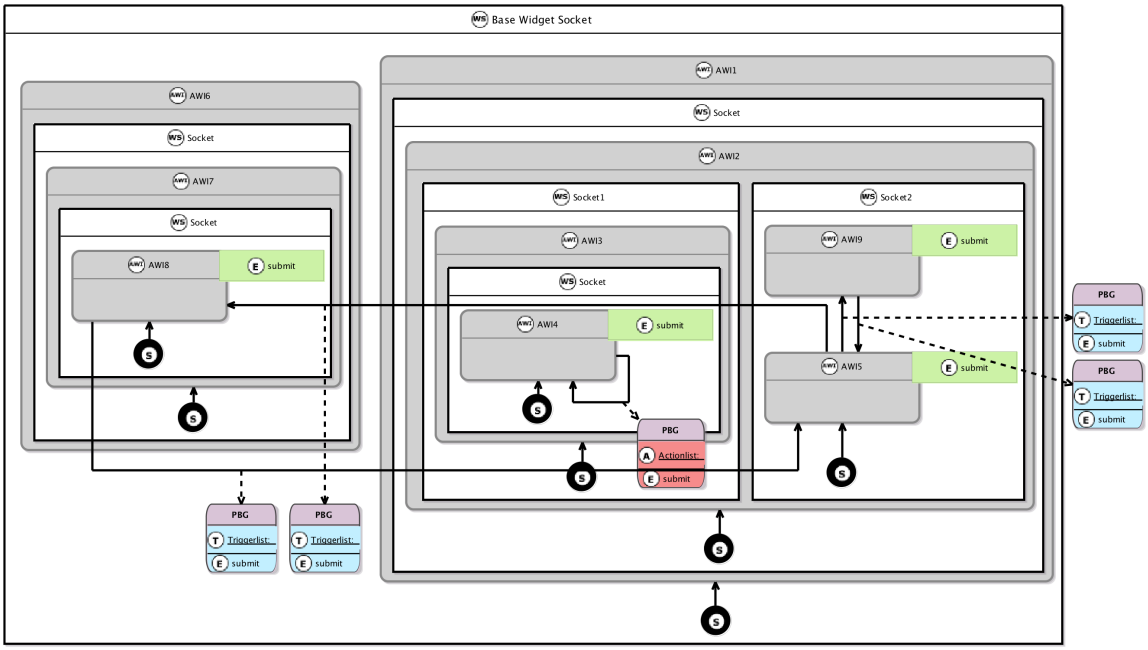


Abb. 20.5.: Interaktionsdiagramm von Testcase005

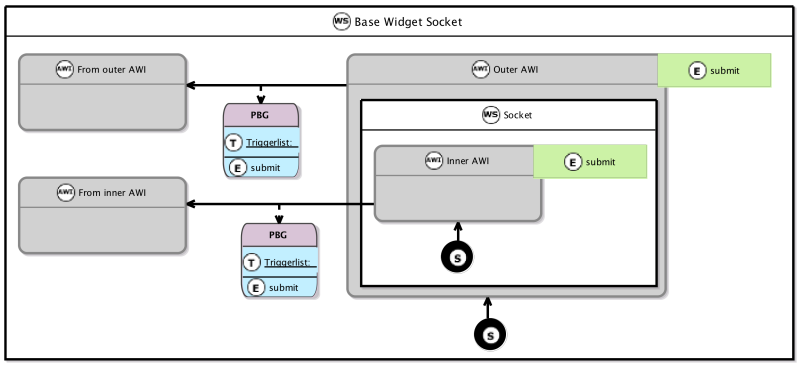


Abb. 20.6.: Interaktionsdiagramm von Testcase006

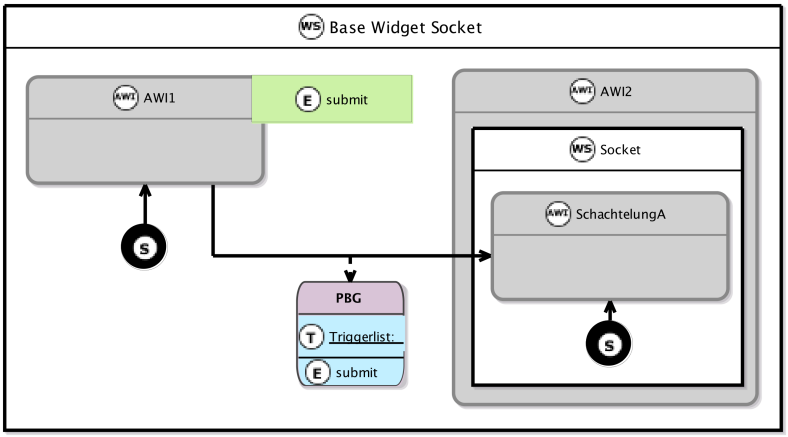
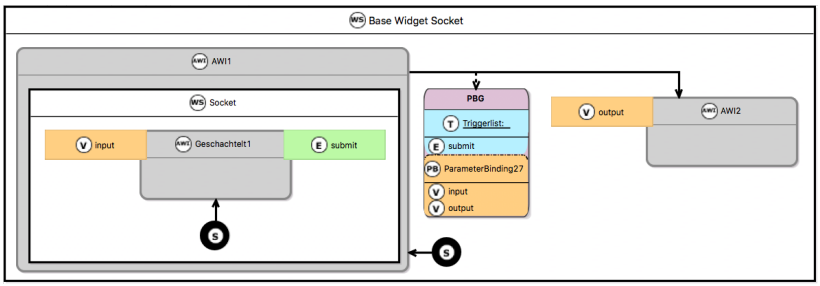


Abb. 20.7.: Interaktionsdiagramm von Testcase007

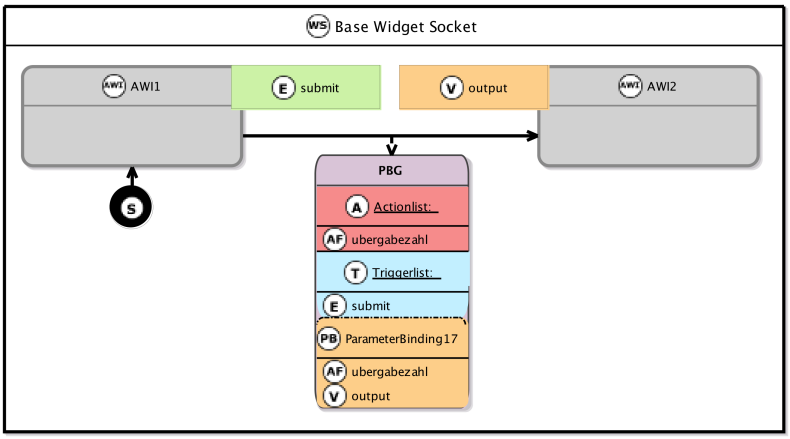


besitzt eine String-Variable, die durch ein ParameterBinding an das nächste Widget übergeben werden soll. Dabei geht allerdings die Transition von dem übergeordneten Widget aus. Getestet werden soll hier, ob auch das übergeordnete Widget die Daten von seinen Schachtelungen kennt und weitergeben kann. Abbildung 20.7 zeigt die syntaktische Darstellung von Testcase007.

Testcase008 Bei diesem Testfall gibt es zwei ungeschachtelte Widgets. Bei dem Wechsel mittels eines Triggers wird eine Funktion aufgerufen. Dieser Testfall dient der Überprüfung zur Erstellung einer Funktion im Editor sowie der Verarbeitung von Funktionen im Interpreter. Die Funktion gibt stets den Integer 42 zurück. Abbildung 20.8 zeigt die syntaktische Darstellung von Testcase008.

Testcase009 Dieser Testfall verfeinert den Testcase008. Bei diesem Testfall werden mittels einer Funktion zwei Integer-Variablen addiert und die Summe an das nächste Widget

Abb. 20.8.: Interaktionsdiagramm von Testcase008



weitergegeben. Die Funktion „summation“ rechnet zwei Integer zusammen. Abbildung 20.9 zeigt die syntaktische Darstellung von Testcase009.

Testcase010 Der Testfall besteht aus drei Widgets. Wird im ersten Widget ein Event ausgelöst, so wird in das zweite Widget gewechselt. Gleichzeitig wird während dieser Transition ein weiteres Event geworfen, was zu einem Wechsel vom zweiten Widget ins dritte Widget führt. Somit befindet man sich nie aktiv im zweiten Widget. Abbildung 20.10 zeigt die syntaktische Darstellung von Testcase010.

Testcase011 Der Testfall zeigt ein Widget mit zwei WidgetSockets. Wird in dem Widget des ersten WidgetSockets ein Event ausgelöst, so soll hier das Widget wechseln. Gleichzeitig wird während dieser Transition ein weiteres Event geworfen, was zu einem Wechsel des Widgets im zweiten WidgetSocket führen soll. Abbildung 20.11 zeigt die syntaktische Darstellung von Testcase011.

Testcase012 Dieser Testfall beschreibt den Wechsel von einem in ein anderes ungeschachteltes Widget, wobei während der Transition eine Funktion ausgeführt wird. Der wesentliche Unterschied zum Testfall009 besteht darin, dass die Funktion einen Wert vom Typ Collection<DString> zurückgibt, welcher vom zweiten Widget angezeigt wird. Die Funktion durchsucht eine Liste mit Worten, ob der Eingabestring in Worten der Liste enthalten ist. Die Rückgabe zeigt die Wörter, die diesen String enthalten. Abbildung 20.12 zeigt die syntaktische Darstellung von Testcase012.

Abb. 20.9.: Interaktionsdiagramm von Testcase009

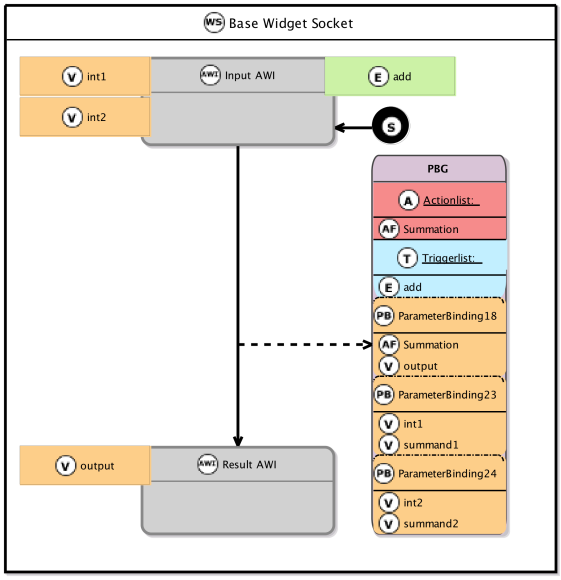


Abb. 20.10.: Interaktionsdiagramm von Testcase010

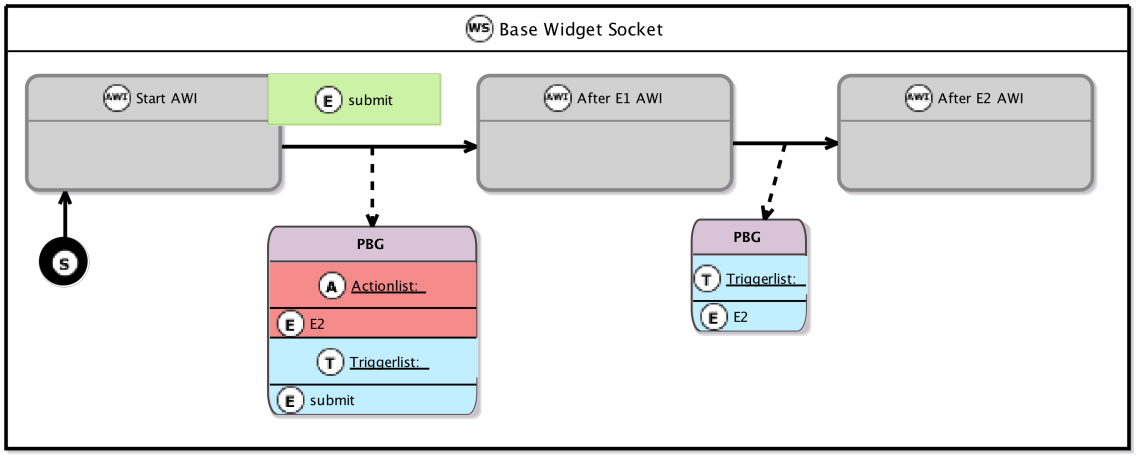


Abb. 20.11.: Interaktionsdiagramm von Testcase011

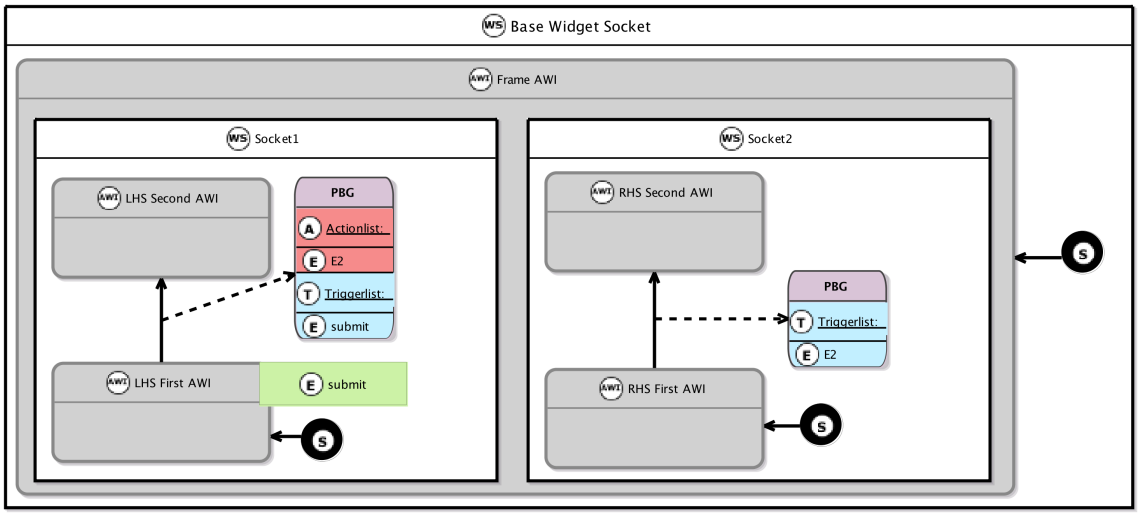


Abb. 20.12.: Interaktionsdiagramm von Testcase012

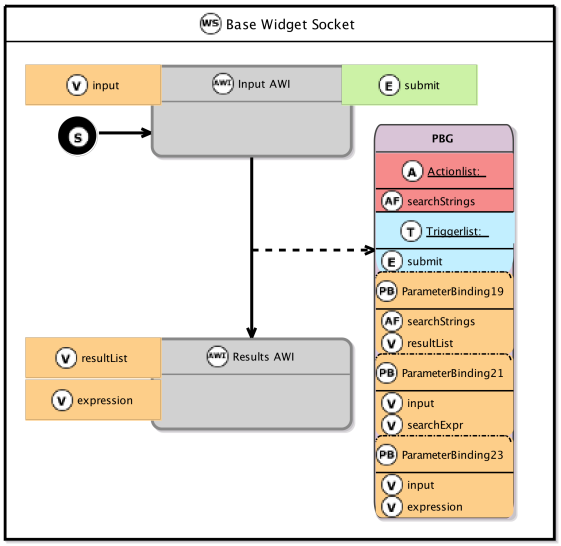
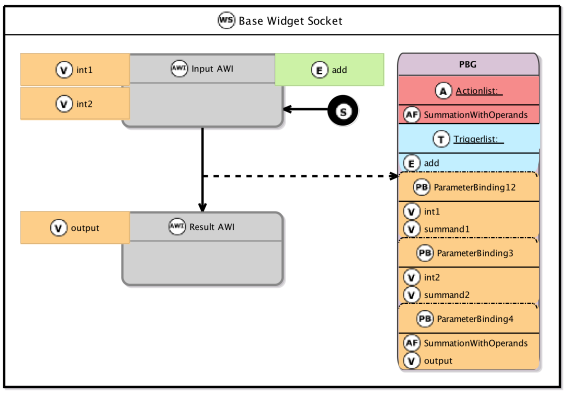


Abb. 20.13.: Interaktionsdiagramm von Testcase013



Testcase013 Dieser Testfall beschreibt den Wechsel von einem in ein anderes ungeschachteltes Widget, wobei während der Transition eine Funktion ausgeführt wird. Der wesentliche Unterschied zum Testcase009 und Testcase012 besteht darin, dass die Funktion einen Wert vom Typ CompositeDatatype bestehend aus drei Integer-Werten zurückgibt, welcher vom zweiten Widget angezeigt wird. Abbildung 20.13 zeigt die syntaktische Darstellung von Testcase013.

Testcase014 Bei diesem Testfall werden die Guards getestet. Im ersten Widget soll ein Integer eingegeben werden. Nach dem Event läuft das Modell in einen DecisionState und es folgt eine Guard-Abfrage, ob der eingegeben Wert größer oder kleiner fünf ist. Je nach Eingabe wird in unterschiedliche Widgets gewechselt. Abbildung 20.14 zeigt die syntaktische Darstellung von Testcase014.

Testcase015 Dieser Testfall testet nochmal einen einfachen Widgetwechsel innerhalb eines geschachtelten Widgets. Dieser Testfall dient zur Überprüfung des Layerkonzeptes. Trotz Wechsel soll das äußere Widget das selbe bleiben und nicht neu geladen werden. Abbildung 20.15 zeigt die syntaktische Darstellung von Testcase015.

Testcase016 Testcase016 überprüft das Layerkonzept noch weiter. Dadurch, dass der Testcase004 sehr groß ist, werden die einzelnen Funktionalitäten noch einmal separat getestet. Bei diesem Testfall wird der Übergang von einem parallel geschachtelten Widget in ein anderes Widget getestet. Abbildung 20.16 zeigt die syntaktische Darstellung von Testcase016.

Abb. 20.14.: Interaktionsdiagramm von Testcase014

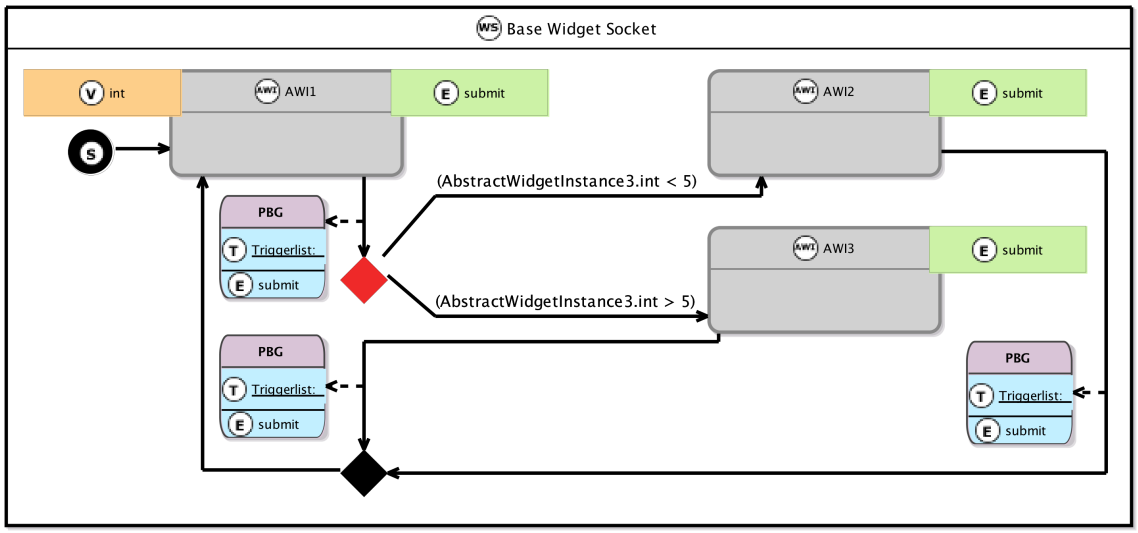


Abb. 20.15.: Interaktionsdiagramm von Testcase015

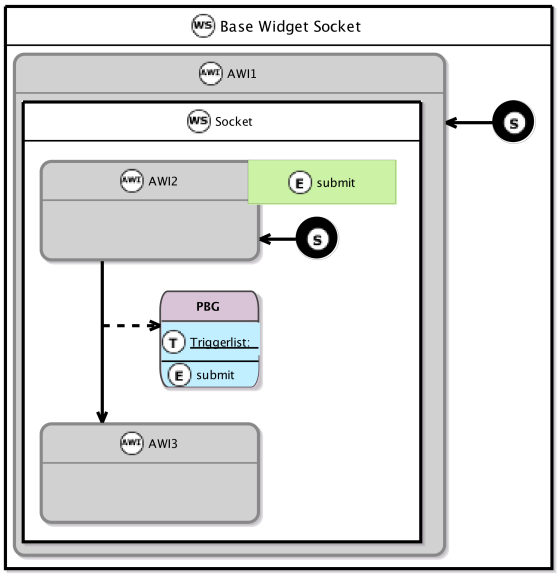
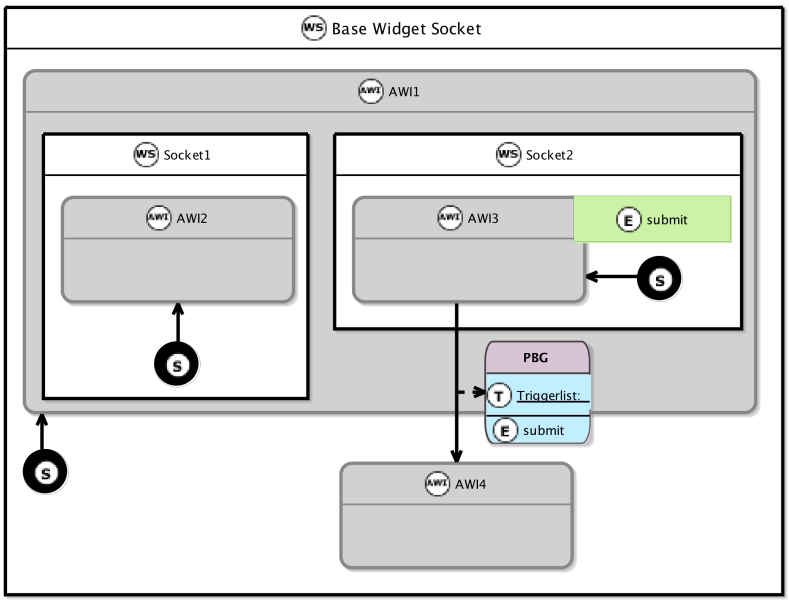


Abb. 20.16.: Interaktionsdiagramm von Testcase016



Testcase017 Testcase017 weist Ähnlichkeiten zu Testcase014 auf. Bei beiden Testfällen werden die gleichen Guards verwendet. Der Unterschied besteht darin, dass bei diesem Testfall von einem geschachtelten Widget gestartet wird und es außerdem die Möglichkeit gibt, das Widget zu wechseln, ohne in die Guard-Abfrage zu gelangen. Wenn die Eingabe kleiner als fünf ist wird außerdem direkt mit der Transition in ein inneres geschachteltes Widget gewechselt und das äußere Widget muss zusätzlich aufgebaut werden. Abbildung 20.17 zeigt die syntaktische Darstellung von Testcase017.

Testcase018 Dieser Testfall soll den Gleich- und Ungleich-Operator in Guards testen. Nach einer Integer-Eingabe im ersten Widget wird abgefragt, ob die Eingabe gleich oder ungleich 42 ist. Je nach Ergebnis wird in unterschiedliche Widgets gewechselt. Die Eingabe wird dabei mit übergeben und in dem jeweiligen Widget angezeigt. Abbildung 20.18 zeigt die syntaktische Darstellung von Testcase018.

Testcase019 In dem Testcase019 werden Integer-Literale getestet. Bei einem einfachen Übergang von einem Widget ins nächste wird der Variablen vom zweiten Widget der Wert 5 zugewiesen. Abbildung 20.19 zeigt die syntaktische Darstellung von Testcase019.

Abb. 20.17.: Interaktionsdiagramm von Testcase017

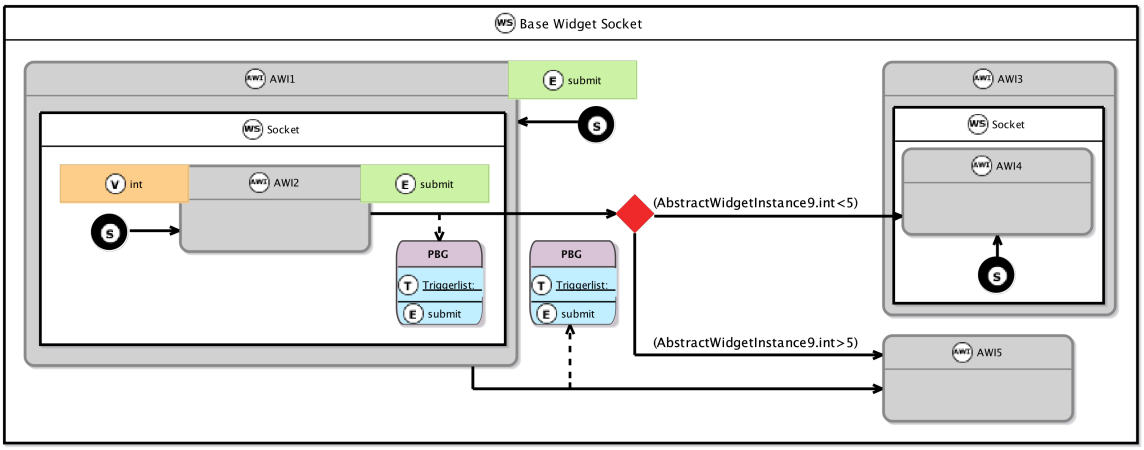


Abb. 20.18.: Interaktionsdiagramm von Testcase018

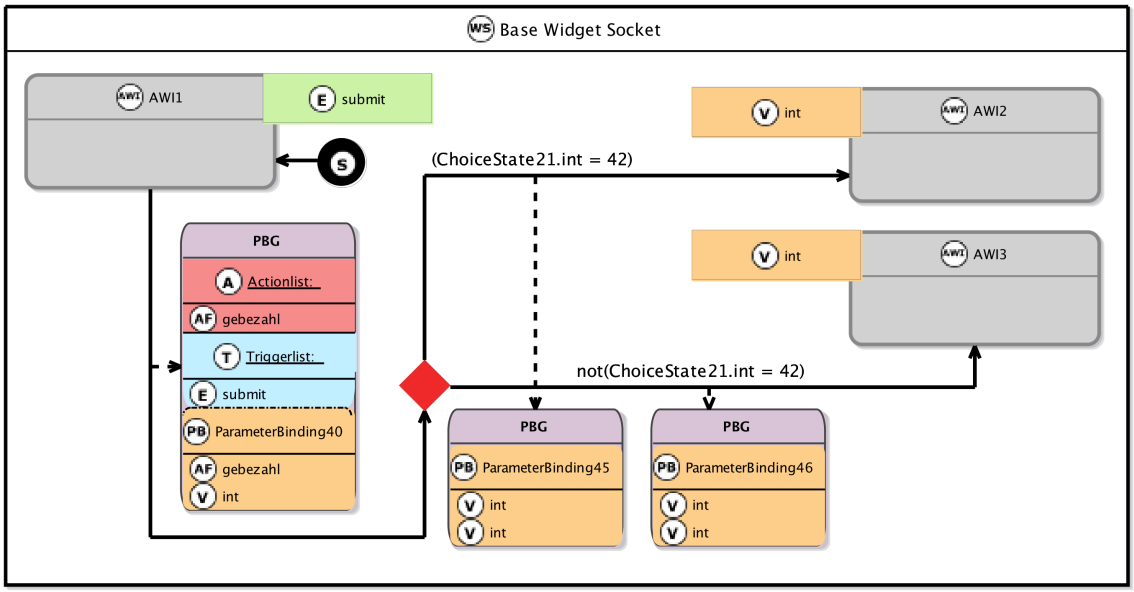
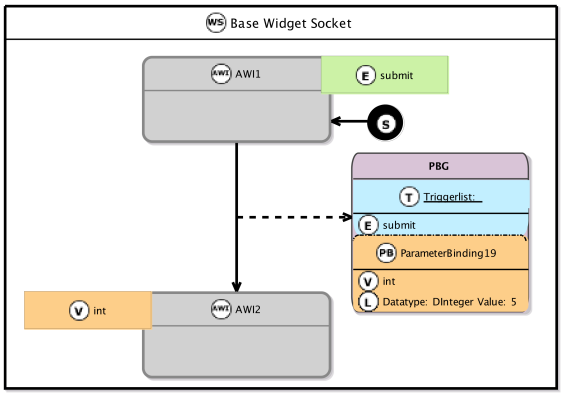


Abb. 20.19.: Interaktionsdiagramm von Testcase019



Testcase020 Dieser Testfall ähnelt dem Testcase019. Auch hierbei werden Literale getestet, es handelt sich jedoch um String-Literale. Es findet ein Wechsel von einem inneren Widget zu einem zweiten normalen Widget statt. Dabei wird der Variablen des zweite Widgets der Wert „Hallo“ übergeben. Abbildung 20.20 zeigt die syntaktische Darstellung von Testcase020.

Testcase021 Auch dieser Testfall testet die Literale, in diesem Fall Boolean-Literale. Es wird von einem inneren Widget in ein anderes inneres Widget innerhalb des gleichen Widget-Sockets gewechselt. Dabei wird dem zweiten Widget der Wert „true“ übergeben. Abbildung 20.21 zeigt die syntaktische Darstellung von Testcase021.

Testcase022 Um auch den letzten primitiven Datentyp der Literale zu testen, wird in diesem Testfall das Real-Literal getestet. Dabei sind zwei Widgets parallel zueinander. Durch ein Event im ersten Widget wird ein weiteres Event geworfen, was im zweiten Widget zur Zuweisung des Wertes 2.5 führt. Abbildung 20.22 zeigt die syntaktische Darstellung von Testcase022.

Testcase023 Bei diesem Testfall sollen andere Werte als Integer in einem Guard getestet werden. Dazu kann der Nutzer in dem ersten Widget einen Boolean-Wert angeben. Danach kommt eine Abfrage, ob dieser „true“ oder „false“ ist. Je nachdem wird entweder in das Widget für die String-Eingabe oder in das Widget für die Real-Eingabe gewechselt. Bei der String-Eingabe wird geprüft, ob die Eingabe „a“ oder „b“ lautete. Je nachdem wird in ein anderes Widget gewechselt. Bei der Real-Eingabe wird geprüft ob diese größer oder kleiner als 2.5 ist. Von allen vier Endwidgets kann wieder mittels eines Events in das erste Widget gewechselt werden. Abbildung 20.23 zeigt die syntaktische Darstellung von Testcase023.

Abb. 20.20.: Interaktionsdiagramm von Testcase020

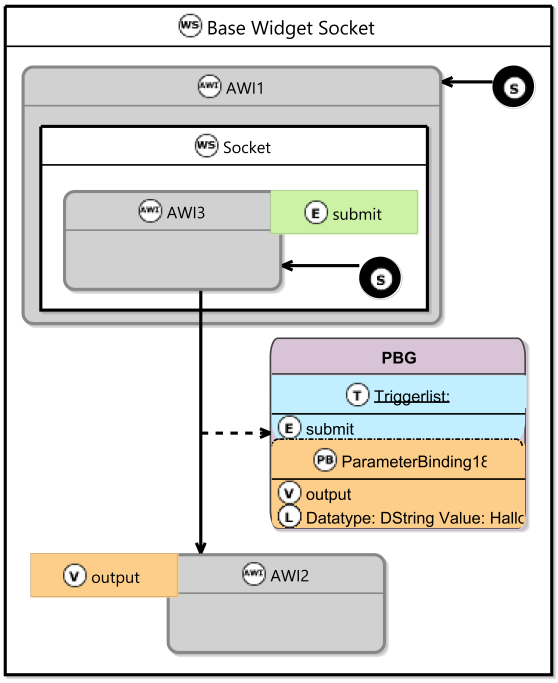


Abb. 20.21.: Interaktionsdiagramm von Testcase021

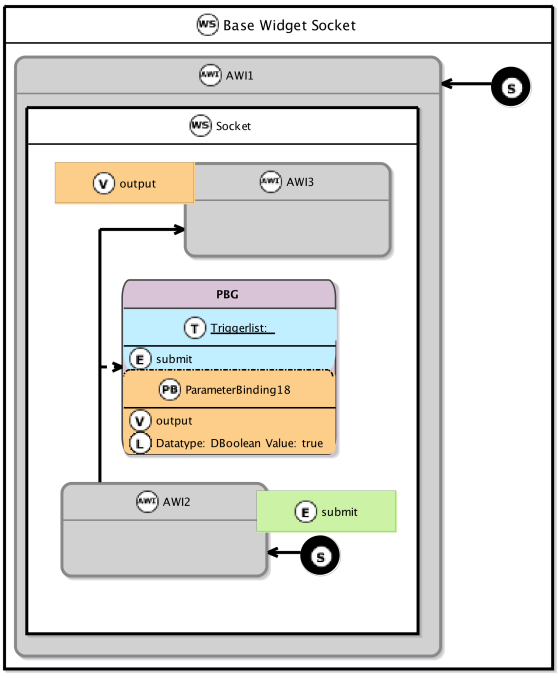


Abb. 20.22.: Interaktionsdiagramm von Testcase022

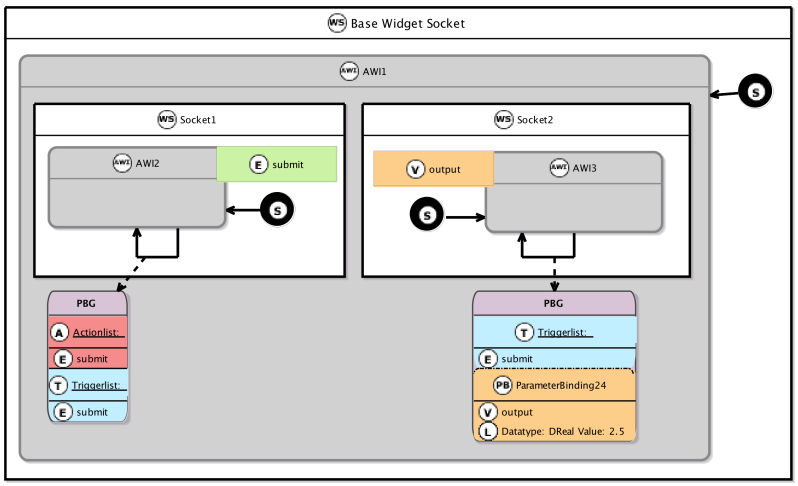


Abb. 20.23.: Interaktionsdiagramm von Testcase023

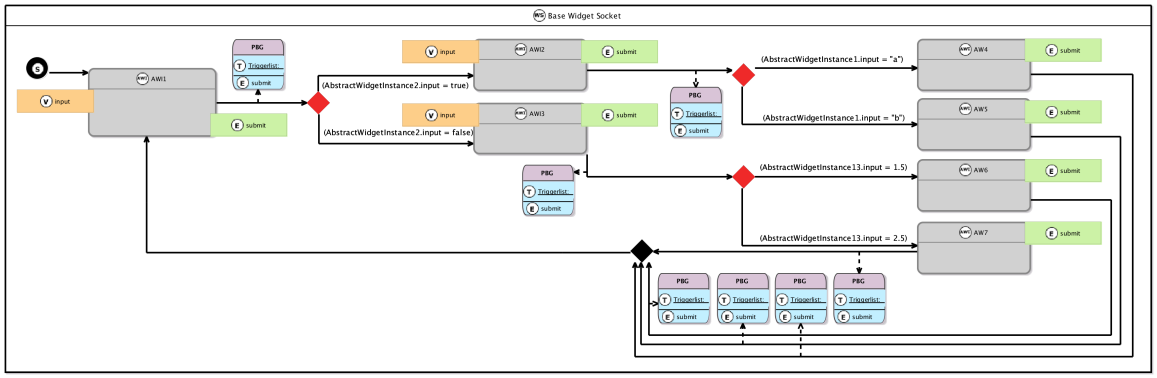
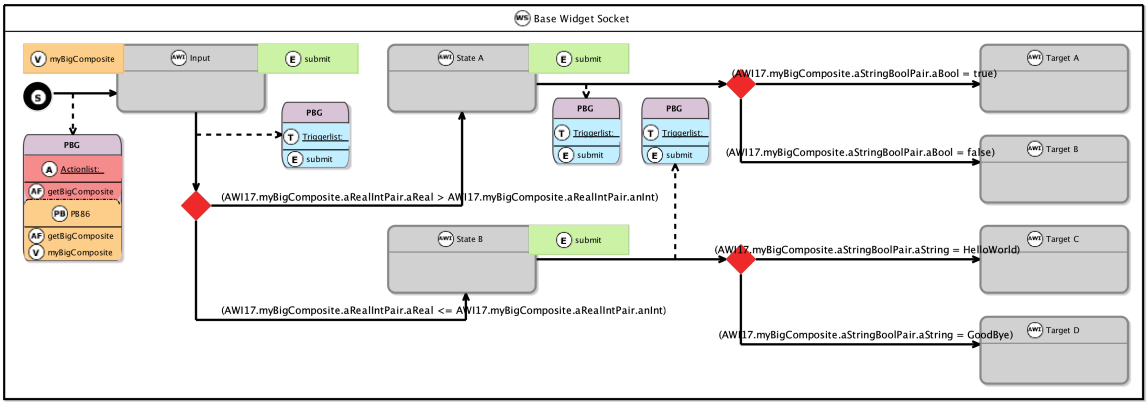


Abb. 20.24.: Interaktionsdiagramm von Testcase024



Testcase024 Dieser Testfall widmet sich CompositeDatatypes in Guards. Zudem wird erstmals eine Variable über die Start-Transition initialisiert. Dies ist in diesem Fall ein Composite-Datatype, wecher sowohl aus einem RealInt- (3.141592,42) als auch einem StringBoolPair („GoodBye“,„true“) besteht. Auf alle diese Bestandteile wird in Guards zugegriffen. Zunächst wird – je nach dem ob der Real-Wert des RealIntPairs größer oder kleiner/gleich dem Int-Wert ist – in ein spezielles Widget gewechselt. Im ersten Fall wird anschließend geprüft, ob der Bool-Wert des StringBoolPairs „true“ oder „false“ ist. Im zweiten Fall wird getestet, ob es sich bei dem String des StringBoolPairs um „HelloWorld“ oder „GoodBye“ handelt. Abbildung 20.24 zeigt die syntaktische Darstellung von Testcase024.

Testcase025 Dieser Testfall enthält eine ParameterBindingGroup, deren ParameterBindings zyklisch voneinander abhängen. Somit kann diese ParameterBindingGroup, welche beim Übergang von einem ersten in ein zweites Widget aufgerufen wird, niemals vollständig aufgelöst werden. Es soll getestet werden, ob und wie der Interpreter auf diesen Fehler im Modell reagiert. Abbildung 20.25 zeigt die syntaktische Darstellung von Testcase025.

Testcase026 Eine Variable des Widgets dieses Testfalls wird mit einem Wert eines verhältnismäßig komplexen Datentypes initialisiert. Es handelt sich um eine Liste, welche wiederum Werte eines zusammengesetzten Datentyps enthält. Der Composite DataType besteht aud einem DString und einem DInteger. Es wird hier getestet, ob und auf welche Weise die Bestandteile der Variable im Widget separat angezeigt werden können. Abbildung 20.26 zeigt die syntaktische Darstellung von Testcase026.

Testcase027 Testcase027 erweitert Testcase026 insofern, dass zusätzlich zum Anzeigen von Bestandteilen eines komplexeren Datentyps in einem Widget nun auch getestet werden

Abb. 20.25.: Interaktionsdiagramm von Testcase025

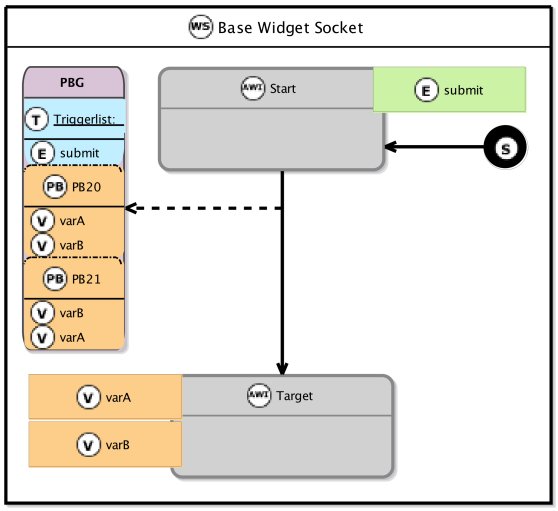


Abb. 20.26.: Interaktionsdiagramm von Testcase026

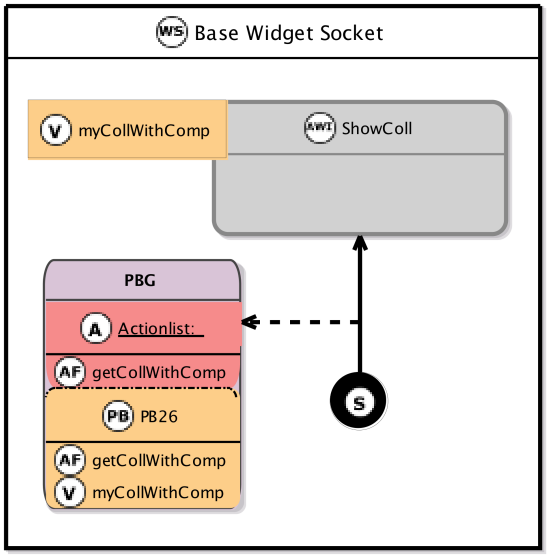
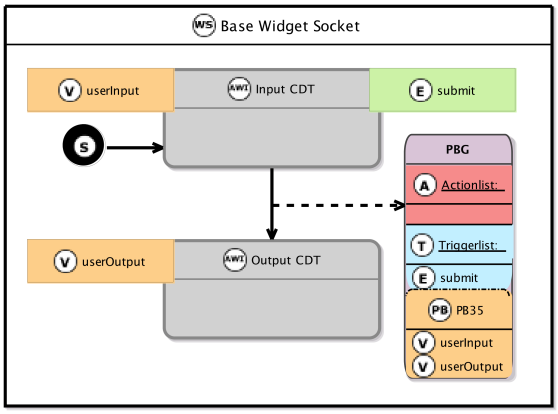


Abb. 20.27.: Interaktionsdiagramm von Testcase027



soll, ob dieser Datentyp dort gefüllt werden kann. Dazu besitzt das erste Widget eine Variable vom CompositeDataType User, welcher jeden primitiven Datentyp einmal enthält. Diese soll über das Widget gefüllt und anschließend im zweiten Widget angezeigt werden. Abbildung 20.27 zeigt die syntaktische Darstellung von Testcase027.

20.2. Abdeckung des Metamodells

Nancy, Hannah

An dieser Stelle soll belegt werden, dass sich sämtliche Elemente des Metamodells in den Testmodellen wiederfinden und somit eine komplette Testabdeckung gewährleistet ist. Abstrakte Elemente des Metamodells sind nicht aufgeführt.

Tab. 20.1.: Abgleich des Metamodells mit den Testmodellen

Element des Metamodells	Abdeckende Testmodelle
Action	TestCase008, 009, 010, 011, 012, 013, 018, 022, 024 und 026
AbstractFunction	TestCase008, 009, 012, 013, 018, 024 und 026
AbstractWidget	Alle Testmodelle
AbstractWidgetInstance	Alle Testmodelle
ChoiceState	TestCase014, 017, 018, 023, 024
Collection	TestCase012 und 026

Fortsetzung auf der nächsten Seite

Tab. 20.1.: *Fortsetzung der* Abgleich des Metamodells mit den Testmodellen

Element des Metamodells	Abdeckende Testmodelle
CompositeDataType	TestCase013, 024 und 026
ConcreteFunction	Alle Testmodelle, welche eine AbstractFunction besitzen
ConcreteWidget	Alle Testmodelle
DBoolean	TestCase021, 023, 024, 025 und 027
DInteger	TestCase008, 009, 013, 014, 017, 018, 019, 024, 026 und 027
DoriModel	Alle Testmodelle
DReal	TestCase022, 023, 024 und 027
DString	TestCase003, 007, 012, 020, 023, 024, 026 und 027
Event	Alle Testmodelle, dabei werfen TestCase010, 011 und 022 ein Event in einer Action
EventName	Alle Testmodelle
Guard	TestCase014, 017, 018, 023 und 024
Literal	TestCase019, 020, 021 und 022
MergeState	TestCase014, 023
Parameter-Binding	TestCase003, 007, 008, 009, 012, 013, 018, 019, 020, 021, 022, 024, 025, 026 und 027
Parameter-Binding-Group	Alle Testmodelle, da die PBG als Container für Trigger, Guard und Action dient
Platform	Alle Testmodelle
StartState	Alle Testmodelle
Transition	Alle Testmodelle
Trigger	Alle Testmodelle
TypeNamePair	Alle Testmodelle, welche eine Variable besitzen
Variable	TestCases003, 007, 008, 009, 012, 013, 014, 017, 018, 019, 020, 021, 022, 023, 024, 025, 026 und 027
WidgetSocket	Alle Testmodelle, da in jedem Modell ein BaseWidgetSocket vorhanden ist. Über den BaseWidgetSocket hinaus besitzen TestCase002, 004, 005, 006, 007, 011, 015, 016, 017, 020, 021, und 002 weitere WidgetSockets

Fortsetzung auf der nächsten Seite

Tab. 20.1.: *Fortsetzung der* Abgleich des Metamodells mit den Testmodellen

Element des Metamodells	Abdeckende Testmodelle
WidgetSocketName	Alle Testmodelle, welche über den BaseWidgetSocket hinaus weitere WidgetSockets besitzen (siehe WidgetSocket)

21. Explorative Tests Editor

Nancy, Hannah

In diesem Kapitel werden die Tests für den Editor beschrieben. Zu diesen Tests gehörte die Durchführung der Use Cases, sowie das manuelle Testen von den verschiedenen Sichten des Editors. Der Editor wurde außerdem hinsichtlich seiner Usability und der konkreten Syntax der DORI-DSL getestet.

21.1. Use Cases

Nancy

An dieser Stelle werden die Use Cases für den Editor getestet. Vorbedingungen, Durchführung und erwartete Nachbedingungen sind jeweils im Kapitel 4 zu finden. Die Use Cases wurden alle zu einem Zeitpunkt getestet, an dem bereits die Grundfunktionalitäten des Editors gegeben waren. Zu diesen Funktionalitäten gehört, dass Elemente aus der DORI-DSL verwendet werden konnten, um eine Interaktion darzustellen. Alle Use Cases, die zu diesem Zeitpunkt noch nicht abgedeckt wurden, wurden zu einem späteren Zeitpunkt erneut getestet. Funktionalitäten, die bereits funktionierten, wurden nicht explizit nochmal getestet, sondern als gegeben angesehen.

21.1.1. Use Case „Modellierung der Gesamtinteraktion“

Nancy

Dieser Use Case ist die Basis der anderen Use Cases. Am 22.11.2017 wurde erstmals versucht, diesen Use Case komplett durchzuführen. Es stellte sich heraus, dass der Stand der Entwicklung noch nicht weit genug vorangeschritten war, um alle Elemente aus der DORI-DSL verwenden zu können. Dieser Test wurde am 18.02.2018 erneut getestet. Zu diesem Zeitpunkt stand fest, dass innerhalb der Projektgruppe nicht alle Elemente, die am Anfang festgelegt wurden, umgesetzt werden und somit dieser Use Case niemals komplett erfüllt

sein wird. Alle Elemente, die umgesetzt wurden, können auch in dem DORI-Editor verwendet werden.

21.1.2. Use Case „Modell bearbeiten“

Nancy

Der erste Schritt zur Durchführung dieses Use Cases besteht in der Erstellung eines neuen Modells. Eine Schritt-für-Schritt-Anleitung dafür ist in der Benutzerdokumentation des Editors zu finden. Nach dieser ist es möglich, ein neues Modell zu erstellen. Wie die Testmodelle zeigen, ist es ebenfalls möglich, einzelne Elemente der DSL mit einer konkreten Repräsentation in den Arbeitsbereich zu ziehen und eine Interaktion zu modellieren. Die Modellierung kann auf verschiedene Weisen beendet werden. Eine Möglichkeit ist das Speichern des Modells, welches im Test des nächsten Use Cases erläutert wird.

21.1.3. Use Case „Modell speichern“

Nancy

Der Benutzer kann sein Modell auf verschiedene Arten speichern. Der DORI-Editor bietet ein Disketten-Symbol an, womit gespeichert werden kann. Sind Änderungen an dem Modell gemacht worden und der Editor wird geschlossen ohne vorher explizit zu speichern, bekommt der Nutzer eine Abfrage ob gespeichert werden soll. Auch der Short-cut „Strg+s“ kann zum Speichern genutzt werden.

21.1.4. Use Case „Gesamtinteraktion bearbeiten“

Nancy

Nach dem Schließen des Editors kann jedes Modell wieder geöffnet und weiter bearbeitet werden. Bearbeiten bedeutet in diesem Fall, dass Elemente gelöscht, editiert oder hinzugefügt werden können.

21.1.5. Use Case „Inhalte in Katalogen organisieren“

Nancy

Dieser Use Case wurde am 22.11.2017 getestet. Zu diesem Zeitpunkt war die Entwicklung des Editors noch nicht so weit fortgeschritten, dass es Kataloge gab. Dieser Test wurde am

18.02.2018 wiederholt und zu diesem Zeitpunkt standen neue Sichten in dem Editor zur Verfügung. In diesen Sichten können konkrete und abstrakte Inhalte angelegt werden. Eine Wiederverwendung dieser Inhalte ist nicht modellübergreifend möglich und wird auch nicht mehr von dieser PG umgesetzt. Eine genauere Beschreibung dieser Kataloge befindet sich im Folgenden.

21.1.6. Use Case „Konsistenz von Modell prüfen“

Nancy

Dieser Test wurde am 22.11.2017 durchgeführt. Zu diesem Zeitpunkt war es möglich, mittels eines Rechtsklicks in dem Arbeitsbereich eine Validierung zu starten. Sollte es Fehler in dem Diagramm geben, so wird ein roter Kreis an die konkreten fehlerhaften Elemente im Arbeitsbereich gesetzt und es öffnet sich ein Feld, in dem die Fehlermeldungen angezeigt werden. Zu diesem Zeitpunkt gibt die Validierung noch kein Feedback aus, wenn sich keine Fehler im Modell befinden. Am 18.02.2018 wurde dieser Test wiederholt und auch zu diesem Zeitpunkt gab es kein Positives Feedback bei fehlerfreier Modellierung. Dieses Feature wird von der PG nicht mehr umgesetzt.

21.1.7. Use Case „Modell an Übersetzter übergeben“

Nancy

Dieser Use Case wurde am 22.11.2017 konkret getestet. Zu diesem Zeitpunkt gab es nur einen Übersetzer, den Java-basierten Interpreter für die Plattform JSF. Das Austauschformat zwischen dem Editor und dem Interpreter ist eine .xml Datei, die vom Editor generiert wird. In dieser Datei sind alle nötigen Informationen zum Interpretieren des Modells enthalten. Die Übergabe des Modells an den Interpreter geschieht durch eine Upload-Page. Am 18.02.2018 wurde dieser Test wiederholt. Zu diesem Zeitpunkt gab es einen neuen Interpreter für die Plattform Android. Dieser Interpreter arbeitet mit dem Kern des JSF-Interpreters und somit ist auch bei diesem das Austauschformat die .xml Datei. Ein automatisches Starten des Interpreters aus dem Editor heraus wird von dieser PG nicht umgesetzt.

21.2. Constraints

Nancy

Die Constraints sind Einschränkungen für das Metamodell. Da diese direkt in OCL formuliert wurden, waren diese auch direkt in dem Editor gegeben. Exemplarisch wurden einige

Constraints getestet, wie zum Beispiel, dass es keine eingehende Transition in einen Startzustand geben darf. Durch diese Automatisierung wird an dieser Stelle auf eine größere Beschreibung von dem Testen der Constraints verzichtet.

21.3. Tests für die Sichten

Nancy, Hannah

In dem Editor wurden unter anderem die Kataloge durch verschiedene baumartige Sichten umgesetzt. Neben den Katalogen stehen dem Benutzer aber auch andere Sichten, wie zum Beispiel die Data Type View zur Verfügung. Um die Funktionalität der verschiedenen Sichten zu garantieren, wurden diese manuell getestet. Eine Beschreibung dieser Tests befindet sich im Folgenden.

21.3.1. Data Type View

Nancy, Hannah

Die Data Type View ist eine spezielle Sicht des Editors, die es dem Nutzer ermöglichen soll, verschiedene Datentypen anzulegen, die dann im Modell verwendet werden können. Der Test soll diese Sicht im Folgenden beschreiben.

Vorbedingung Es gibt ein Modell mit Variablen, denen noch kein Datentyp zugewiesen wurde. Es wurde noch keine Data Type View angelegt.

Erwartete Nachbedingung Am Ende des Tests soll allen Variablen ein Datentyp zugeordnet worden sein.

Beschreibung der Durchführung Für den Test wurde der Dokumentation entsprechend eine neue Datensicht für das Modell angelegt. Über einen Button können sowohl primitive Datentypen, als auch Collections und Composite Datatypes angelegt werden. In dem Modell selber können die Properties der Variablen angezeigt werden. Unter diesen Punkt können den Variablen alle Datentypen zugewiesen werden, die vorher in der Data Type View angelegt wurden. Dieser Test erfüllte die erwartete Nachbedingung.

Properties-View

Die Properties View ist ein Bereich des Editors, in dem die Eigenschaften des aktuell fokussierten Elements eingesehen und manipuliert werden können. Dieser Test beschreibt, wie die individuellen Properties Views getestet wurden.

Vorbedingungen Es gibt ein Modell, welches Instanzen jeder Klasse des Metamodells enthält.

Erwartete Nachbedingungen Manipulierbare Eigenschaften sowie Eigenschaften, in die der Nutzer häufig Einsicht nehmen möchte, sind in der Properties View jeweils unter dem Tab „Properties“ zu finden. Eigenschaften, in die der Nutzer nur selten Einsicht nehmen möchte, sind dort unter dem Tab „Information“ zu finden.

Beschreibung der Durchführung Für den Test wurde jeweils die Properties View jedes Elementtyps in Augenschein genommen und die angezeigten Informationen auf ihre Manipulierbarkeit und Häufigkeit der Einsichtnahme hin bewertet. Anschließend wurde geprüft, ob die jeweilige Information unter dem diesbezüglich richtigen Tab aufzufinden ist. Der Test erfüllt die erwarteten Nachbedingungen.

21.3.2. Katalogsichten

Nancy

Die Katalogsichten sich spezielle Sichten im Editor, welche dazu dienen die Konkreten und Abstrakten Widget, sowie Funktionen gesammelt an einem Ort zu erstellen und diese aus der Modellierungsfläche heraus zu halten. Die Erstellung dieser Sichten stammen zum einen aus den Anforderungen und zum anderen aus den Anmerkungen zu der konkreten Syntax, die im Folgenden noch näher beschrieben werden.

Die Sichten teilen sich in fünf unterschiedliche Sichten auf. Für die Erstellung von konkreten Inhalten wurde jeweils eine Sicht erstellt, sowie für die abstrakten Inhalte. Die letzte Sicht ist für die Erstellung von Plattformen. Die einzelnen Sichten werden nachfolgend beschrieben mit einer Vorbedingung, einer Nachbedingung sowie einer Beschreibung der Durchführung von dem Test.

Abstract Widget with Mapping Catalog

Vorbedingung: Es gibt bereits ein Projekt und es ist somit eine Datei für die Modellierung einer Interaktion vorhanden. Außerdem benötigt die Interaktion eine abstrakte Widget Instance, auf die ein abstraktes Widget gemapped werden muss.

Erwartete Nachbedingung: Am Ende dieses Tests muss einer AWI (Abstract Widget Instance) ein abstraktes Widget zugeordnet worden sein, welches in der Katalogsicht erstellt wurde. Das abstrakte Widget darf keine grafische Repräsentation mehr in der Modellierungsfläche haben.

Durchführung: Bei der Erstellung eines neuen Projektes wird diese Katalogsicht in manchen Deployments automatisch angelegt. Sollte diese Sicht noch nicht vorhanden sein, kann diese über das DORI-Modell durch eine neue Repräsentation angelegt werden. In dieser Sicht gibt es am oberen Rand einen Button, der ein neues Widget erstellt. Alternativ kann durch einen Rechtsklick ein neues abstraktes Widget erstellt werden, sobald es schon ein abstraktes Widget gibt. Diesem kann ein Name gegeben werden. Im Properties-Fenster werden einem Nutzer verschiedene Möglichkeiten gegeben, Daten hinzuzufügen. Die Zuordnung zu einem AWI ist hier nicht gegeben. Diese Möglichkeit wird dem Nutzer allerdings in den Properties der AWI, durch ein Drop-Down-Menü gegeben. Diese Zuordnung ist auch lediglich in den Properties gegeben und es gibt keine grafische Repräsentation in der Modellierungsfläche des abstrakten Widgets. Somit ist die Nachbedingung erfüllt.

Abstract Function with Mapping Catalog

Vorbedingung: Es gibt bereits zwei AWIs die miteinander verbunden sind und an der Transition hängt eine Action, in der eine abstrakte Funktion gespeichert werden soll.

Erwartete Nachbedingung: Es gibt eine abstrakte Funktion, die in der Sicht erstellt wurde und die keine grafische Repräsentation in der Modellierungsfläche hat, sondern lediglich in der Action verwendet werden kann.

Durchführung: Genau wie bei der Katalogsicht der abstrakten Widgets wird diese Sicht bei manchen Deployments bereits automatisch erstellt. Sollte dies nicht der Fall sein, kann diese Sicht auch manuell erstellt werden durch einen Rechtsklick auf das DORI-Modell. In der Sicht gibt es auch wieder einen Button, wo eine neue Funktion angelegt werden kann. Auch hier ist eine andere Erstellungsweise durch einen Rechtsklick auf die freie Fläche möglich, sobald es bereits eine abstrakte Funktion gibt. Dieser kann ein Name gegeben werden sowie ein Rückgabe-Datentyp zugeordnet werden. Der abstrakten Funktion können ebenfalls durch die Properties verschiedene Parameter übergeben werden. Diese sind in diesem Fall Variablen, die in einer anderen Sicht erstellt werden können. Die Verwendung dieser Funktion geschieht in den Properties der Action. Über eine Liste kann die abstrakte Funktion gewählt werden, die der Nutzer haben möchte und diese wird dann nur in der Action angezeigt und hat keine eigenen grafische Repräsentation. Somit ist die Nachbedingung erfüllt.

Concrete Widget Catalog

Vorbedingung: Es gibt bereits eine Interaktion, die abstrakte Widgets benötigt. Zu den jeweiligen abstrakten Widgets gibt es noch keine konkreten Widgets.

Erwartete Nachbedingung: Es konnten konkrete Widgets in der Sicht erstellt werden. Außerdem kann einem konkreten Widget, in dieser Sicht, ein abstraktes Widget zugeordnet werden. Es befindet sich keine grafische Repräsentation des konkreten Widgets in der Modellierungsfläche.

Durchführung: Für die Vorbedingung dieses Tests konnte der Test von dem Abstract Widget with Mapping Catalog wiederverwendet werden. Der Concrete Widget Catalog war entweder angelegt oder er konnte wieder, genau wie die anderen Sichten manuell angelegt werden. Über einen Button können konkrete Widgets angelegt werden. Diesen kann ein Name gegeben werden und eine Plattform (sobald diese erstellt wurde) zugeordnet werden. Ebenfalls ist in den Properties ein Drop-Down-Menü gegeben, über das das abstrakte Widget gewählt werden kann. Nach dem Speichern dieser Erstellung und Auswahl kann wieder auf das Interaktionsdiagramm gegangen werden. Es befindet sich dort keine grafische Repräsentation des konkreten Widgets. Somit ist die Nachbedingung erfüllt.

Concrete Function Catalog

Vorbedingung: Es gibt bereits eine Interaktion, die abstrakte Funktionen benötigt. Es sind zu einer abstrakten Funktion noch keine konkrete Funktion vorhanden.

Erwartete Nachbedingung: Es können konkrete Funktionen in der Sicht erstellt werden und außerdem können diese dann den abstrakten Funktionen zugeordnet werden. Es befindet sich keine grafische Repräsentation der konkreten Funktion in der Modellierungsfläche.

Durchführung: Auch in dieser Durchführung wird die Vorbedingung durch den Test von dem Abstract Function with Mapping Catalog erfüllt. Der Katalog ist wie bei den anderen auch entweder automatisch vorhanden oder kann manuell erstellt werden. Über einen Button wird wieder eine neue konkrete Funktion angelegt. Dieser kann ein Name zugeordnet werden. Genau wie bei den konkreten Widgets kann hier die Plattform (falls bereits erstellt) und die abstrakte Funktion zugeordnet werden. Dies geschieht in den Properties. Eine grafische Repräsentation ist nicht in dem Interaktionsdiagramm vorzufinden und somit ist die Nachbedingung erfüllt.

Plattform Catalog

Vorbedingung: Es wurden noch keine Plattformen in dem Modell angelegt.

Erwartete Nachbedingung: In der Sicht konnten verschiedene Plattformen angelegt werden. Diese Plattformen können dann den konkreten Inhalten zugeordnet werden. Eine grafische Repräsentation, in der Modellierungsfläche, von den Plattformen, liegt nicht vor.

Durchführung: Auch diese Sicht ist entweder automatisch angelegt oder kann manuell, wie die anderen Sichten, angelegt werden. In dieser Sicht können auch wieder über einen Button verschiedene Plattformen angelegt werden. Die Verwendung dieser Plattformen wurde bereits in den Katalogen für die konkreten Inhalte getestet. Über eine Drop-Down-Menü konnte eine Plattform ausgewählt werden. Eine grafische Repräsentation dieser Plattformen ist nicht im Interaktionsmodell vorzufinden und somit ist auch diese Nachbedingung erfüllt.

Abstract Widget Properties Tree

Vorbedingung: Für diese Sicht gibt eine keine Vorbedingung.

Erwartete Nachbedingung: Die Sicht konnte erfolgreich geöffnet werden und abstrakte Widgets angelegt werden. Zu diesen abstrakten Widgets konnten TypeNamePairs und Eventnames, sowie WidgetSocketnames angelegt werden. Diese werden in der Sicht in einer Baumstruktur angeordnet.

Durchführung: Diese Sicht wird bei der Neuerstellung eines Projektes automatisch mit erstellt. Sollte dies nicht der Fall sein, kann diese auch individuell wie die Katalogsichten erstellt werden. Nach der Erstellung kann über ein kleines Symbol in der oberen Leiste ein Abstraktes Widget erstellt werden. Dieses erscheint dann in schriftlicher Form in der freien Fläche. Anschließend können unten in der Properties-View die TypeNamepairs, die Eventsnames und die WidgetSocketNames erstellt werden. Diese erscheinen dann eingerückt unter dem abstrakten Widget. Damit entsteht eine Baumstruktur. Zusätzlich kann durch das Klicken auf das jeweilige Element, diesem die Properties zugewiesen werden. Die Nachbedingung ist erfüllt.

Abstract Function Properties Tree

Vorbedingung: Für diese Sicht gibt eine keine Vorbedingung.

Erwartete Nachbedingung: Die Sicht konnte erfolgreich geöffnet werden und eine AbstractFunction angelegt werden. Zu dieser AbstractFunction konnte ein Datentyp der Rückgabe festgelegt sowie Variablen als Parameter zugeordnet werden. Letztere werden in der Sicht in einer Baumstruktur angeordnet.

Durchführung: Diese Sicht wird bei der Neuerstellung eines Projektes automatisch mit erstellt. Sollte dies nicht der Fall sein, kann diese auch individuell wie die Katalogsichten erstellt werden. Nach der Erstellung kann über das Symbol „CreateAbstractFunction“ in der oberen Leiste eine AbstractFunction erstellt werden. Dieses erscheint dann in schriftlicher Form in der freien Fläche. Anschließend kann unten in der Properties-View der Datentyp

des Rückgabewerts ausgewählt werden. Zudem können vorher definierte Variablen der AbstractFunction als Parameter zugeordnet werden. Diese erscheinen dann eingerückt unter der AbstractFunction. Damit entsteht eine Baumstruktur. Zusätzlich kann durch Klicken auf die jeweilige Variable dieser die Properties zugewiesen werden. Die Nachbedingung ist erfüllt.

Variable Viewer

Vorbedingung: Es gibt bereits ein Interaktionsmodell, welches eine AWI oder einen Decisionstate enthält. Außerdem gibt es bereits eine abstrakte Funktion oder ein Parameter-Binding.

Erwartete Nachbedingung: Es konnte erfolgreich die Sicht geöffnet werden und eine Variable erstellt werden. Dieser konnte ein Datentyp, sowie eine AWI oder ein Decisionstate zugeordnet werden. Außerdem kann in dieser Sicht festgelegt werden, in welcher Abstrakten Funktion und welchem Parameter-Binding die Variable zugeordnet ist. Bei der Zuordnung zu einer AWI erscheint eine grafische Repräsentation in dem Interaktionsmodell.

Durchführung: Diese Sicht wird bei der Neuerstellung eines Projektes automatisch erstellt. Sollte dies nicht der Fall sein, kann diese Sicht auch noch manuell, genau wie die anderen Sichten, erstellt werden. Über einen kleinen Button, in der oberen Leiste kann eine Variable angelegt werden. Durch verschiedene Drop-Down Menüs können die gewünschten Informationen zugeordnet werden. Dabei fällt auf, dass eine Variable einer AWI und einem Decisionstate zugeordnet werden kann. Erst durch die Validierung des Interaktionsmodells wird dieser Fehler sichtbar. Der neu erstellten Variable wurde eine AWI zum Test zugeordnet. Nach der Speicherung der Zuordnung, wurde auf das Interaktionsmodell gewechselt. Hier konnte eine grafische Repräsentation der Variablen an der AWI vorgefunden werden und damit ist die Nachbedingung erfüllt.

21.4. Konkrete Syntax der DSL

Nancy

Die konkrete Syntax der DSL hat sich im Laufe der PG verändert. Durch die Arbeit mit dem Editor zur Erstellung von Testfällen konnte die QA einige Tipps geben, was an der konkreten Syntax verbessert werden könnte. Mittels der von der Editorgruppe durchgeführten Interviews mit den Betreuern und der QA konnte die Editorgruppe Informationen bezüglich der gewünschten Anpassungen an der konkreten Syntax erlangen. Im Folgenden werden diese Syntax-Wünsche einmal aufgelistet sowie deren Umsetzung dargestellt.

Tab. 21.1.: Konkrete Syntax Editor

Anmerkungen konkrete Syntax	Umsetzung der Anmerkungen
Die Events und Variablen sollen direkt in der AWI stehen und die AWI soll ähnlich wie eine Klasse in einem Klassendiagramm aufgebaut sein.	Diese Anmerkung entstand durch den Platzmangel in der Modellierungsfläche. Diese Anmerkung wurde so nicht umgesetzt. Stattdessen wurde ein Filter eingeführt, der es dem Nutzer ermöglicht die Events und Variablen an den AWIs auszublenden, um mehr Platz in der Modellierungsfläche zu schaffen.
In den Choice- und Mergestates sollte jeweils nur noch der Anfangsbuchstabe drin stehen und die Farbe sollte weggelassen werden.	Die Farben der beiden States unterscheiden nun die States an sich. Es sind keine Kürzel mehr in den States enthalten. Diese Anmerkung ging von der QA aus, während die Betreuer eher keine Label in den States wünschten. Somit wurde diese Anmerkung dementsprechend anders umgesetzt.
Die Elemente der Transition sollten näher an der Transition platziert sein. Eventuell sollten diese wie bei einem Zustandsdiagramm direkt an der Transition stehen.	Diese Anmerkung wurde so nicht umgesetzt. Die einzelnen Elemente der Transition wurden nun in einem Kasten zusammengeführt, sodass alle Elemente an einer Stelle vorzufinden sind.
Die Parameter-Binding-Group nimmt zu viel Platz weg und muss deshalb kleiner ausfallen.	Diese Anmerkung wurde dadurch umgesetzt, dass ein Filter eingerichtet wurde, mit dem die Parameter-Bindung-Group ausgeblendet werden kann.

Fortsetzung auf der nächsten Seite

Tab. 21.1.: *Fortsetzung der* Usability Probleme

Anmerkungen konkrete Syntax	Umsetzung der Anmerkungen
In dem Parameter-Binding soll deutlicher hervor gehen, was davon die Source und was das Target ist.	Diese Anmerkung wurde so umgesetzt, dass nun durch ein Icon davor steht, was dieser Name für ein Element ist. Durch dieses Icon wird dennoch nicht ersichtlich was die Source ist und was das Target ist.
Die Abstract Widget Instance ID soll mit AWI ID abgekürzt werden.	Diese Anmerkung wurde umgesetzt. Außerdem befindet sich die ID nun in einem Informationsfenster und ist somit nicht mehr in den Properties.

21.5. Usability Editor

Nancy

Im Laufe der Erstellung von den Testfällen konnten durch die Rolle der QA verschiedene Mängel an der Usability aufgedeckt werden. Diese wurden im Confluence festgehalten und außerdem an die Editorgruppe weiter getragen. Im Folgenden werden diese Usability Probleme beschrieben und dargelegt ob und wie diese Probleme gelöst wurden.

Folgende Usability-Probleme sind aufgefallen:

Tab. 21.2.: Usability Editor

Problem	Beschreibung	Umsetzung
Beim Löschen von Knoten werden ebenfalls die Kanten gelöscht	Durch fehlerhaftes Modellieren von den Testfällen, konnte bereits eine komplette Parameter-Binding-Group erstellt worden sein und wenn aber der eine Knoten falsch war und dieser gelöscht wurde, wurde auch die Kante mit gelöscht und somit die kompletten Sachen, die an dieser Kante dran hingen.	Dieses Problem wurde nicht gelöst.
Das Verschieben von Transitionen zwischen verschiedenen Elementen ist nicht möglich.	Wenn eine Kante zum Beispiel zwischen zwei AWIs gezogen wurde und die eine Ziel-AWI doch falsch war, so musste die Transition gelöscht werden und die Transition konnte nicht umgezogen werden.	Das Verschieben einer Transition ist nun möglich. Das Ziel der Transition kann geändert werden, nicht aber die Herkunft. Um die Transition zu verschieben wird die Spitze genommen und zu dem anderen Ziel gezogen.
Die Properties-View ist sehr unübersichtlich, sowie die Reihenfolge der Informationen ist nicht eindeutig.	Beim Modellieren fiel auf, dass wichtige Informationen, die in dieser Sicht festgelegt wurden, sich sehr weit unten in der Sicht befanden und nicht so wichtige Dinge eher oben. Außerdem waren Informationen enthalten, die für den Nutzer für die Modellierung unwichtig waren.	Die Properties-View wurde angepasst. Eine Beschreibung dazu gibt es in dem dazugehörigen Kapitel.

Fortsetzung auf der nächsten Seite

Tab. 21.2.: *Fortsetzung der* Usability Probleme

Problem	Beschreibung	Umsetzung
Die Namen der Elemente in der Palette sind nicht intuitiv.	Ein Widget Socket hieß in der Palette WidgetSocketContainerCreation. Aus diesem Namen wurde bei der Modellierung der Testfälle nicht ersichtlich, dass dieses Element das Widget Socket darstellt.	Die Namen wurden angepasst und sind nun für den Nutzer intuitiv.
In dem Diagramm werden keine eindeutigen IDs vergeben.	Durch ein Testmodell, in dem zwei Elemente die gleiche ID aufwiesen, fiel auf, dass dies der Interpreter nicht interpretieren kann und dies somit in dem Editor angepasst werden muss.	Dieses Problem wurde behoben. Die IDs sind nun fortlaufend, sodass für jeden Element eine Nummer angelegt ist, die einen größer ist als die vorherige. Beim Löschen von einem Element, wird diese Nummer nicht ersetzt.
In den Elementen der Transition (Trigger, Action etc.) sollen keine IDs angezeigt werden. Durch die Gestaltung des Mappings fiel auf, dass der Benutzer zwar die ID von den AWIs und Sockets gebrauchen kann, aber nicht die von den Elementen der Transition. Da diese ID in dem Modellierungsbereich Platz weg nahm, sollte diese ausgeblendet werden.	Die IDs werden nicht mehr in diesen Elementen angezeigt.	

Fortsetzung auf der nächsten Seite

Tab. 21.2.: *Fortsetzung der* Usability Probleme

Problem	Beschreibung	Umsetzung
Die Palette ist eingeklappt, dies sollte geändert werden.	Das Ausklappen einer Palette ist ein unnötiger Schritt für den Benutzer. Außerdem ist auf dem ersten Blick nicht ersichtlich, dass diese Palette ausklappbar ist und somit ist es ein Usability-Problem.	Dieses Problem wurde nicht gelöst.
Es sollen mehrere .mm Dateien in einem Projekt angelegt werden können.	Durch die Erstellung von vielen Testfällen fiel auf, dass es viel einfacher wäre nicht für jeden Testfall ein neues Projekt zu erstellen, sondern einfach ein Projekt Testmodelle zu haben, in dem sich mehrere .mm Dateien befinden.	Dieses Problem kann durch technische Einschränkungen nicht gelöst werden.
Bei der Modellierung soll auch eine Meldung kommen, wenn alles gut ist.	Durch die Validierung der Testmodelle konnte festgestellt werden, ob ein Modell Fehler enthält. Wenn Fehler enthalten waren, zeigte dies die Validierung an. Wenn kein Fehler vorlag zeigte die Validierung nichts an und ein unwissender Benutzer könnte dies auch so deuten, dass die Validierung fehl geschlagen ist.	Dieses Problem wurde nicht gelöst.

Fortsetzung auf der nächsten Seite

Tab. 21.2.: *Fortsetzung der* Usability Probleme

Problem	Beschreibung	Umsetzung
Elemente der Transition sollen auch innerhalb des Base-WidgetSockets erstellt werden können.	Bei einer größeren Modellierung muss in der Modellierungsfläche viel gescrollt werden. Zum Erstellen von beispielsweise einem Trigger oder einer Action war der Benutzer gezwungen ganz zum Rand zu scrollen, um diese zu erstellen, da sich die Elemente einer Transition nur außerhalb des Base-WidgetSockets erstellen ließen.	Dieses Problem wurde gelöst, in dem die Erstellung nicht mehr durch den Nutzer geschieht. Durch das Ziehen einer Transition werden Trigger/Action und die anderen Elemente automatisch erstellt. Dennoch ist auch diese Erstellung außerhalb des Base-WidgetSockets und diese Elemente müssen selbstständig an die richtige Stelle gezogen werden.
Anpassung der Größe der Elemente an ihren Namen.	Da manche Namen von den Elementen sehr lang waren und das Element trotzdem sehr klein war, war es schwierig den Namen vollständig zu lesen und somit gestaltete sich die Modellierung schwieriger.	Dieses Problem wurde gelöst und die AWIs passen ihre Größe automatisch dem Namen an.
Platzsparendere konkrete Syntax der WidgetSockets.	Dadurch, dass die Sockets einen Namen hatten und einen großen Kasten als konkrete Syntax, nehmen diese sehr viel Platz in der Modellierungsfläche weg.	Dieses Problem wurde nicht gelöst.

22. Anwendungsbeispiel „Telefonbuch“

Nancy, Christopher, Hannah

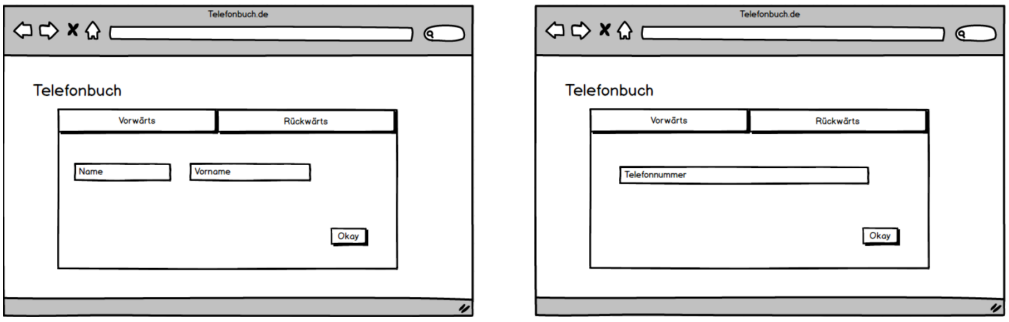
In dem folgenden Kapitel wird das Anwendungsbeispiel erläutert, welches im ersten technischen Durchstich für das Testen des Zusammenspiels aller Komponenten des Gesamtsystems umgesetzt wurde. Dieses Anwendungsbeispiel besteht aus einem Telefonbuch, welches verschiedene Elemente der DORI-DSL verwendet. Es wurde ausschließlich als Webanwendung umgesetzt, da JSF als erste Technologie festgelegt wurde.

22.1. Konzeption

Nancy, Christopher

Die Startseite des Telefonbuches besteht aus zwei ineinander geschachtelten Widgets. Das äußere Widget zeigt die Überschrift „Telefonbuch“. Die eigentliche Interaktion erfolgt im inneren Widget, hier kann initial eine Vorwärtssuche durchgeführt werden. Bei der Vorwärtssuche wird dem Benutzer die Möglichkeit gegeben, die Telefonnummer einer Person anhand ihres Namens zu suchen. Dabei ist der Name in Vor- und Nachname unterteilt. Mit „Okay“ wird die Suche angestoßen. Mindestens eines der beiden Eingabefelder muss für eine Suche mit einem Wert bestückt sein, andernfalls wird die Suche nicht angestoßen und stattdessen eine Fehlermeldung angezeigt. Die Abbildung 22.1a zeigt den Entwurf des Widgets mit aktivierter Vorwärtssuche. Statt einer Vorwärtssuche kann auch eine Rückwärtssuche durchgeführt werden. Bei der Rückwärtssuche wird zu einer Nummer der Eigentümer dieser Nummer herausgefunden. Sollte das Eingabefeld der Nummer leer sein, wird auch hier die Suche nicht angestoßen und eine Fehlermeldung angezeigt. Die Abbildung 22.1b zeigt ein beispielhaftes Aussehen des Widgets mit aktivierter Rückwärtssuche.

Sollten keine Ergebnisse für die Suche gefunden werden, wird der Nutzer auf eine Errorpage geleitet, welches diesen über fehlende Ergebnisse informiert. Nach erfolgreicher Suche werden alle Nummern zum eingegebenen Namen beziehungsweise Namen zur eingegebenen Nummer angezeigt, die von dem System gefunden worden sind. Der Benutzer kann mittels „Zurück“ sowohl die Errorpage als auch die Seite mit Ergebnissen wieder verlassen



(a) Aktivierte Vorwärtssuche (b) Aktivierte Rückwärtssuche

Abb. 22.1.: Entwurf der Startseite des Telefonbuchs

und wird zurück auf die Startseite geleitet. Die ausgewählte Suchart ist dabei die selbe wie beim Verlassen des Widgets.

22.2. Umsetzung

Hannah

Das modellierte Anwendungsbeispiel besteht aus sieben AWIs, der Interaktionsfluss ist in Abbildung 22.2 im Stand der konkreten Syntax zum Zeitpunkt der Modellierung sowie in Abbildung 22.3 in aktueller konkreter Syntax abgebildet. Innerhalb des WidgetSockets der *frame AWI* befinden sich die AWIs *forward AWI* und *backward AWI*, wobei initial *forward AWI* aktiv ist. Mittels der Events *AbstractWidgetInstance6.backward* und *AbstractWidgetInstance7.forward* kann jeweils zwischen den Suchen gewechselt werden, wobei eine eventuell vorhandene Fehlermeldung wieder zurückgesetzt wird. Die AWIs selbst sind zusätzlich jeweils mit dem Event *search* sowie mit einer Boolean-Variable *error* und den String-Variablen *surname* und *name* beziehungsweise *number* versehen.

Tritt das Event *search* auf, so werden letztgenannte Variablen einer Programmlogik *isEmpty* beziehungsweise *isEmpty2* übergeben, welche überprüft ob die Eingabe beziehungsweise mindestens eine der Eingaben nicht leer sind. Das Ergebnis wird jeweils im nachfolgenden ChoiceState in der Integer-Variable *stringEmpty* gespeichert. Dabei steht „0“ für „false“ und „1“ für „true“. Ist der Wert `1`, so wird der Nutzer zurück zur jeweiligen Such-AWI geleitet und deren Variable *error* auf true gesetzt. Bei `0` wird nun die Suche *forwardSearch* beziehungsweise *backwardSearch* mit den selben Parametern gestartet. Das Ergebnis der Suche ist eine Instanz des Datentyps „ResultsInfo“, welcher aus einer Liste mit Telefonbucheinträgen sowie einem Indikator über den Erfolg der Suche in Form eines Integers besteht.

Diese Instanz wird im nachfolgenden ChoiceState als Variable *resultsInfo* gespeichert. Je nach Wert des Indikators wird in die AWI *results AWI* oder *error AWI* gewechselt. Beide besitzen ein Event *back*, über das zurück in die*frame AWI* gewechselt werden kann. Die *results AWI* besitzt zusätzlich eine Variable vom Typ „ResultsInfo“, in die das Ergebnis der Suche gespeichert wird, damit die Liste mit den Telefonbucheinträgen angezeigt werden kann. Außerdem wurde hier die Variable von Typ eines Telefonbucheintrags *focus* sowie ein gleichnamiges Event modelliert, mit dessen Hilfe ursprünglich die Möglichkeit einer Detailanzeige für Listeneinträge umgesetzt werden sollte. Dies wurde jedoch aufgrund der Komplexität auf der GUI-Seite nicht umgesetzt, sodass bei Ausführung stets die *idle AWI* – eine leere GUI – und niemals die *details AWI* aktiv ist.

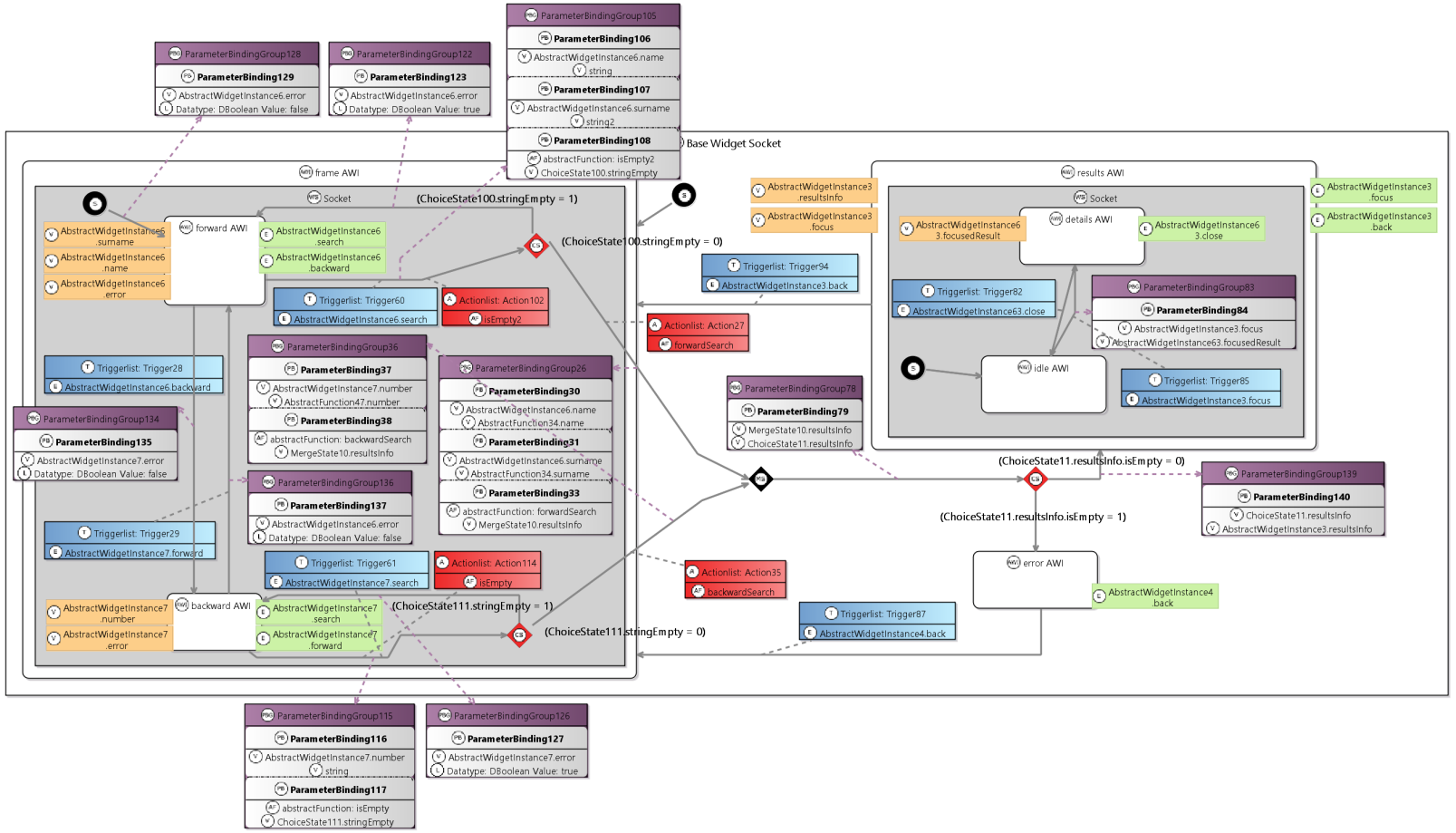


Abb. 22.2.: Interaktionfluss des Anwendungsbeispiels „Telefonbuch“ in konkreter Syntax zur Zeit der Modellierung

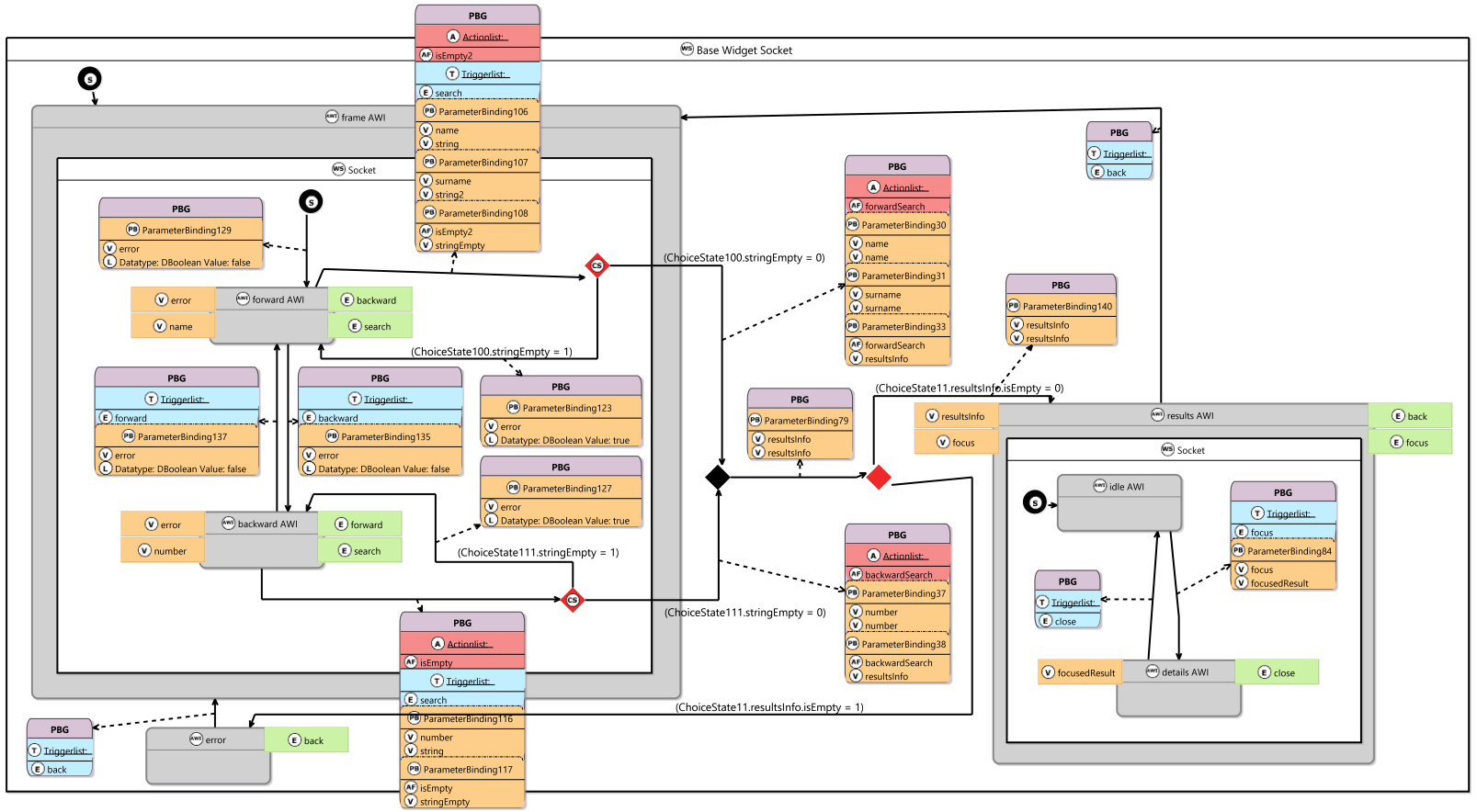


Abb. 22.3.: Interaktionfluss des Anwendungsbeispiels „Telefonbuch“ in aktueller konkreter Syntax

22.3. Evaluation

Hannah

Dieses Anwendungsbeispiel beinhaltet getriggerte und ungetriggerte Transitionen, Schachtelung von Widgets, bedingte Transitionsausführungen, Aufrufe von Programmlogik und Datenübergaben. Mit diesen Komponenten ist bereits ein nicht unerheblicher Teil der DORI-DSL abgedeckt. Das Anwendungsbeispiel „Telefonbuch“ konnte im ersten technischen Durchstich auf der Plattform JSF erfolgreich zum Laufen gebracht werden, womit das Zusammenspiel der Systemkomponenten grundsätzlich nachgewiesen wurde.

Die wichtigsten Verbesserungsmöglichkeiten, welche aus diesem ersten Anwendungsbeispiel gezogen wurden, betrafen die konkrete Syntax und den Entwicklungsprozess im Gesamten. Die konkrete Syntax wurde als zu wuchtig und bunt angesehen. Elemente, welche an eine Transition angefügt werden können, mussten jeweils separat platziert werden. Diese Punkte wurden als Grundlage für eine Überarbeitung der konkreten Syntax herangezogen, was auch an dem obig gezeigten Vorher-Nachher-Vergleich des Modells erkennbar ist. Zudem stellte sich heraus, das ein Debuggen des Modells und der zugehörigen GUIs allein durch den Interaktionsentwickler schwierig ist und vorzugsweise ein Entwickler des Interpreters anwesend sein sollte. Dem wurde im weiteren Verlauf der Projektgruppe durch Forcieren von Logging und Einführung einer Errorpage sowie der Möglichkeit einer Modell-Analyse vor Ausführung entgegengewirkt. Auch Verbesserungsmöglichkeiten bezüglich der Usability des Editors wurden aus diesem Anwendungsbeispiel gezogen. Schließlich stellte sich während der Modellierung heraus, dass die Grammatik der Guards keinen Vergleich von Booleansondern ausschließlich von Integer-Instanzen zuließ, dies konnte jedoch innerhalb dieses Modells leicht mit dem Workaround gelöst werden, statt „true“ und „false“ die ganzen Zahlen „0“ und „1“ zu verwenden. Darüber hinaus wurde die Guard-Grammatik darauffolgend überarbeitet und erweitert.

23. Anwendungsbeispiel „Konferenzverwaltung“

Nancy, Hannah

Wie bereits aus den Anforderungen hervor geht, ist ein großes Anwendungsbeispiel, welches alle Elemente der DORI-DSL abbildet und auf beiden Plattformen funktioniert, ein Deliverable der Projektgruppe. Die Projektgruppe hat sich in diesem Kontext für eine Konferenzverwaltung entschieden. Ziel der Konferenzverwaltung ist im Wesentlichen das Anlegen von Veranstaltungen und das Einladen von Freunden zu selbigen. Zudem können eigene Freunde, Veranstaltungen und Einladungen angezeigt und – im Falle der Einladungen – angenommen oder abgelehnt werden. Im Folgenden wird die Konzeption und Umsetzung dieses Anwendungsbeispiels beschrieben. Auf Basis dieses Anwendungsbeispiels wurden die DORI-Tools final evaluiert, diese Evaluation ist am Ende des Kapitels zu finden. Es ist zu beachten, dass das Modell des Anwendungsbeispiels in englischer Sprache formuliert wurde und daher dort die Rede von *Events* im Sinne von *Veranstaltungen* sein wird. Diese sind nicht mit den Events der DORI-DSL gleichzusetzen.

23.1. Konzeption

Nancy, Hannah

Die Konferenzverwaltung soll es einem Nutzer ermöglichen, sich bei dem System zu registrieren und anschließend einzuloggen. Dazu wird dem Nutzer zunächst eine Login-Page mit Eingabefeldern für Username und Password angezeigt, von der aus er auf die Registrierung wechseln kann. Für die Registrierung sind der Username sowie ein Passwort zwingend anzugeben. Da die Identifikation des Benutzers über den Usernamen erfolgt, dürfen zwei Nutzer nicht den gleichen Benutzernamen haben. Der Benutzer kann optional zusätzlich seinen vollen Namen, seine Adresse, seine Email sowie sein Geburtsdatum angeben. Beim Absenden der Registrierung wird der Nutzer automatisch eingelogt, falls die Registrierung erfolgreich war. Andernfalls wird er zurück zur Registrierung geleitet und bekommt eine Fehlermeldung angezeigt. Bei erfolgreicher Registrierung kann sich der Nutzer im Folgenden

direkt über die Login-Page einloggen. Werden hier falsche Daten eingegeben, so wird ebenfalls eine Fehlermeldung eingeblendet.

Nach dem Einloggen beziehungsweise Registrieren gelangt der Nutzer auf sein persönliches Dashboard und kann von da aus verschiedene Aktionen vornehmen und Informationen einsehen. Das Dashboard teilt sich in vier Abschnitte auf. In der ersten Anzeige werden dem Benutzer die Veranstaltungen angezeigt, an denen er teilnimmt. Dies impliziert sowohl die Veranstaltungen, zu denen er zugesagt hat, als auch seine selbst erstellten Veranstaltungen. Über einen Klick auf „Details“ werden alle Veranstaltungsdetails angezeigt. Im nächsten Abschnitt bekommt der Nutzer seine Freundesliste angezeigt, hier kann das Profil mit einem Klick auf „Show Profile“ angezeigt werden. Zudem können über „Add Friend“ Nutzer im System gesucht und Freundschaftsanfragen verschickt werden. Die dritte Anzeige enthält einen einfachen Kalender, auf dessen Basis Events an einem bestimmten Datum erstellt werden können. In der letzten Anzeige werden dem Benutzer seine Einladungen angezeigt. Initial werden die Einladungen zu Veranstaltungen eingeblendet, ein Wechsel zu den Freundschaftsanfragen ist möglich. Bei Annahme einer Einladung wird das Dashboard aktualisiert und der Freund beziehungsweise die Veranstaltung wird in der entsprechenden Anzeige eingeblendet. Mit einem Klick auf „Logout“ wird der Nutzer ausgeloggt und gelangt zurück auf die Login-Page.

Eine neue Veranstaltung kann der Nutzer – neben der Kalenderanzeige – direkt vom Dashboard aus erstellen. Es wird in ein neues Widget gewechselt, wo der Nutzer Titel, Ort, Datum, Uhrzeit und eine Beschreibung der Veranstaltung angeben kann. Außer dem Titel sind alle Eingaben optional. Da eine Veranstaltung über ihren Titel identifiziert wird, muss dieser eindeutig vergeben werden. Zu jeder Veranstaltung können Freunde eingeladen werden. Dazu werden in einer Liste alle Freunde angezeigt und können einer Liste mit einzuladenden Freunden hinzugefügt beziehungsweise aus dieser wieder entfernt werden. Mit einem Klick auf „Add“ wird die entsprechende Veranstaltung erstellt und die Einladungen versendet.

23.2. Umsetzung

Hannah

Das Modell des Anwendungsbeispiels besteht aus dreizehn AWIs, der Interaktionsfluss ist in Abbildung 23.1 dargestellt. Zur besseren Übersichtlichkeit sind hier ParameterBindingGroups, Variablen und Events ausgeblendet. Als äußere AWIs sind *Login*, *Registry*, *Events*, *AddEvent*, *ShowDetails*, *ShowProfile* und *AddFriend* vorhanden. Das AWI *Events* besitzt als Dashboard vier WidgetSockets, in denen sich die AWIs *AcceptedEvents*, *AcceptedFriends*, *CalenderSmall* und *Invitations* befinden. Das AWI *Invitations* beinhaltet wiederum einen WidgetSocket, welcher die AWIs *EventInvitations* und *FriendInvitations* enthält. So können auf

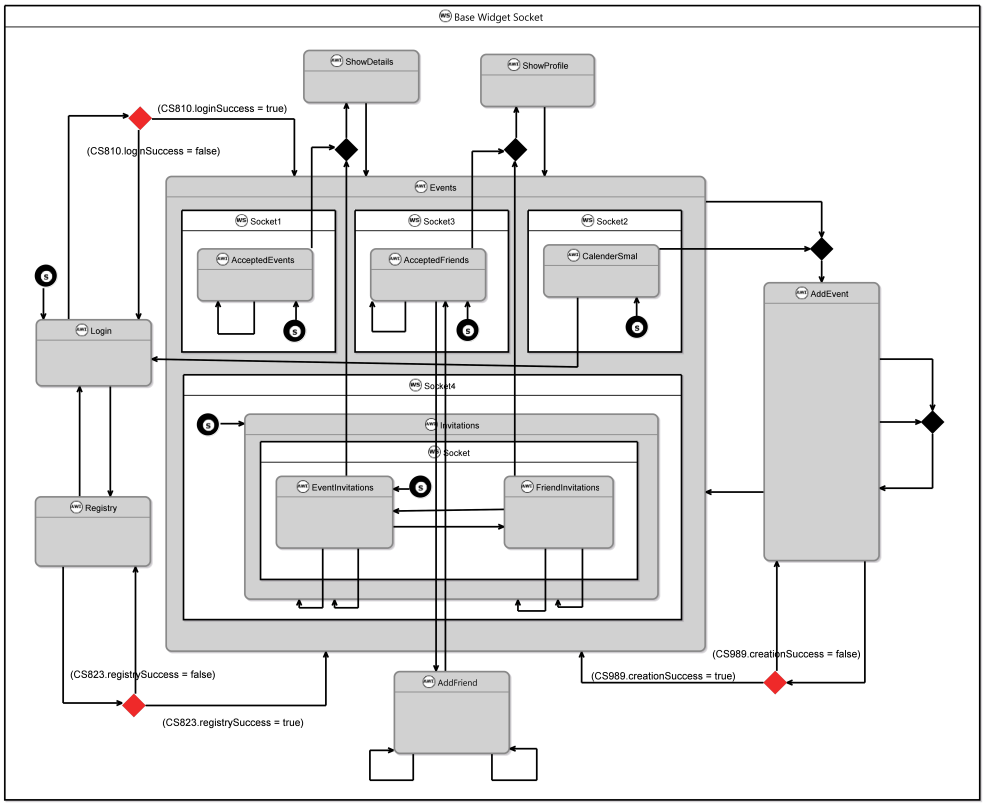


Abb. 23.1.: Interaktionsfluss des Anwendungsbeispiels „Konverenzverwaltung“

dem Dashboard Einladungen zu Events und Freundschaftsanfragen im Wechsel angezeigt werden.

Da die Login-AWI auf höchster Modellebene mit einem StartState versehen worden ist, ist sie das erste anzuzeigende Widget. Der detaillierte Informationsfluss im Bereich der Login-AWI ist in Abbildung 23.2 zu finden. Die Login-AWI ist mit den String-Variablen *loginUsername* und *loginPassword* ausgestattet, welche die entsprechenden Nutzereingaben speichern. Zudem ist die Boolean-Variable *loginError* vorhanden, welche initial auf *false* gesetzt wird. Als Events wurden *login* und *toRegistry* modelliert, letzteres ermöglicht einen Wechsel zur Registry-AWI. Ersteres übergibt die Nutzereingaben der Programmlogik *login*, welche bei erfolgreichem Login *true* und andernfalls *false* zurückgibt, was als Variable im nachfolgenden ChoiceState gespeichert wird. War der Vorgang erfolgreich, wird der Nutzer zur Events-AWI weitergeleitet, diese wird mit dem eingegebenen Usernamen initialisiert. Bei fehlgeschlagenem Login wird der Nutzer zurück zur Login-AWI geleitet und deren Variable *loginError* auf *true* gesetzt, sodass eine Fehlermeldung eingeblendet werden kann. Da der Interaktionsfluss im Bereich der Registry-AWI analog aufgebaut ist, wird auf diesen hier nicht im Detail eingegangen.

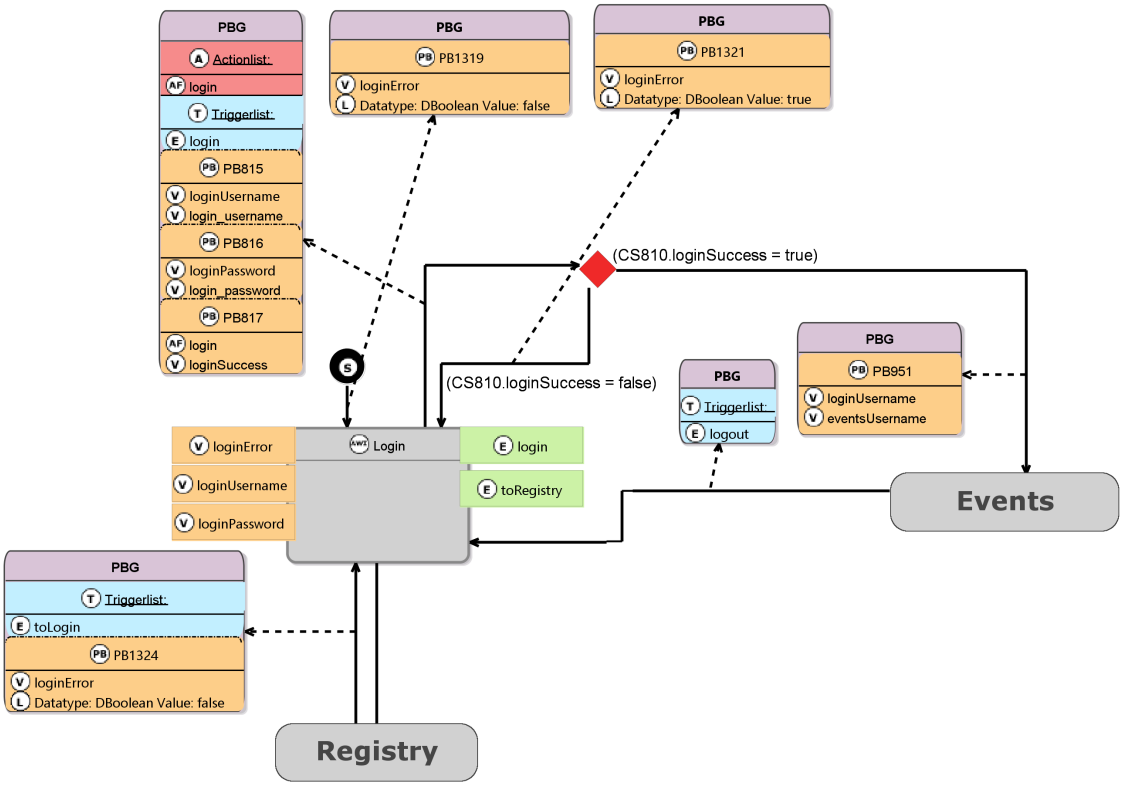


Abb. 23.2.: Interaktionsfluss im Bereich „Login“

Wir betrachten im Folgenden zunächst die Events-AWI im Allgemeinen – sprich das Dashboard – und anschließend dessen Bestandteile. Der Interaktionsfluss im Bereich des Dashboards ist in Abbildung 23.3 abgebildet. Die Events-AWI ist mit der String-Variable *eventsUsername* versehen, diese dient zur Identifikation des Nutzers bei Programmlogik-Aufrufen. Über das Event *createEvent* gelangt man in die AddEvent-AWI, wo Veranstaltungen angelegt werden können und welche weiter unten im Text separat beschrieben wird. In diese gelangt man ebenfalls über das Event *addEvent* in der CalenderSmall-AWI, hier wird außerdem das im Kalender gewählte Datum – modelliert als String-Variable – als Datum der Veranstaltung übergeben. Zusätzlich ist in der Events-AWI ein Event *updateProfile* hinterlegt, welchem jedoch im Modell keine weitere Funktionalität zukommt. Selbiges gilt für das Event *showEvents* in der CalenderSmall-AWI. An diesen Stellen kann das Modell des Anwendungsbeispiels erweitert werden. Das Event *logout* in der CalenderSmall-AWI führt zurück zur Login-AWI.

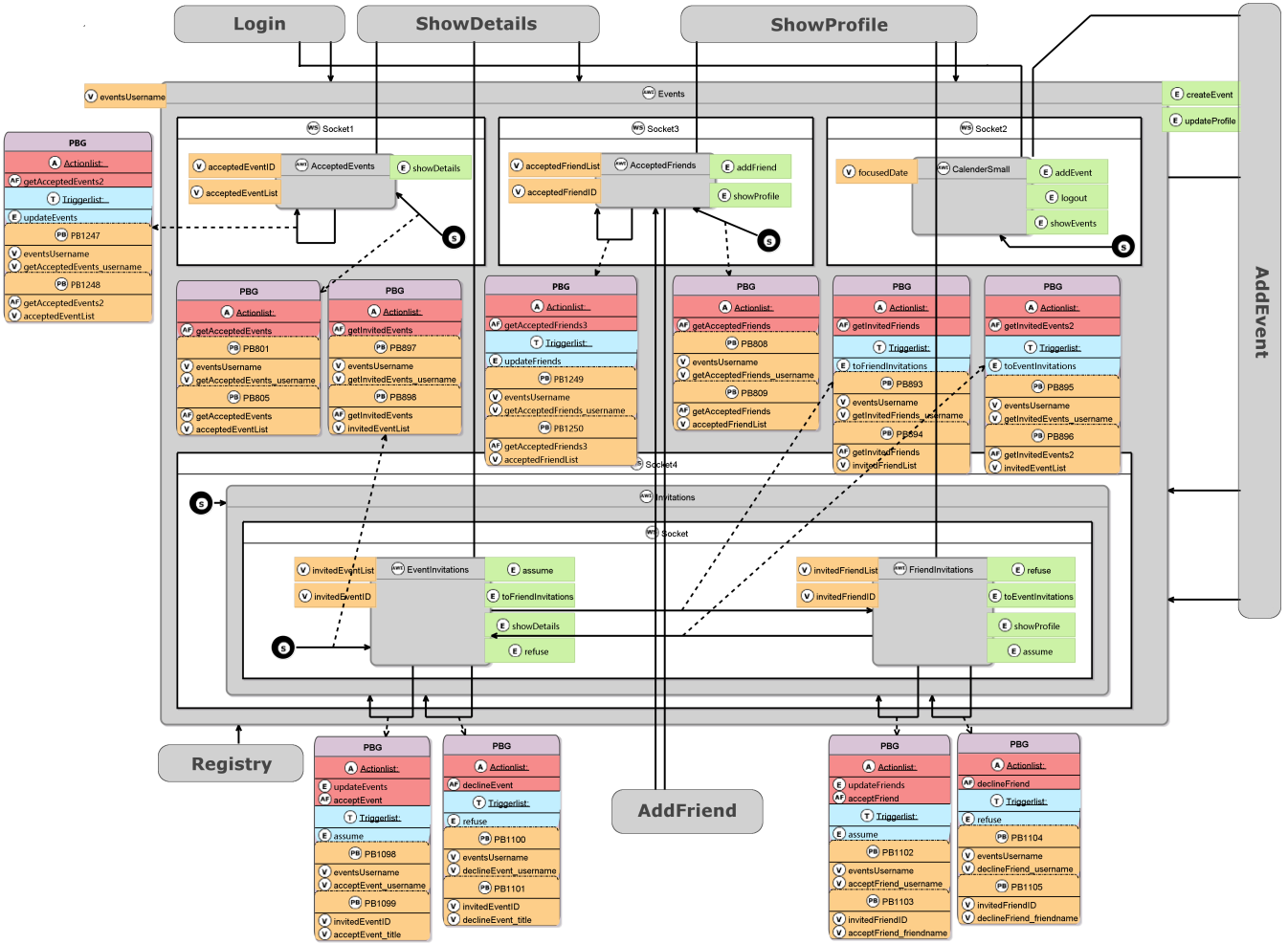


Abb. 23.3.: Interaktionfluss im Bereich „Dashboard“

Die AWIs *EventInvitations* und *FriendInvitations* im AWI *Invitations* sind strukturell identisch aufgebaut: Sie besitzen beide jeweils eine Collection des Datentyps *Event* beziehungsweise *Person* – die Einladungen zu Veranstaltungen *invitedEventList* beziehungsweise Freundschaftsanfragen *invitedFriendList*. Diese werden durch den Programmlogik-Aufruf *getInvitedEvents* beziehungsweise *getInvitedFriends* initialisiert, wobei jeweils der Username aus dem Dashboard als Parameter übergeben wird. In den String-Variablen *invitedEventID* beziehungsweise *invitedFriendID* kann der Name der aktuell fokussierten Einladung gespeichert werden, sodass beim Auftreten des Events *assume* oder *refuse* die passende Einladung angenommen beziehungsweise abgelehnt wird. Dies geschieht durch einen Aufruf der Programmlogik *accept/declineEvent* beziehungsweise *accept/declineFriend*, welcher ebenjene ID sowie der Nutzername aus dem Dashboard übergeben wird. Im gleichen Zuge wird ein Event *updateEvents* beziehungsweise *updateFriends* geworfen, sodass die bereits angenommenen Veranstaltungen oder Freunde aktualisiert werden. Über das Event *toFriendInvitations* beziehungsweise *toEventInvitations* kann zur jeweils anderen Ansicht gewechselt werden.

Die AWIs *AcceptedEvents* und *acceptedFriends* funktionieren sehr ähnlich dazu. Ihre Listen von Events beziehungsweise Freunden werden initial gefüllt und bei Annahme oder Neuerstellung von Elementen aktualisiert. In den AWIs *AcceptedEvents* und *EventInvitations* kann über das Event *showDetails* in die AWI *ShowDetails* gewechselt werden, wo die Details der Veranstaltung angezeigt werden. Gleiches ist für das Profil von Personen aus der AWI *AcceptedFriends* und *FriendInvitations* möglich. Über das Event *back* gelangt man jeweils wieder ins Dashboard zurück.

Der Interaktionsfluss im Bereich „AddEvent“ ist in Abbildung 23.4 dargestellt. Beim Betreten der AWI *AddEvent* wird eine Liste *addEventAllFriends* mit allen akzeptierten Freunden gefüllt. Dies sollte mit der zuvor bereits verwendeten AbstractFunction *getAcceptedFriends* geschehen. Da bei der Erstellung des Anwendungsbeispiels jedoch ein Multiplizitätsfehler im Metamodell festgestellt wurde, welcher aus Zeitmangel nicht mehr behoben werden konnte, konnte eine AbstractFunction nur höchstens einer Action zugeordnet werden. Daher mussten im Modell AbstractFunctions und ihre entsprechenden ConcreteFunctions pro Plattform dupliziert werden. Diese wurden jeweils mit dem selben Namen plus einer inkrementierten Nummer versehen. Generell besitzt die AddEvent-AWI neben einer Menge von String-Variablen zur Beschreibung des anzulegenden Events zwei Listen *AddEventAllFriends* sowie *AddEventSelectedFriends*. Letztgenannte Liste wird über den Programmlogikaufruf *getEmptyFriendList* mit einer leeren Liste initialisiert. Mit den Events *addFriend* und *removeFriend* können Elemente zwischen den Listen ausgetauscht werden. Der Zugriff auf die Listenelemente erfolgt analog zum obig beschriebenen Vorgehen. Über das Event *addEvent* wird die Veranstaltung erstellt, über *back* gelangt man ohne Erstellung zurück zum Dashboard.

Von der AWI *AcceptedFriends* aus kann man über das Event *addFriend* in die AWI *AddFriend* gelangen. Diese ist in Abbildung 23.5 veranschaulicht und besitzt eine String-Variable

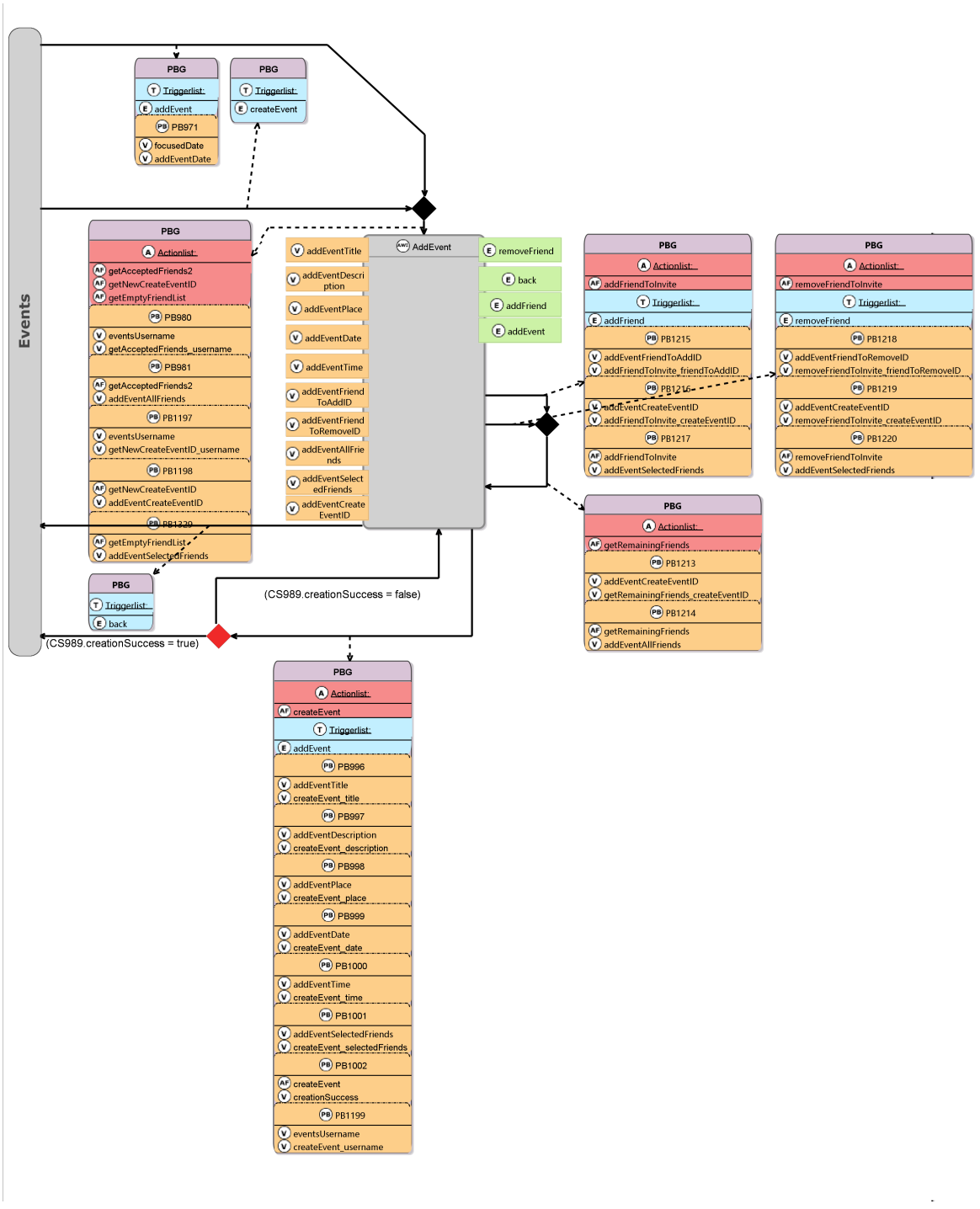


Abb. 23.4.: Interaktionsfluss im Bereich „AddEvent“

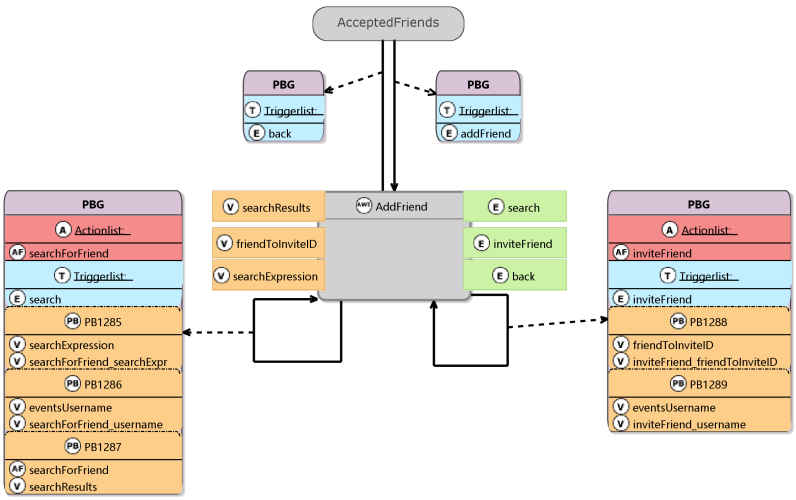


Abb. 23.5.: Interaktionsfluss im Bereich „AddFriend“

searchExpression, welche die Sucheingabe des Nutzers speichert. Bei Auftreten des Events *search* wird diese sowie der Username aus dem Dashboard an die Programmlogik *searchForFriend* übergeben und eine Liste von passenden Freundesvorschlägen in der Variable *searchResults* gespeichert. Einzelne Elemente dieser Liste können über *inviteFriend* eingeladen werden. Über *back* gelangt man zurück zum Dashboard.

23.3. Evaluation

Hannah

Die Umsetzung des Anwendungsbeispiels „Konferenzverwaltung“ hat gezeigt, dass das System DORI die Modellierung und Ausführung komplexer Interaktion ermöglicht. Das Anwendungsbeispiel läuft sowohl auf der Plattform JSF als auch Android. Da zudem alle Elemente der DORI-DSL im Anwendungsbeispiel verwendet wurden, validiert das Anwendungsbeispiel das Gesamtsystem der Projektgruppe. Das Ziel 4 der Projektgruppe, inklusive der darauf bezogenen Anforderungen, wurde demnach erfüllt. Im Folgenden werden die negativen Aspekte der finalen Evaluation beleuchtet.

Bei der Erstellung des finalen Anwendungsbeispiels im Editor stellte sich heraus, dass der Zeitaufwand für die Erstellung des Interaktionsmodells inklusive der benötigten GUIs und Programmlogik nicht gering ist. Dies liegt an dem hohen Synchronisationsaufwand zwischen den einzelnen Entwicklern. Es ist eine genaue Übereinstimmung etwa der Parameter von AbstractFunctions mit den Parametern der REST-Services, den Variablen beziehungsweise

Events im Modell und in den GUIs sowie den CompositeDataTypes im Modell und in der Programmlogik notwendig. Dieser Aufwand könnte durch ein integriertes System verringert werden, in dem GUIs und Programmlogik direkt auf Basis des Interaktionsmodells entwickelt würden. Dazu kommt die Länge eines einzelnen Entwicklungszykluses (Abspeichern des Modells, Neupacken des Archivs, manuelles Ausführen mit dem Interpreter) – hier ist jedoch zu beachten, dass dies mit wachsender Erfahrung des Nutzers ausgeglichen werden kann.

Generell kann man sagen, dass für eine abschließende Evaluation des Gesamtsystems ein längerer Zeitraum benötigt worden wäre, welcher der Projektgruppe nicht zur Verfügung stand. So hätte ein Repertoire an AbstractWidgets und AbstractFunctions sowie GUIs und Programmlogik aufgebaut werden können und der Nutzen der Wiederverwendbarkeit gegen die Investition in ein Interaktionsmodell abgewogen werden können.

Am Anwendungsbeispiel lässt sich gut erkennen, dass beim Entwickeln des Modells Namenskonventionen etwa für Funktionsparameter und Variablen an AWIs entwickelt wurden. So sind Funktionsparameter stets durch <Name der AbstractFunction>_<Name des Parameters> und Variablen an AWIs durch <Name der AWI><Name der Variable> benannt. Dies dient dazu, Elemente des Modells auseinanderzuhalten und ihrem Zweck zuzuordnen. In dieser Hinsicht kann der Editor noch verstärkt Unterstützung bieten, indem derartige Benennung entweder automatisch erzeugt wird oder die konkrete Syntax verstärkt die Verbindung der Modellelemente untereinander visualisiert.

Der Zugriff auf Elemente in Listen dynamischer Länge ist mit den Sprachmitteln der DORI-DSL nicht direkt möglich. Jedoch lässt sich leicht ein Workaround finden: Innerhalb der Modellierung geschieht dies über eine ID des fokussierten Elements als Variable in der AbstractWidgetInstance. Für genauere Einblicke in die Umsetzung auf Seiten der GUIs siehe Kapitel 6.2.2 für JSF beziehungsweise 6.3.2 für Android. Generell ist der Übergang vom Konzept des Anwendungsbeispiels zur Umsetzung zwar nicht immer offensichtlich, jedoch können mit den Sprachmitteln der DORI-DSL auch komplexe Funktionalitäten umgesetzt werden.

Die Arbeit an dem Anwendungsbeispiel deckte – wie bereits bei der Umsetzung erwähnt – den Fehler im Metamodell auf, dass eine AbstractFunction nur in höchstens einer Action verwendet werden kann. Mit obig beschriebenem Workaround war eine Modellierung doppelt vorkommender Funktionalität dennoch möglich, jedoch steht dieser Fehler im Gegensatz zur Wiederverwendbarkeit, welche im Zentrum der Ziele des Projektes steht.

Trotz dieser negativen Punkte hat die Projektgruppe ihr Ziel erfüllt, eine Möglichkeit zur plattformunabhängigen Modellierung von Gesamtinteraktionen zu schaffen. Damit wurde der Grundstein für eine Weiterentwicklung in diesem Bereich gelegt.

Teil V.

Ausblick und Fazit

24. Ausblick

Christopher

Die Projektgruppe sieht die folgenden Verbesserungsmöglichkeiten an den Ergebnissen. Diese werden an dieser Stelle nur in sehr kurzer Art und Weise beschrieben. Für eine ausführlichere Erklärung ist der Referenz zu folgen, ebenso wie eine Beschreibung wie eine solche Erweiterung forzunehmen wäre. Die Punkte sind thematisch angeordnet.

- Erweiterung des Metamodells um Kataloge um die Wiederverwendbarkeit zu erhöhen (Benutzerdokumentation, Kapitel 8.3.1).
- Erweiterung des Metamodells um „SubInteraktion“ um die Usability des Editors zu erhöhen (Benutzerdokumentation, Kapitel 8.3.1).
- Dynamische ForkStates/Dynamische Anzahl an WidgetSockets (Benutzerdokumentation, Kapitel 8.3.1), damit beispielsweise die Anzahl an ausgehenden Transitionen eines ForkStates erst zur Laufzeit bestimmt wird.
- Parametrisierte/Dynamische Events (Benutzerdokumentation, Kapitel 8.3.1), Events die um weitere Informationen angereichert werden können.
- Erweiterung des Metamodells um AbstractFunctionInstances (Benutzerdokumentation, Kapitel 8.3.1), analog zu AbstractWidgetInstances beziehungsweise um grundlegend die Mehrfachverwendung von AbstractFunctions erweitern.
- Erweiterung der Guards um AbstractFunctions beziehungsweise ReservedFunctions (Benutzerdokumentation, Kapitel 8.8).
- Löschen von Altlasten beim Re-Mapping im Editor (Benutzerdokumentation, Kapitel 5.4.4).
- Bessere Integration von XText in den Edtor, um z.B. Intellisense nutzen zu können.
- Erweiterung der Sicht für Daten des Interaktionsdiagrammes (Benuterdokumentation, Kapitel 5.4.1) um UML-Klassendiagramm.
- In ParameterBindingGroups füll- und erstellbare CompositeDataTypes.

- Export des Modells in andere Sprachen als Ecore.
- Erweiterung des Editors um Sichten mit denen Programmlogikentwickler und GUI-Entwickler am selben Modell mitarbeiten können. Zudem wäre eine Generierung von „Schablonen“ auf Basis der AbstractWidgets wünschenswert. Aufgrund der daraus resultierenden Komplexität ist eine Umsetzung mit Eclipse Sirius eher unwahrscheinlich, vielmehr wäre ein komplett selbst erstellter Editor von nöten.
- Erweiterung der Programmlogik um ein Transaktionskonzept, sodass Aufrufe von Programmlogik rückgängig gemacht werden können.
- Implementierung der nicht mehr umgesetzten PseudoStates des Metamodells (Benutzerdokumentation, Kapitel 8.3.1).
- Besserer Integration von REST.
- Erweiterung des Interpreters um eine zweite Art Programmlogik aufzurufen, abseits von REST.

Wegen des Fokus auf Änder- und Erweiterbarkeit innerhalb der Projektgruppe sollten die meisten der oben genannten Punkte mit verhältnismäßig wenig Aufwand umzusetzen sein. Für viele ist das Vorgehen hierzu detailliert an den angegebenen Stellen zu finden.

25. Fazit

Christopher

Abschließend lässt sich sagen, dass es der Projektgruppe möglich war alle gesteckten Ziele (vergleiche Kapitel 2) zu erreichen. So konnte eine eigene domänenspezifische Sprache, die DORI-DSL, erstellt werden, mit welcher sich Interaktion detailliert beschreiben und modellieren lässt. Zudem konnte für diese Sprache ein Editor erstellt werden. Da dieser mit Hilfe der Eclipse-Plattform gebaut wurde hat er eine intuitive Bedienbarkeit, welche sich mit den Erfahrungen ähnlicher Produkte deckt und daher wenig Einarbeitungszeit benötigt. Der Editor bietet hierbei die Möglichkeit Datentypen zu definieren, Interaktionsflüsse abzubilden und den Fluss von Daten zwischen den GUI-Elementen und Programmlogik zu definieren. Für diese Aufgaben stellt der Editor dem Benutzer eine Vielzahl an Werkzeugen zur Verwendung bereit.

Mit diesem Editor erstellte Modelle können ohne weitere Anpassungen von zwei verschiedenen Interpretern ausgeführt werden sobald plattformspezifische GUI-Masken hinterlegt und entsprechende Programmlogik implementiert wurde. So war es im Rahmen der Projektgruppe möglich, einen Interpreter für zwei verschiedene Plattformen, nämlich die Plattform Web sowie die Plattform Android zu schreiben. Wegen der guten Abkapselung von der Interpretation des Modells und der Anzeige plattformspezifischer GUI-Masken, benutzen beide dabei den selben Core-Interpreter. Eine einfache Erweiterbarkeit ist hier als großer Vorteil zu nennen, da Änderungen am Metamodell lediglich im Core-Interpreter berücksichtigt werden müssen, Web- und Android-Interpreter können ansonsten unverändert weiter betrieben werden.

Auch das letzte zu erfüllende Ziel konnte erreicht werden, die Evaluierung des erarbeiteten Ansatzes an einem Anwendungsbeispiel. An diesem, wie auch an anderen kleineren Beispielen, konnte der Ansatz – also eine von GUIs und Programmlogik unabhängige Beschreibung der Interaktion einer Anwendung – konsequent erprobt werden. Wurden in diesen zwar einige Schwächen festgestellt, so ist dies der geringen zur Verfügung stehenden Arbeitskraft geschuldet. Mit einer Aufstocken der solchen oder einer längeren Laufzeit des Projektes hätten viele der oben (Kapitel 24) beschriebenen Erweiterungsmöglichkeiten ebenfalls umgesetzt werden können. Mit diesen Erweiterungen wäre eine noch harmonischere Bedienbarkeit und erhöhte Funktionalität der Komponenten erzielt worden.

Die vorliegende Arbeit stellt somit eine gute Grundlage zur Schaffung der Rolle eines Interaktionsentwicklers dar, welche von den Rollen „Programmlogikentwickler“ und „GUI-Entwickler“ getrennt ist. Es konnte ebenfalls gezeigt werden, dass sich diese Rollen separat ausführen lassen, für die jeweiligen Arbeiten also kein Wissen aus den anderen Domänen von Nöten ist. Aus diesem Grund erachtet die Projektgruppe das umgesetzte Werk als gelungenes „Proof of Concept“ einer modellorientierten Beschreibung von Interaktion und Erstellung von Anwendungen.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[1] GÓMEZ, Prof. Dr.-Ing. Jorge M.: *NEMo - Mobilität Nachhaltige Erfüllung von Mobilitätsbedürfnissen im ländlichen Raum*. – URL https://www.nemo-mobilitaet.de/blog/de/. – Zugriffsdatum: 17.05.2017

[2] IFMLEDIT.ORG.: *IFMLEdit.org is a web based tool that will help you prototype and develop web and mobile apps!*. – URL http://info.ifmledit.org/. – Zugriffsdatum: 25.08.2017

[3] JELSCHEN, Jan: *SENSEI: Software evolution service integration*. 2014. – URL 10.1109/CSMR-WCRE.2014.6747220

[4] JÄNICKE, Julian: *Mobile App Design: Was ist UX, UI oder IxD?*. – URL https://de.yeeply.com/blog/mobile-app-design-was-ist-ux-ui-oder-ixd/. – Zugriffsdatum: 17.05.2017

[5] KLEUKER, Stephan: *Grundkurs Software-Engineering mit UML : Der pragmatische Weg zu erfolgreichen Softwareprojekten*. 3., korr. und erw. Aufl. 2013. Wiesbaden s.l : Springer Fachmedien Wiesbaden, 2013. – URL 10.1007/978-3-658-00642-6. – ISBN 9783658006426

[6] KOMPENDIUM, Elektronik: *FTP - File Transfer Protocol*. – URL https://www.elektronik-kompendium.de/sites/net/0902241.htm. – Zugriffsdatum: 17.10.2017

[7] KOUHEN, A. E. ; DUMOULIN, C. ; S. GERAD, P. B.: *Evaluation of Modeling Tools Adaptation*. 2012

[8] MÜLLER-PROVE, Matthias: *Interaktionsdesign in der Software- und Web-Entwicklung*. – URL https://mprove.de/script/09/ixdhh/index.html. – Zugriffsdatum: 17.05.2017

[9] OBJECT MANAGEMENT GROUP®, Inc.: *Documents Associated With Interaction Flow Modeling Language™ (IFML™) Version 1.0*. – URL http://www.omg.org/spec/IFML/1.0/. – Zugriffsdatum: 15.06.2017

[10] OBJECT MANAGEMENT GROUP®, Inc.: *What is UML*. – URL http://www.uml.org/what-is-uml.htm. – Zugriffsdatum: 20.08.2017

[11] RUPP, Chris ; QUEINS, Stefan: *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. Hanser, 2012. – ISBN 978-3-446-43057-0

[12] SCHLÖMER, Timo: *Modellgetriebene GUI-Erstellung für serviceorientierte Anwendungen*. 2017

[13] SUTHERLAND, Ken Schwaber J.: *The Scrum Guide - The Definitive Guide to Scrum: The Rules of the Game*. Internet. Juli 2013. – URL https://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-us.pdf

[14] UBUNTUUSERS: *Git*. – URL https://wiki.ubuntuusers.de/Git/. – Zugriffsdatum: 17.10.2017

[15] WEBRATIO: *STAND OUT IN THE DIGITAL BUSINESS AGE Be unique. Just like our mobile app and web dev platforms.*. – URL https://www.webratio.com/site/content/en/home. – Zugriffsdatum: 25.08.2017

Teil VI.

Appendix

A. Anforderungen

Hier folgt eine Liste der Anforderungen. Diese sind nur eine Zusammenfassung, die eigentliche Erhebung ist genauer im vorigen Kapitel aufgeschlüsselt.

A.1. Anforderungen für die DORI-DSL

Die folgenden Anforderungen ergeben sich für die DORI-DSL. Genauere Informationen finden sich im Kapitel 4.2.

Use Case „Definition und Verwendung einer abstrahierten GUI-Komponente“ (Tabelle 4.4)

- **DSL-AGK-1:** Die DORI-DSL muss ein Konstrukt bereitstellen, um eine GUI-Komponente zu abstrahieren. Dieses Konzept wird im Folgenden „Abstraktes Widget“ genannt.
- **DSL-AGK-2:** Einem abstrakten Widget muss eine Beschreibung zugeordnet werden können, aus der der Zweck der abstrahierten GUI-Komponente hervorgeht.
- **DSL-AGK-3:** GUI-Komponenten setzen sich gegebenenfalls aus weiteren GUI-Komponenten zusammen. Ein abstraktes Widget muss daher beschreiben können, wie viele weitere abstrakte Widgets es enthält.
- **DSL-AGK-4:** In einem abstrakten Widget müssen die zur abstrahierten GUI-Komponente gehörigen und für die Interaktionsbeschreibung relevanten Daten beschrieben werden können.
- **DSL-AGK-5:** In einem abstrakten Widget müssen die Events beschrieben werden können, welche die abstrahierte GUI-Komponente emmittieren kann und welche für die Interaktionsbeschreibung relevant sind.

- **DSL-AGK-6:** Die DORI-DSL muss ein Konstrukt bereitstellen, um eine abstrahierte GUI-Komponente im Kontext der Interaktionsbeschreibung einzusetzen. Dieses Konstrukt wird im Folgenden „Instanz eines abstrakten Widgets“ genannt.
- **DSL-AGK-7:** Die DORI-DSL muss darstellen können, welche Daten und Events die Instanz eines abstrakten Widgets verwalten können muss.
- **DSL-AGK-8:** Eine Instanz eines abstrakten Widgets muss weitere Instanzen von abstrakten Widgets beinhalten können.

Use Case „Definition von abstrahierter Programmlogik“ (Tabelle 4.5)

- **DSL-APL-1:** Die DORI-DSL muss ein Konstrukt bereitstellen, um Programmlogik zu abstrahieren. Dieses Konzept wird im Folgenden „Abstrakte Funktion“ genannt.
- **DSL-APL-2:** Einer abstrakten Funktion muss eine Beschreibung zugeordnet werden können, aus der die Funktionalität der abstrahierten Programmlogik hervorgeht.
- **DSL-APL-3:** Die DORI-DSL muss ein Konstrukt bereitstellen, um abstrahierte Programmlogik im Kontext der Interaktionsbeschreibung einzusetzen.

Use Case „Modellierung des Wechsels zwischen GUI-Komponenten“ (Tabelle 4.6)

- **DSL-WGK-1:** Die DORI-DSL muss gerichtete Transitionen enthalten.
- **DSL-WGK-2:** Die DORI-DSL muss ein Konstrukt bereitstellen, um das Nehmen von Zustandsübergängen zu beschränken.
- **DSL-WGK-3:** Die DORI-DSL muss ein Konstrukt bereitstellen, um während eines Zustandsübergangs Programmlogik auszuführen.
- **DSL-WGK-4:** Die DORI-DSL muss ein Konzept bereitstellen, um das Eintreten von Ereignissen zu beschreiben. Dieses Konzept wird im Folgenden „Events“ genannt.

Use Case „Modellierung von Datenflüssen“ (Tabelle 4.7)

- **DSL-DFL-1:** Die DORI-DSL muss ein Konstrukt bereitstellen, um Datenübergaben zwischen Instanzen abstrakter Widgets und abstrakten Funktionen zu beschreiben.

Use Case „Verknüpfung von Abstraktion mit Implementierung“ (Tabelle 4.8)

- **DSL-VAI-1:** Die DORI-DLS muss einer abstrahierten GUI-Komponente (Abstraktes Widget) eine konkrete Implementierung dieser GUI-Komponente zuordnen können.
- **DSL-VAI-2:** Die DORI-DLS muss einem abstrahierten Programmlogikaufruf (Abstrakte Funktion) eine konkrete Implementierung dieses Programmlogikaufrufs zuordnen können.

Sonstige Anforderungen an die DSL

- **DSL-SON-1:** Die DORI-DSL muss plattformunabhängige Datentypen (im Folgenden „primitive Datentypen“) bereitstellen.
- **DSL-SON-2:** Die DORI-DSL muss die Definition von zusammengesetzten Datentypen ermöglichen.
- **DSL-SON-3:** Syntax und Semantik der DORI-DSL sind in der Projektdokumentation festgehalten.

A.2. Anforderungen für den DORI-Editor

Use Case „Modellierung der Gesamtinteraktion“ (Tabelle 4.10)

- **E-MDG-1:** Der DORI-Editor muss die Erstellung eines neuen Modells ermöglichen.
- **E-MDG-2:** Der DORI-Editor muss bereits erstellte Modelle öffnen können.
  Beschreibung: Da Modelle dafür gespeichert werden müssen besteht eine Verknüpfung zum Use Case E-MSP (siehe Tabelle 4.11
- **E-MDG-3:** Der DORI-Editor muss die Bearbeitung eines geöffneten Modells ermöglichen. Dies bedeutet der Nutzer fügt Elemente zum Modell hinzu, entfernt Elemente oder definiert Beziehungen zwischen Elementen, gemäß der DORI-DSL.

Use Case „Modell speichern“ (Tabelle 4.11)

- **E-MSP-1:** Der DORI-Editor muss ein geöffnetes Modell persistent speichern können.
  Beschreibung: Das Modell muss bei der Bearbeitung gespeichert werden können und auch beim Schließen.
- **E-MSP-2:** Der DORI-Editor muss beim Schließen dem Nutzer die Möglichkeit zum Speichern bieten.
- **E-MSP-3:** Der DORI-Editor muss das Schließen ohne Speichern zulassen.
- **E-MSP-4:** Der DORI-Editor muss die persistente Speicherung des Modells in einem versionierbaren Format ermöglichen.
  Quelle: interne Gruppengespräche
- **E-MSP-5:** Der DORI-Editor muss den Speicherstand des aktuell geöffneten Modells überwachen.
- **E-MSP-6:** Der DORI-Editor muss den Nutzer auf ungespeicherte Änderungen hinweisen.

Use Case „Gesamtinteraktion bearbeiten“ (Tabelle 4.12)

- **E-GIB-1:** Der DORI-Editor muss die abstrakte Syntax der DORI-DSL einhalten.
- **E-GIB-2:** Der DORI-Editor muss die konkrete Syntax der DORI-DSL darstellen können.
  Beschreibung: Die grafische Darstellung des Modells auf Basis der konkreten Sytanx kann beispielsweise in einem Arbeitsbereich erfolgen. Der Zugriff auf die Elemente der DORI-DSL kann beispielsweise über eine Toolbox erfolgen. Die konkrete Syntax wird von der DORI-DSL bereitgestellt und in den Editor eingebunden.
- **E-GIB-3:** Der DORI-Editor muss die Bearbeitung eines Modelles auf Basis der DORI-DSL ermöglichen.
  Beschreibung: Der DORI-Editor muss die Bearbeitung von Modellen auf Grundlage aller bereitgestellten Sprachmittel der DORI-DSL ermöglichen.

Use Case „Konsistenz des Modells prüfen“ (Tabelle 4.13)

- **E-KMP-1:** Der DORI-Editor muss dem Nutzer eine Funktionalität zum Prüfen der Konsistenz eines Modells bereitstellen.
- **E-KMP-2:** Die Konsistenzprüfung muss auf Basis der DORI-DSL das Modell prüfen.
- **E-KMP-3:** Die Konsistenzprüfung muss dem Nutzer eine Übersicht über die Fehler des geprüften Modells bereitstellen können.
- **E-KMP-4:** Die Konsistenzprüfung muss dem Nutzer eine Übersicht über die Fehlerfreiheit des geprüften Modells bereitstellen können.

Use Case „Modell an Übersetzer übergeben“ (Tabelle 4.14)

- **E-MUU-1:** Der DORI-Editor muss die modellierte Gesamtinteraktion an den Übersetzer übergeben können.
- **E-MUU-2:** Der DORI-Editor und der Übersetzer müssen eine gemeinsame Schnittstelle zum Übergeben des Modells haben.

Sonstige Anforderung an den DORI-Editor

- **E-SON-1:** Der DORI-Editor muss die gängigen Designrichtlinien einhalten.
  Beschreibung: Die Festlegung der verwendeten Richtlinien erfolgt später im Projektverlauf und hängt von der Festlegung des Editor-Tools ab.
  Quelle: interne Gruppengespräche
- **E-SON-2:** Der DORI-Editor muss die gängigen Richtlinien der Software-Ergonomie einhalten.
  Quelle: interne Gruppengespräche
- **E-SON-3:** Der DORI-Editor muss eine Funktionalität zum Erstellen eines Berichtes bieten, welcher die Beschreibungen der abstrakten Inhalte beinhaltet.
  Beschreibung: Der Bericht dient dazu dem GUI- und Programmlogikentwickler eine Übersicht über zu implementierende Widgets und Funktionen zu bieten.
  Quelle: interne Gruppengespräche

- **E-SON-4:** Zum DORI-Editor muss es ein Installations- und Benutzerhandbuch geben.
  Beschreibung: Das Installationshandbuch muss es dem Nutzer ermöglichen die Software zu installieren. Das Benutzerhandbuch muss den Umgang mit dem DORI-Editor für den Nutzer erläutern.
  Quelle: Anforderung der Betreuer

A.3. Anforderungen für den Übersetzer

Use Case „Interaktionsmodell in Endanwendung übersetzen“ (Tabelle 4.16)

- **U-IEU-1:** Ein Interaktionsmodell muss in eine Endanwendung auf mehreren Plattformen übersetzbar sein.
- **U-IEU-2:** Der Übersetzer muss die Auswahl der Zielplattform ermöglichen.
  Beschreibung: Es ist denkbar, dass für jede Plattform ein neuer Übersetzer geschrieben wird. Diese Übersetzer können entweder in einem alleinstehenden Programm oder durch den DORI-Editor als Übersetzer-Sammlung gekapselt werden.
  Quelle: interne Gruppengespräche
- **U-IEU-3:** Der Übersetzer muss die Endanwendung in Konformität zur DORI-DSL übersetzen.
- **U-IEU-4:** Der Interpreter muss die Endanwendung auf der Zielplattform ausführen.
- **U-IEU-5:** Der Generator muss die Endanwendung für die Zielplattform übersetzen.
- **U-IEU-6:** Der Übersetzvorgang muss dokumentiert sein.
  Beschreibung: Bevor das Übersetzen für eine Zielplattform durchgeführt werden kann müssen eventuell zusätzliche Dateien oder Programme installiert werden. Die Installation der zusätzlichen Software muss daher in einem Dokument beschrieben werden.
- **U-IEU-7:** Der Übersetzer muss dokumentiert werden.

A.4. Anforderungen an das Gesamtsystem

Use Case „Gesamtsystem erweitern“ (Tabelle 4.17)

- **GS-ERW-1:** Das Gesamtsystem muss erweiterbar sein.
  Beschreibung: Der Entwicklungsprozess in der Projektgruppe muss stets ein Augenmerk auf die Erweiterbarkeit des Systems legen. Das bedeutet, dass zum Beispiel Software modular aufgebaut sein muss und stets ausreichend Dokumentation vorliegt. In dieser Dokumentation muss ein besonderer Fokus auf Beschreibungen zu Auswirkungen von Änderungen auf andere Systembestandteile gelegt werden.
  Quelle: Interview mit den Betreuern
- **GS-ERW-2:** Es muss eine Architekturdokumentation mitgeliefert werden.

A.5. Anforderung für die Validierung

- **V-GSV-1:** Das Gesamtsystem muss anhand eines Anwendungsbeispiels validiert werden.
- **V-GSV-2:** Die Gesamtinteraktion des Anwendungsbeispiels muss mit dem DORI-Editor modelliert werden können.
- **V-GSV-3:** Das Anwendungsbeispiel muss mit dem Übersetzer in mindestens zwei Endanwendungen auf unterschiedlichen Plattformen übersetzt werden
- **V-GSV-4:** Das Anwendungsbeispiel muss hinreichend komplex sein.
  Beschreibung: Wenn eine gewisse Komplexität nicht gegeben ist, dann findet keine Überprüfung statt, ob das Gesamtsystem für die Software-Entwicklung geeignet ist. Beispiele für komplexe Endanwendungskomponenten sind: Client-Server Kommunikation, Navigationsleisten, geschachtelte Widgets, dynamisch reagierende Eingabefelder. Der Endnutzer muss multiple Interaktionen vornehmen können.
- **V-GSV-5:** Das Interaktionsmodell des Anwendungsbeispiels muss alle Elemente der DORI-DSL verwenden.
  Beschreibung: Das Interaktionsmodell muss die DORI-DSL ausreizen, um eine Validierung zu ermöglichen.