Endbericht

Modellbasierte Entwicklung von Alexa-Skills

Projektgruppe 621

Jim-Marvin Bergmann
Tim Berthold
Robin Czarnetzki
Annika Fuhge
Nicole Funk
Christoph Meyer
Benedict Schubert
Fabian Storek
Jonathan Thöne

November 2019

Veranstalter:
Prof. Dr. Bernhard Steffen
Dr. Stefan Naujokat
M.Sc. Philip Zweihoff

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Programmiersysteme (LS5)
http://ls5-www.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Projektgruppe (PG) *Modellbasierte Entwicklung von Alexa-Skills* wird im Wintersemester 2018/19 und dem folgenden Sommersemester an der Technischen Universität Dortmund durchgeführt. Sie läuft unter der Leitung von Prof. Dr. Bernhard Steffen, Dr. Stefan Naujokat und M.Sc. Philip Zweihoff des Informatik Lehrstuhls für Programmiersysteme (LS5).

Der LS5 beschäftigt sich mit der Entwicklung und Erforschung von computerbasierten Werkzeugen zur Unterstützung der Modellierung, des Designs, der Implementierung und des Betriebs komplexer Anwendungssysteme. Während der PG verwenden die Teilnehmer die vom Lehrstuhl bereitgestellten Entwicklungsumgebungen (IDEs) Cinco und Dime. Sie werden zur Entwicklung eines Tools verwendet, das mittels verschiedener Modellierungssprachen die Modellierung von Sprachabläufen ermöglicht. Nach einem vorgegebenen Schema können diese Abläufe für die Implementierung von Anwendungen für Amazon's Echo Geräte genutzt werden [40].

1.1 Motivation und Hintergrund

Im *Internet of Things* sollen diverse Soft- und Hardwarekomponenten so miteinander verbunden werden, dass sie ein interaktives und intelligentes Gesamtsystem bilden. Zur Entwicklung neuer Systeme werden regelmäßig vereinfachende und veranschaulichende Modelle erstellt, um sich auf *wesentliche Aspekte der Systemmodellierung* zu fokussieren. In der Informatik werden hierzu vor allem Techniken aus der textuellen und graphischen Modellierung verwendet und neue Modellierungssprachen entwickelt. Im Fokus stehen vor allem Modellierungssprachen mit hohem Wiederverwendungsgrad, sogenannte Universalsprachen (*General Purpose Language*, GPL), und domänenspezifische Sprachen (*domain-specific lan-*

guage, DSL), letztere werden ausschließlich zur Modellierung spezifischer Domänen entwickelt.

Das Unternehmen *Amazon* ist mit ihrem Sprachassistenten *Alexa* und dem zugehörigen Skillsystem stark in den Bereichen *Smart Home*, durch die Vernetzung ihrer *Amazon Echo* Geräte, im *Internet Of Things* vertreten. Zur Erstellung eigener Anwendungen für Alexa gewährt Amazon externen Entwicklern Zugriff auf unterschiedliche *Software Development Kits* (SDKs), die für verschiedene Programmiersprachen bereitgestellt werden. Hierzu gehören die *Alexa Voice Service SDK* (AVS SDK), mit der Alexa in beliebige Geräte integriert werden kann, und die *Alexa Skills Kit SDK* (ASK SDK), mit der neue Skills zum Beispiel über die Sprache JAVA entwickelt werden können [6, 5].

Im Bereich der effizienten Softwareentwicklung durch graphische Modellierung werden im Rahmen dieser Projektgruppe die Modellierungssoftware CINCO und die darauf basierende Software DIME verwendet. CINCO bietet eine auf dem ECLIPSE Modeling Framework und zugehöriger Schnittstellen basierende Lösung zur Entwicklung von graphischen Modellierungssprachen, die zur Programmierung und Programmgenerierung verwendet werden können. Die Nutzer-Schnittstellen in CINCO werden in den Sprachen XTEXT und XTEND sowie CINCO-internen Modellierungssprachen entwickelt. DIME ist eine auf CINCO basierende Lösung zur modellbasierten graphischen Entwicklung von Webseiten samt Datenbankanbindung. Sowohl CINCO als auch DIME basieren auf JAVA und verwenden MAVEN als *Build Management Tool*.

Die bisher genannten Softwares bieten ausreichend Schnittstellen, um miteinander verbunden zu werden und somit ein interaktives Gesamtsystem zur Modellierung, zur Entwicklung, zum Deployment und zur Anwendung von Alexa-basierten Skills innerhalb von *Amazon Echo* Geräten zu erstellen. Mit DIME werden Web-Apps entwickelt, die zusätzlich zur graphischen Nutzeroberfläche eine Datenstruktur bereitstellen. Zusammen mit der Meta-Software CINCO kann ein Tool mit einer Sammlung von DSLs zur Skill-Modellierung entwickelt werden, wobei die Modellierung des Skills und der Web-App primär graphisch erfolgen können. Mit diesem Tool modellierte und entwickelte Skills und die DIME-Webschnittstelle können die gleiche Datenbasis teilen. So könnten zum Beispiel auf einer DIME Web-App Daten verwaltet werden, welche im Kontext eines Ablaufs eines Skills benötigt werden. In einem zugehörigem Skill könnten Dime-Daten dann zum Abgleich von Nutzerinput verwendet werden.

1.2 Anforderungen

Es werden zwei Hauptziele festgehalten, die im Verlauf der PG umgesetzt werden sollen. Zunächst sollen domänenspezifische Modellierungssprachen zur Entwicklung von Alexa-Skills mit der Software Cinco entwickelt werden. Im Zuge dessen wird ein Cinco-Produkt, das Alexa Skill Development Tool (ASDT), entwickelt. Im Anschluss muss das ASDT so erweitert werden, dass eine entwickelte Dime-App parallel zur Webschnittstelle zusätzlich mit Alexa über eigene Funktionen bedient oder um neue erweitert werden kann. Die Ziel-Architektur, die die Zusammenarbeit der Technologien, die Interaktionsmöglichkeiten des Anwenders sowie die Steuerungsmöglichkeiten des Entwicklers zeigt, ist in Abbildung 1.1 zu sehen.

Entwicklung des Alexa Skill Development Tools

Als erster Meilenstein soll eine graphische DSL zur modellbasierten Entwicklung von Alexa-Skills (Skill-DSL) entwickelt werden. Hierzu muss zunächst ein Konzept zur Modellierung von Skillabläufen entworfen werden, das an Konzepte von Zustandsdiagrammen angelehnt ist. Bei der Umsetzung des Modellierungskonzepts soll dann ein auf Cinco basierendes Tool zur Modellierung und Generierung von Alexa-Skills entwickelt werden. Hierzu müssen Codegeneratoren entwickelt werden, mit denen aus einem entworfenen Skill-Modell eine auf der ASK SDK basierende Codeumsetzung des Skills generiert werden kann. Dieses wird um zwei graphische DSLs erweitert, die es dem Entwickler vereinfachen die Konfigurationen für den Skill zu verwalten.

Zuletzt muss das ASDT getestet werden. Hierzu werden Skill-Beispiele modelliert, generiert und deployt. Für das Deployment muss eine Server-Schnittstelle - im Kontext des ASK SDK als Endpoint bezeichnet - entwickelt werden, mit der die lauffähigen ASK-Implementierungen aufgesetzt werden können. Die deployte Anwendung wird dabei durch eine WildFly Instanz verwaltet.

Integration von Dime

Nachdem die mit dem ASDT entwickelten Skills erfolgreich getestet wurden, soll das Tool so erweitert werden, dass eine parallel entwickelte Dime-App mittels Alexa angesteuert werden kann. Die bestehende Skill-DSL muss so erweitert werden, dass Dime-spezifische Elemente in den Skill-Modellierungsvorgang integriert werden können, wodurch ein Skill auf bereitgestellte Dime-Funktionalitäten, wie Prozesse und Daten aus der Dime-Datenbank, zugreifen kann. Zu diesem Aufgabenbereich gehören mehrere Erweiterungen der Skill-DSL,

Codegeneratoren und Compilern sowie an den Deploy-Mechanismen. Letztere arbeiten mit einem Docker-Container, in dem die Anwendung betrieben wird.

1.3 Aufbau der Arbeit

Zunächst wird in Kapitel 2 auf wichtige Begrifflichkeiten, notwendige Technologien und Konzepte eingegangen. Die Unterkapitel basieren zum Teil auf Ausarbeitungen von einer zu Beginn der Projektgruppe gehaltenen Seminarphase.

Zur Entwicklung der Skill-DSL mit CINCO werden abgrenzbare Module festgehalten und in Kapitel 3 beschrieben, wie mit dem ASDT ein Alexa-Skill modelliert wird. Die drei wichtigsten konzeptuellen Module hierbei sind:

- Graphische DSL zur Modellierung des Skill-Ablaufs (siehe Abschnitt 3.2)
- Textuelle DSL zur Modellierung von Alexa-Intents (siehe Abschnitt 3.3)
- Textuelle DSL zur Modellierung der Manifest-Datei (sieheAbschnitt 3.4)

Nachdem die Modellierung eines Skills beschrieben wurde, wird in Kapitel 4 auf notwendige Generierungsschritte während der Skillentwicklung eingegangen. Hierzu gehören das Generieren der vom Entwickler zu implementierenden Klassen und das Generieren des Skills beziehungsweise finaler Skill-bezogener Dateien.

Durch die Erweiterung des ASDTs zur Integration von DIME werden bei den vorangegangenen Modulen die folgenden Teilaspekte neu entwickelt oder angepasst:

- Erweiterung der graphischen Skill-DSL um DIME-Elemente wie Prozessknoten und Datenobjekte
- Anpassungen an den Codegeneratoren der Skill-DSL zum Aufrufen von DIME-Prozessen und Synchronisieren von DIME-bezogenen Daten
- Entwicklung eines Konzeptes zur Verschachtelung von Skill-Modellen

Im Kapitel 5 werden anschließend die notwendigen Schritte zum Deployen des entwickelten Skills beschrieben. Darauffolgend werden die Ergebnisse dieser Arbeit in Kapitel 6 zusammengefasst. Zuletzt wird in Kapitel 7 ein Ausblick gegeben, welche Erweiterungen an dem ASDT zusätzlich entwickelt werden könnten.

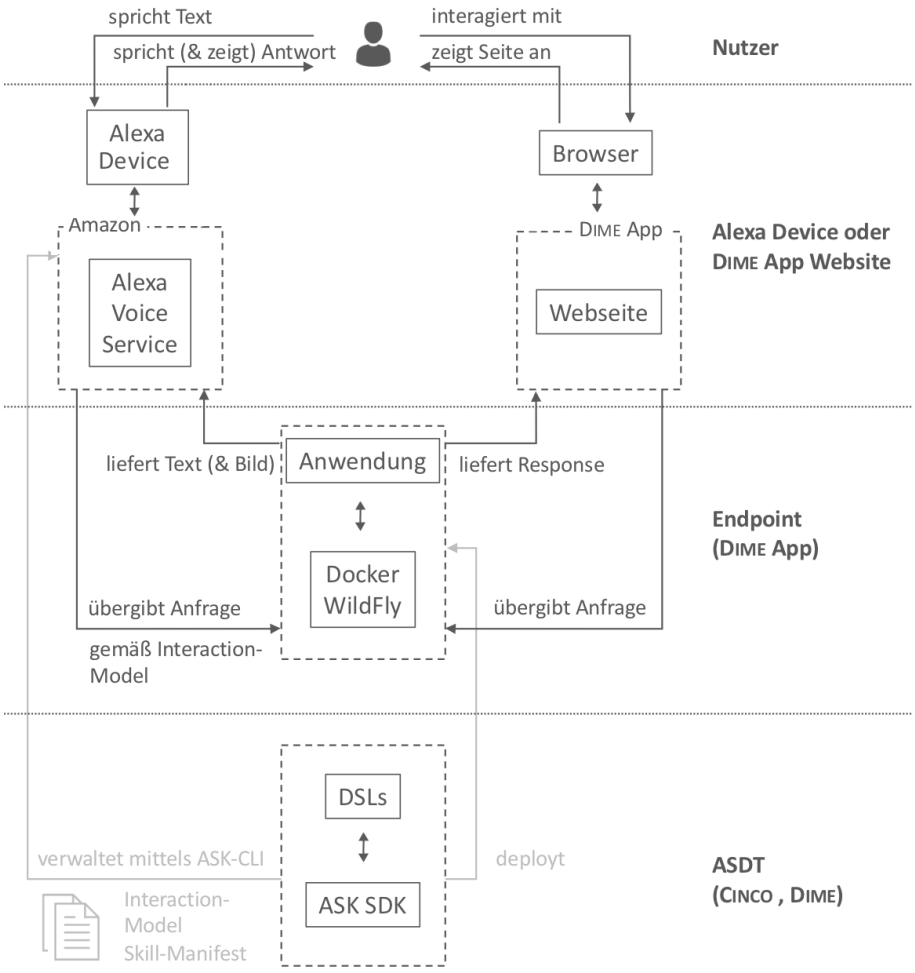


Abbildung 1.1: Architektur der Technologien

Kapitel 2

Grundlagen

In diesem Kapitel werden konzeptuelle und technische Grundlagen vorgestellt, die zum Verständnis der im Rahmen der Projektgruppe entwickelten Software hilfreich oder notwendig sind. Diese basieren zum größten Teil auf Ausarbeitungen, die im Rahmen einer Seminarphase vor dem Beginn der Projektgruppe von allen Teilnehmern angefertigt und für diese Arbeit zusammengefasst wurden.

2.1 Modellierungsgrundlagen

Die Modellierung von Systemen und dessen Visualisierung hat vor allem im Informationszeitalter eine besondere Bedeutung eingenommen. So soll die Modellierung die Effektivität des Menschen bei verschiedenen Problemlösungen verbessern, wie zum Beispiel bei der Entwicklung und Umsetzung von unterschiedlichen Verfahren [31]. In den folgenden Kapiteln werden die für diese Arbeit notwendigen Modellierungsgrundlagen vermittelt. Viele dieser Konzepte und Techniken wurden im Verlauf der Projektgruppe zur Umsetzung der festgelegten Anforderungen verwendet.

2.1.1 Modellbegriff der Informatik

Nachschlagen der Definition des Begriffs „Modell“ im Duden [20] liefert unter anderem folgende Beschreibung:

> *"Modell, das*
> *1. a. Form, Beschaffenheit, Maßverhältnisse veranschaulichende Ausführung eines vorhandenen oder noch zu schaffenden Gegenstandes in bestimmtem (besonders verkleinerndem) Maßstab"[20]*

Diese Beschreibung des Duden leitet die Thematik des Modellbegriffs im *Model Driven Software Engineering* (MDSE) passend ein. Wichtig bei der Definition ist hierbei der Fokus auf den Aspekt, dass ein Modell etwas beschreiben kann, das entweder bereits existiert, beispielsweise für Dokumentationszwecke, als auch etwas, das noch nicht vorhanden ist. Letztere Variante wird später in der Thematik des MDSE der Ansatzpunkt sein, ein Modell zu definieren, um daraus etwas zu generieren, was vorher noch nicht existiert hat, so wie im Modell beschrieben. Darüber hinaus existiert die Modellbeschreibung von Stachowiak nach dem ein Modell durch drei Merkmale geprägt wird, dem Abbildungsmerkmal, dem Verkürzungsmerkmal und dem Pragmatismusmerkmal, siehe dazu [39]. Neben diesen Ansätzen gibt es viele weitere Definitionen, auf die hier nicht weiter eingegangen werden soll.

2.1.2 Modellarten: Graphisch und Textuell

Grundsätzlich wird zwischen graphischen und textuellen Modellen unterschieden. Diese unterscheiden sich vor allem durch ihre Darstellungsform. Hierbei bestehen *textuelle Modelle* „aus Sätzen einer natürlichen Sprache oder einer Übersetzung dieser Sätze in eine (semi-)formale Sprache“[31]. *Graphische Modelle* hingegen bestehen aus verschiedenen graphischen Elementen, die zueinander in Verbindung gebracht werden.

2.1.3 Modellierungssprachen

Allgemein besteht eine Sprache aus einer Menge an Zeichenliteralen, einer darauf basierenden Syntax und einer Semantik. Die Literale können sowohl als Buchstaben, allerdings auch als simple graphische Elemente wie zum Beispiel Rechtecke, Striche oder Kreise vorliegen. Eine Syntax beschreibt die Regeln, nach denen die Literale der Sprache in Beziehung gebracht werden dürfen und die Semantik beschreibt dessen Bedeutung. Aus Zeichenliteralen erstellte Modelle können je nach Verständnis der Syntax eindeutig, mehrdeutig oder nicht interpretiert beziehungsweise übersetzt werden.

Grundlegend wird zwischen Universalsprachen und domänenspezifischen Sprachen unterschieden. Diese können zur Entwicklung von sowohl graphischen als auch textuellen Modellen verwendet werden [17].

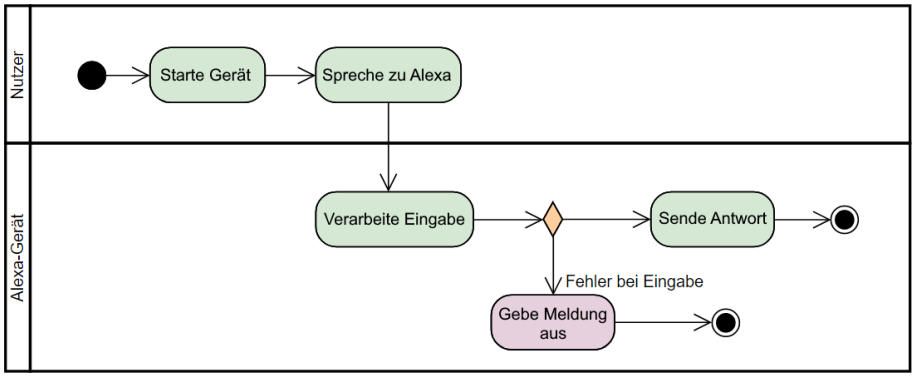


Abbildung 2.1: Beispiel eines Aktivitätsdiagrammes

Universalsprachen

Mit Universalsprachen können Lösungen für viele unterschiedliche Problemstellungen entwickelt werden. Sie werden auch Mehrzwecksprachen oder *General Purpose Languages* (GPL) genannt. Zu textuellen GPLs gehören zum Beispiel herkömmliche Programmiersprachen wie JAVA oder C++. Zu den bekanntesten Sprachen der graphischen *Mehrzweckmodellierung* gehört die *Unified Modelling Language* (UML) [32]. Diese kann bei der Entwicklung von Softwaresystemen und anderen Systemen oder Verfahren in verschiedenen Bereichen eingesetzt werden. In der UML gibt es mehrere Spracheinheiten und Syntaxkonzepte zum Erstellen verschiedener Diagrammtypen. Diese werden in der UML in Struktur- und Verhaltensdiagramme aufgeteilt. Strukturdiagramme sollen statische Komponenten eines Systems beschreiben, wie zum Beispiel Klassendiagramme. Verhaltensdiagramme sollen Änderungen zur Systemlaufzeit visualisieren. Zu Letzterem gehören zum Beispiel Aktivitätsdiagramme (siehe Abbildung 2.1) [36].

Domänenspezifische Sprachen

Den Gegensatz zu Universalsprachen bilden sogenannte domänenspezifische Sprachen oder auch *Domain Specific Languages* (DSL). Sie werden zur Lösung von Problemen einer spezifischen Domäne verwendet und sind somit in ihrem Nutzen limitiert, liefern dafür aber für die jeweilige Domäne eine bessere Lösungsmöglichkeit als die Verwendung von Universalsprachen [32]. Derzeit sind vor allem textuelle DSLs stark verbreitet. Zu den Verbreitetsten gehört zum Beispiel die Sprache SQL, die ausschließlich für Datenbankmanagement entwickelt wurde [18].

2.1.4 Metamodellierung

Ein wichtiges Thema für das Verständnis der folgenden Kapitel ist der Bereich der *Metamodellierung*. Neben der Nutzung von GPLs ist es auch möglich eigene domänenspezifische Sprachen zu erstellen und diese als Basis für Modelle zu benutzen. Hierfür muss eine DSL, sei sie textuell oder graphisch, vorab beschrieben werden. Die Definition einer DSL, also einer Modellierungssprache, wird mit der Metamodellierung umgesetzt [17].

Modellierungsebenen

Um eine Modellierungssprache zu definieren, führt die Metamodellierung sogenannte Modellierungsebenen ein. Eine Modellierungsebene definiert die Syntax und Semantik der Ebene unter ihr. Typischerweise existieren vier Modellierungsebenen [17]:

- M_3: Meta-metamodell
- M_2: Metamodell
- M_1: Modell
- M_0: Konkretes Objekt

Beispielsweise startend bei einem konkreten Objekt, werden die Elemente des Objekts und deren Bedeutung in der Modellierungsebene über ihr definiert, in diesem Fall die Ebene des Modells. Die Syntax und Semantik dieses Modells muss ebenfalls wieder beschrieben werden, dies erfolgt im Metamodell. In dieser Logik setzen sich die unterschiedlichen Modellierungsebenen fort und können durch Verwendung von reflexiven Metamodellierungssprachen terminieren.

Die Modellierungsebenen können auch an dem Beispiel von Software gezeigt werden. Die im Speicher eines Computers liegende Ausführung einer Software entspricht hierbei der Ebene des konkreten Objekts. Die Daten, die sich im Speicher befinden, wurden in der eigentlichen Software modelliert, welche ausgeführt wird, entsprechend der Ebene des Modells. Im Kontext von MDSE wird diese Software durch ein beispielsweise graphisches Modell modelliert, wobei sich dieses Modell auf derselben Ebene befindet wie die Software. Die Syntax und Semantik des Modells, in dem die Software beschrieben wird, muss ebenfalls spezifiziert werden, dies passiert auf der Metamodell Ebene.

2.2 Modellgetriebene Softwareentwicklung

Die *Modellgetriebene Softwareentwicklung* ist ein Grundlagenthema für alle folgenden Themen. Zunächst wird auf den Transformationsbegriff eingegangen, um anschließend den Unterschied zwischen einem Generator und einem Interpreter zu betrachten und auf partielle und volle Codegenerierung einzugehen.

Das allgemeine Ziel von modellgetriebener Softwareentwicklung lässt sich als die vollständige Codegenerierung aus Modellen zusammenfassen. Dabei steht im Zentrum von modellgetriebener Softwareentwicklung einerseits das Artefakt Modell und andererseits die Codegenerierung selbst [17]. Hierfür existieren verschiedene Konzepte, die im Folgenden näher betrachtet werden. Im späteren Verlauf wird diese Idee im Kontext der Metamodellierung aufgegriffen.

2.2.1 Transformationsbegriff

Der klassische Ansatz der *Transformation* eines Modells in einen Programmcode, wird als *Model-To-Code-Transformation* [19] oder auch als *Model-To-Text-Transformation* [17] bezeichnet. Unabhängig von der Bezeichnung umschreibt Model-To-Code/-Text den Vorgang, aus einem vorhandenen Modell typischerweise einen Programmcode zu erzeugen. Dabei muss das Modell selbst nicht graphisch formuliert worden sein, es kann auch ein textuelles Modell vorliegen, welches in Code/Text transformiert wird.

Darüber hinaus existiert neben Model-To-Code/-Text ein weiterer Ansatz der sogenannten *Model-To-Model-Transformation* [19, 17]. Das Ergebnis der Transformation eines Modells ist weiterhin ein Modell, welches anschließend entweder erneut in ein Modell oder in einen Text/Code transformiert werden kann.

2.2.2 Generator und Interpreter

Eine wichtige technische Frage beim modellgetriebenen Softwareentwicklungs-Ansatz ist die Frage, ob überhaupt Code aus einem Modell generiert werden soll, oder anstelle dessen die erstellten Modelle interpretiert werden sollen, ohne Quellcode zu erzeugen. Diese beiden Ansätze beschreiben die Gedanken hinter einem *Codegenerator* und einem *Interpreter* im Rahmen der modellgetriebenen Softwareentwicklung [17].

Der wichtigste Unterschied der beiden Varianten besteht in der Frage, ob zu dem bisher vom Entwickler zusammengetragenen Programmquellcode zusätzlicher generiert werden

soll oder nicht. Soll zu einem Modell durch einen Generator Quellcode generiert werden, entsteht aus dem Modell idealerweise direkt eine lauffähige Software. Beim Einsatz eines Generators haben Änderungen am Modell nach Generierung der Software keinen Einfluss mehr auf die Software, hierfür müsste der Quellcode und damit die Software erneut generiert werden. Bei Verwendung eines Interpreters werden hingegen die Modelle vom Interpreter gedeutet und kein zusätzlicher Quellcode generiert. Änderungen an den Modellen haben dann auch direkt Änderungen an der Software zufolge.

2.2.3 Partielle und volle Codegenerierung

Trotz des eigentlichen Ziels von modellgetriebener Softwareentwicklung, den gesamten Quellcode für die Software aus einem oder mehreren Modellen zu generieren, existieren grundsätzlich zwei unterschiedliche Konzepte der Codegenerierung bei Wahl eines Generators. Einerseits gibt es die Möglichkeit der vollständigen Quellcodegenerierung, andererseits ist es auch möglich nur einen Teil des Quellcodes der späteren fertigen Software generieren zu lassen. Der Unterschied dieser beiden Varianten, also entweder ein Generator mit *partieller Codegenerierung* oder ein Generator mit *voller Codegenerierung*, wird insofern wichtig, als dass bei partieller Codegenerierung nach der Generierung des Quellcodes noch Änderungen an diesen durch Programmierer vorgenommen werden müssen [17, 34].

Ein Beispiel für eine partielle Codegenerierung sind UML Klassendiagramme, die nur den Rahmen der Klassen in Quellcode generieren, entsprechend der Vorgaben im graphischen Klassendiagramm-Modell. Die eigentlichen Inhalte der Funktionen der Klassen werden vom Programmierer erst nach Generierung des Quellcodes geschrieben.

Ein Beispiel für eine volle Codegenerierung ist das CINCO-Framework [40]. In diesem Framework wird nach Generierung der Software keine Änderungen am generierten Quellcode mehr vorgenommen werden.

Bei partieller Codegenerierung stellt sich die Frage, wie vorzugehen ist, falls ein bereits generierter Programmcode manuell angepasst wurde und durch Änderungen am Modell neu generiert werden müsste. Konzeptuell müsste der manuell angepasste Programmcode nun durch die neue Generierung überschrieben werden, da das Modell keine Kenntnis von den Änderungen am Programmcode hat. Dieses Problem wird typischerweise als *Roundtrip-Problem* bezeichnet. Ein Lösungsansatz ist hier, falls Änderungen am generierten Programmcode vorgenommen werden, die Änderungen im Model ebenfalls vorzunehmen und umgekehrt. Diese Thematik wird mit dem Begriff des *Round-trip Engineering* bezeichnet [30]. Es beschreibt, wie manuelle Änderungen am generierten Quelltext im Modell ebenfalls

hinterlegt werden und umgekehrt, wie Änderungen am Modell im Quelltext übernommen werden können.

Ein Ansatz dieses Problem zu lösen wird mit dem Begriff des *Generation Gap Patterns* bezeichnet [42]. Die Idee des Generation Gap Pattern sieht vor, die Änderungen nicht am generierten Programmcode selbst zu tätigen, sondern stattdessen eine neue Unterklasse der generierten Klasse einzubauen und dort die Änderungen vorzunehmen. Bei Neugenerierung des Programmcodes aus dem Modell würde nun zwar die Originalklasse überschrieben, nicht jedoch die Klasse mit den vorgenommenen manuellen Änderungen.

Eine weitere Möglichkeit das Problem zu lösen, sind sogenannte *Protected Code Regions* [21]. Wird generierter Programmcode geändert, so kann der Programmierer seine Änderungen als sogenannte Protected Code Regions markieren. Der Generator erkennt diese geschützten Regionen und überschreibt sie nicht. Somit bleiben sie auch bei erneuter Code-Neugenerierung weiterhin vorhanden. Bei dem Beispiel UML müssten die Klassendiagramme jedoch nach Änderungen am Programmcode manuell vom Programmierer auf den aktuellen Stand gehalten werden.

2.3 Xtend

Für die Codegenerierung ist Xtend eine geeignete Programmiersprache [27]. Sie baut auf Java auf, sorgt für kompakteren Code und ergänzt um einige nützliche und mächtige Funktionen. Darunter zählen beispielsweise Typinferenz, Reduktion redundanter Schlüsselwörter, Templates für bessere Lesbarkeit großer Textblöcke und die Möglichkeit Klassen im Nachhinein um Methoden zu erweitern.

Xtend bleibt dabei vollständig kompatibel zu Java. Jede Java-Bibliothek und jede Java-Klasse wird nativ unterstützt. Dies liegt unter anderem daran, dass Xtend-Quellcode direkt in Java-Quellcode kompiliert werden kann. Die generierten Java-Dateien können dann vom Java-Compiler in Bytecode umgesetzt werden.

Auf die Funktionen von Xtend, die für die Codegenerierung am nützlichsten sind, wird im Folgenden näher eingegangen. Zunächst folgt jedoch eine kurze Einführung zur Syntax von Xtend.

2.3.1 Hello World

Für den Einstieg in die Programmiersprache Xtend wird zunächst ein einfaches „Hello World“-Programm vorgestellt (siehe Listing 2.1) und mit dem entsprechenden Code in Java verglichen (siehe Listing 2.2).

```
1 // Hello World Programm in Xtend
2 class HelloWorld {
3   def static void main(String[] args) {
4     println('Hello World')
5   }
6 }
```

Listing 2.1: „Hello World“-Programm in Xtend

```
1 // Hello World Programm in Java
2 public class HelloWorld {
3   private static void main(String[] args) {
4     System.out.println("Hello World");
5   }
6 }
```

Listing 2.2: „Hello World“-Programm in Java

Aus dem „Hello World“-Beispiel werden die ersten Unterschiede zwischen XTEND und JAVA ersichtlich:

- Die Sichtbarkeit von Klassen und Methoden wird standardmäßig auf `public` gesetzt.
- Methoden werden mit dem Schlüsselwort `def` eingeleitet.
- Es steht eine vereinfachte `println`-Methode ohne Angabe eines `PrintStream` zur Verfügung.
- String-Literale können mit einfachen Anführungszeichen (`'`) statt mit doppelten (`"`) geschrieben werden.
- Das Semikolon (`;`) am Ende einer Zeile kann ausgelassen werden.

2.3.2 Templates

Templates ermöglichen das Erzeugen von langen Zeichenketten in einer lesbaren Notation. Das Einbinden von Variablen geschieht ebenfalls einfacher und ist übersichtlicher. Templates sind von drei einfachen Anführungszeichen (`'''`) umgeben. Sie erzeugen eine `CharSequence`, die jedoch automatisch in einen `String` umgewandelt wird, wenn es der Kontext vorschreibt. Sie können also überall verwendet werden, wo Strings erwartet werden.

Statt mit Konkatenationen (`"Hello " + getAdjective + "World!"`) können Codeelemente wie Variablen und Methoden direkt in Templates eingebunden werden. Dazu wird der einzubettende Code in Guillemets (`«»`, französische Anführungszeichen) gefasst. Kommentare innerhalb eines Templates werden mit drei Guillemets (`«««`) angeführt.

Es ist auch durchaus üblich einzelne Methoden als Templates zu verwenden, wie in Listing 2.3 zu sehen ist. Der Methodenrumpf enthält dann ausschließlich das Template. Ein Methodenaufruf `someHTML('Hello World!')` resultiert in den Text aus Listing 2.4. Die Einrückungen im Template werden intelligent in der Ausgabe übernommen, was für eine ebenso gut lesbare Ausgabe sorgt. Im Editor werden Quellcode und Whitespaces markiert, die in der Ausgabe enthalten sein werden, um schon während des Programmierens besser erkennen zu können, wie das Ergebnis aussehen wird.

```
def someHTML(String content) '''
  <html>
    <body>
      «content»
    </body>
  </html>
  «« Dies ist ein Kommentar innerhalb eines Templates
  «« und wird nicht in der Ausgabe erscheinen.
'''
```

Listing 2.3: Template-Methode

```
<html>
  <body>
    Hello World!
  </body>
</html>
```

Listing 2.4: Ergebnis der Template-Methode

```
def someHTML(List<Paragraph> paragraphs) '''
  <html>
    <body>
      «« Verschachtelt jeden Paragraphen in je einen <div>-Block
      «FOR p: paragraphs BEFORE '<div>' SEPARATOR '</div><div>' AFTER '</div>'»
        «« Fügt die Überschrift nur hinzu, wenn der Paragraph eine besitzt
        «IF p.headLine !== null»
          <h1>«p.headline»</h1>
        «ENDIF»
        <p>«p.text»</p>
      «ENDFOR»
    </body>
  </html>
'''
```

Listing 2.5: Schleifen und bedingte Ausführung in Templates

Des Weiteren sind auch Ausdrücke mit `For`-Schleifen und `If`-Abfragen möglich. Die entsprechende Syntax ist aus Listing 2.5 zu entnehmen. `For`-Schleifen bieten zusätzlich die Funktion verschiedene Textstücke vor (`BEFORE`) und nach (`AFTER`) der Schleife sowie zwischen jeder Iteration (`SEPARATOR`) einzufügen.

2.3.3 Extensions

Eine weitere wichtige Funktion, die Xtend bietet, sind Extensions. Sie sind der Namensgeber der Programmiersprache und erlauben bereits vorhandene Klassen um Funktionen zu erweitern, ohne sie selbst zu bearbeiten. Dadurch können auch Klassen, dessen Quellcode unzugänglich ist (wie beispielsweise bei Bibliotheken), um eigene Methoden ergänzt

werden. Es wird dafür ein syntaktischer Trick genutzt: Bei einem Methodenaufruf darf der erste Parameter statt in den Klammern auch vor dem Methodennamen als Empfänger gestellt werden (siehe Listing 2.6).

Um nun eine fremde Klasse zu erweitern, kann beispielsweise eine gesonderte statische Helferklasse erstellt und mit den benötigten Extension-Methoden bestückt werden. Diese müssen als `static` markiert sein (Listing 2.7). Wenn sie schließlich eingesetzt werden sollen, muss die Helferklasse mit `import static extension HelperClass` importiert werden (Listing 2.8).

Es ist ebenfalls möglich die Methoden einer einzelnen Referenz als Extensions zur Verfügung zu stellen. Dazu muss bei der Deklaration der Variable oder des Attributs das Schlüsselwort `extension` hinzugefügt werden (siehe Listing 2.9).

```
1 class Student {
2   def doSomething(Object obj) {
3     // ...
4   }
5
6   def doSomethingViaExtensions(Object obj) {
7     obj.doSomething // Ruft this.doSomething(obj) auf
8   }
9 }
```

Listing 2.6: Lokale Extension-Methode

```
1 class StringExtensions {
2   def static toFirstUpper(String s) {
3     // ...
4   }
5 }
```

Listing 2.7: Extension via einer Helferklasse

```
1 import static extension StringExtensions.*
2
3 class Student {
4   def doSomethingViaExtensions() {
5     'hello'.toFirstUpper // Ruft StringExtensions.toFirstUpper("hello") auf
6   }
7 }
```

Listing 2.8: Einsatz einer Extension-Helferklasse

```
1 class Course {
2   var extension student = new Student
3
4   def doSomethingViaExtensions() {
5     'Henry'.setName // Ruft student.setName("Henry") auf
6   }
7 }
```

Listing 2.9: Extension via Schlüsselwort `extension`

Dies sind noch nicht alle Verbesserungen und Unterschiede von XTEND gegenüber JAVA. Die hier aufgeführten Funktionen beschränken sich lediglich auf die für die Codegenerierung wichtigen Aspekte von XTEND. Auf weitere Details in den technischen Unterschieden soll hier nicht weiter eingegangen werden. Genauere Informationen sind der XTEND-Dokumentation zu entnehmen [27].

2.4 Xtext[1]

Im Folgenden soll die Modellierung *textueller domänenspezifischer Sprachen (DSLs)* mithilfe des Xtext-Frameworks vorgestellt werden. Dabei sollen die grundlegenden Funktionalitäten des Frameworks vorgestellt werden.

Xtext ist ein Framework zur Entwicklung von textuellen Programmiersprachen und ihrer Entwicklungsumgebung basierend auf dem *Eclipse Modeling Framework* (EMF) [34, 15]. EMF ermöglicht das Bauen von Modelleditoren durch Bereitstellung der benötigten Ressourcen und der Metamodellumgebung inklusive Generierung, wobei die Modelle in Form von Metamodellen (*Ecore-Modell* [23]) definiert werden [22]. Dafür bietet das Xtext-Framework selbst eine Backus-Naur-Form (BNF) ähnliche Grammatiksprache zur Erstellung von eigenen textuellen DSLs. Xtext ist dabei Teil des *Eclipse Modelling Projects* und kann als Eclipse-Plug-in installiert werden [34].

Sprachkonstrukte auf Basis der mit Xtext definierten Grammatik werden von einem Parser (Antlr [38]) in einen sogenannten abstrakten Syntaxbaum (*Abstract Syntax Tree*, AST) überführt. Der AST beruht auf einem Datenmodell, einem Ecore-Modell, welches entsprechend der Grammatik von Xtext generiert wird. Es ist jedoch auch möglich, die Grammatik auf Basis eines bereits bestehenden Ecore-Modells zu entwickeln. Darüber hinaus können bestehende Sprachen erweitert werden. Das Ecore-Modell ist somit das Metamodell des AST und kann für die weitere modellgetriebene Entwicklung von zum Beispiel Validatoren und Generatoren verwendet werden.

Xtext generiert aus der Grammatik und dem dazugehörendem Ecore-Modell Editor, Parser sowie Vorlagen für Codegeneratoren und Validierungen [34]. Die Programmiersprache Xtend bietet einen Template-basierten Ansatz der Entwicklung an (siehe Unterabschnitt 2.3.2) und eignet sich daher besonders für das Schreiben von Codegeneratoren.

2.4.1 Codegenerierung

Anhand des Beispiels aus dem Xtext-Tutorial soll dargestellt werden, wie Codegenerierung mit Xtend für eine Xtext Sprache realisiert werden kann [28]. Als erster Schritt wird in der Eclipse IDE ein neues Xtext-Project erstellt. Es werden automatisch die benötigten Klassen und Ordnerstrukturen erstellt. Bei diesem Schritt müssen Name und Dateiendungen festgelegt werden. Nun wird die Grammatik betrachtet, welche die Sprache und ihre erlaubte Syntax definieren soll. Die erste Regel ist die Startregel `Domainmodel`.

[1] Sofern nicht anders angegeben beziehen sich sämtliche Aussagen in diesem Abschnitt auf die Quelle [28], auf Praxiserfahrungen oder mündliche Aussagen der Xtext-Entwickler.

Von ihr werden alle anderen Regeln abgeleitet. Mit der nächsten Regel werden die erlaubten Paketdefinitionen (`PackageDeclaration`) festgelegt. Dieses kann eine beliebige Anzahl von `AbstractElem` enthalten und muss einen Namen haben, der `ErlaubterName` entspricht. Ein `AbstractElem` kann entweder eine *PackageDeclaration*, ein Datentyp (`DatenTyp`) oder ein Import (`Import`) sein. Diese beispielhafte Grammatik ist in Listing 2.10 dargestellt.

```
1  grammar org.xtext.mydsl.DslTest with org.eclipse.xtext.common.Terminals
2  generate dslTest "http://www.xtext.org/mydsl/DslTest"
3
4  Domainmodel:
5    ( elems += AbstractElem )* ;
6
7  PackageDeclaration:
8    'package' name = ErlaubterName '{'
9      ( elems += AbstractElem )*
10   '}' ;
11
12 AbstractElem:
13   PackageDeclaration | DatenTyp | Import ;
14
15 ErlaubterName:
16   ID ( '.' ID )* ;
17
18 Import:
19   'import' importedNamespace = ErlaubterNameMitPunkt ;
20
21 ErlaubterNameMitPunkt:
22   ErlaubterName '.*'? ;
23
24 DatenTyp:
25   EinfacherDatenTyp | Elem ;
26
27 EinfacherDatenTyp:
28   'datentyp' name = ID ;
29
30 Elem:
31   'elem' name = ID ( 'extends' superType = [Elem|Erlaubtername] )? '<'
32     ( features += Feature ( ',' features += Feature )* )?
33   '>' ;
34
35 Feature:
36   ( list ?= 'list' )? name = ID ':' type = [DatenTyp|ErlaubterName] ;
```

Listing 2.10: Beispiel einer Grammatik für DSLs in XTEXT

```
class DslTestGenerator extends AbstractGenerator {

  @Inject extension IQualifiedNameProvider
  override void doGenerate(Resource recource, IFileSystemAccess2 fsa,
      IGeneratorContext context) {
    for (e : resource.allContents.toIterable.filter(Elem)) {
      fsa.generateFile(e.fullyQualifiedName.toString('/') + '.java', e.compile)
    }
  }

  def compile(Elem e) '''
    «IF e.eContainer.fullyQualifiedName !== null»
      package «e.eContainer.fullyQualifiedName»;

    «ENDIF»
    public class «e.name»«IF e.superType !== null» extends «
        e.superType.fullyQualifiedName»«ENDIF» {
      «FOR f : e.features»
        «f.compile»
      «ENDFOR»
    }
  '''

  def compile(Feature f) '''
    private «f.type.fullyQualifiedName» «f.name»;

    public «f.type.fullyQualifiedName» get«f.name.toFirstUpper»() {
      return «f.name»;
    }

    public void set«f.name.toFirstUpper»(«f.type.fullyQualifiedName» value) {
      «f.name» = value;
    }
  '''

  // ...

}
```

Listing 2.11: Beispiel eines Generators in XTEND

Der Generator wird in XTEND entwickelt (siehe Listing 2.11). Die Methode `doGenerate` soll alle JAVA-Klassen generieren. Dafür werden zunächst alle Inhalte aus der Ressource ausgelesen und zu einem iterierbaren Objekt konvertiert. Daraus sollen alle `EObjects` des Typs `Elem` gefiltert werden. Der Name der zu erstellenden JAVA-Datei wird dem Objekt entnommen und auf einen korrekten Namen überprüft. Der Inhalt wird dann mit der `compile`-Methode geschrieben, die einen String ausgibt.

```
1  import ordner.andererordner.*
2
3  datentyp int
4  datentyp str
5  datentyp voiceCommand
6  datentyp test
7
8  package alexa.skills {
9    elem Skill <name: str, beschreibung: str>
10   elem AlexaSkill extends Skill <trigger: voiceCommand, list fns: Function>
11   elem Function <name: str>
12 }
```

Listing 2.12: Beispiel einer textuellen DSL

Da der Generator für dieses Beispiel vollständig ist, kann die Eclipse-Applikation gestartet werden. Dies erfolgt mit einem Rechtsklick auf das übergeordnete Paket, in welchem der Generator liegt. Es wird eine neue Eclipse-Instanz geöffnet, in der Modelle erstellt werden können. Für jedes Objekt des Modells wird, sofern dieses syntaktisch korrekt ist, automatisch eine entsprechende Java-Klasse erzeugt.

In Listing 2.12 ist ein mögliches Modell der Grammatik aus Listing 2.10 zu sehen. Es werden dazu die vier Datentypen `int`, `str`, `voiceCommand` und `test` definiert. In dem Package `alexa.skills` gibt es die Klassen `Skill`, `AlexaSkill` und `Function`. `Skill` besitzt die Attribute `name` und `beschreibung`, die jeweils von dem Datentyp `str` sind. `AlexaSkill` erbt von `Skill`, besitzt also beide Attribute und zusätzlich ein Attribut `trigger` des Datentyps `voiceCommand` und eine Liste von `Functions`. Die `Function` Klasse hat lediglich einen `name` des Typs `str`.

2.4.2 Bestandteile von Xtext-Editoren

Integrale Bestandteile von modernen Entwicklungsumgebungen sind Features wie *Code Completion*, semantische Validierung oder Syntaxhighlighting, die den Entwickler bei seiner Arbeit unterstützen und die Entwicklung erleichtern. Im Folgenden sollen nun einige der im vorherigen Abschnitt bereits angesprochenen Bestandteile von Xtext-basierten Editoren vorgestellt werden.

Validierung

XTEXT-basierte Editoren bieten eine standardmäßige Syntaxvalidierung an, welche die richtige Abfolge von Regeln und Schlüsselwörtern überprüft. Diese syntaktische Validierung kann vom *Language Engineer* (dem Entwickler der Sprache) um Validierungen der statischen Semantik ergänzt werden. Dazu generiert XTEXT bei der Generierung der Sprache Basisklassen, welche vom Entwickler individuell angepasst werden können.

Scoping und Code Completion

Mit *Scoping* können referenzierbare Elemente definiert werden. Wenn in einer Grammatik eine *Cross Reference* (Referenz auf ein Objekt eines bestimmten Grammatikelements) definiert wurde, wird zwar der Typ des Grammatikelements bestimmt, aber nicht wo ein solches Element gefunden werden kann. Hier kommt der Scope zum Einsatz. XTEXT generiert bei der Sprachgenerierung zugleich einen *Scope Provider*. Dieser hat die Aufgabe, Scopes für eine gegebene Referenz in einem gegebenen Kontext zu liefern. Der Scope enthält alle Elemente, die in diesem Kontext referenzierbar sind. Dabei können auch mehrere Scopes ineinander verschachtelt sein. Das XTEXT-Framework bietet außerdem Basisimplementierungen für Standard-Usecases an, beispielsweise das Importieren von anderen Dateien oder Elementen aus anderen Dateien.

Die Hierarchie eines Scopes ist normalerweise so aufgebaut, dass seine äußerste Schicht durch den sogenannten *Global Scope* gegeben ist. Dieser enthält unter anderem alle importierten referenzierbaren Elemente. Innerhalb des Global Scope ist der *Local Scope* eingenistet. Dieser enthält alle referenzierbaren Elemente, die aus derselben Datei stammen. In dem Local Scope können noch weitere Scopes, wie ein Funktions-Scope eingenistet sein, welche dann alle referenzierbaren Elemente einer Funktion enthalten würde.

Auf dem Inhalt des Scopes aufbauend, bietet XTEXT *Code Completion* (Codevervollständigung) an. Standardmäßig generiert XTEXT bei der Sprachgenerierung einen *Proposal Provider*, welcher Code Completion für Schlüsselwörter und Cross References bereitstellt. Dieser Proposal Provider kann vom Entwickler nach den eigenen Anforderungen und Wünschen erweitert und angepasst werden.

Eine weitere Möglichkeit der Code Completion ist die Integration von *XML Templates*, mit der es auf einfache Art und Weise möglich ist ganze Sprachkonstrukte durch Code Completion im Editor der DSL einzufügen.

Weitere Editor Features von Xtext basierten Editoren

Weitere Editor Features, die Xtext-basierte Editoren standardmäßig haben und die vom Entwickler gegebenenfalls individuell angepasst und/oder erweitert werden können, sind folgende:

Outline View: Xtext bietet für jede generierte Editorumgebung eine standardmäßige Eclipse Outline View zur übersichtlichen Darstellung des Inhaltes von Dateien der entwickelten Sprache. Nach Wunsch kann diese vom Language Engineer entsprechend angepasst und modifiziert werden.

Hyperlinking: Jede Cross Reference in einer Datei der entwickelten Sprache verfügt im Editor über einen Hyperlink auf die Definition des referenzierten Elements. Dieser Hyperlink ist über das Drücken der `STRG`-Taste bei gleichzeitigem Mausklick auf die Referenz erreichbar.

Syntax Highlighting: Xtext bietet für jede generierte Editorumgebung das standardmäßige Eclipse-Highlighting an (Highlighting von Keywords, Kommentaren, Strings und weiteren).

2.5 Cinco SCCE Meta Tooling Suite

Wie bereits erwähnt, soll im Rahmen dieser Projektgruppe das Metatool Cinco verwendet werden, um eine DSL zur Entwicklung von Alexa-Skills zu entwerfen. Hierzu werden im Folgenden die einzelnen Metamodellierungssprachen sowie notwendigen Features des Tools im Detail erläutert.

Die Cinco *SCCE Meta Tooling Suite* (Cinco) ist eine integrierte Entwicklungsumgebung, welche die Entwicklung von domänenspezifischen, graphischen Modellierungswerkzeugen ermöglicht. Besonders im Fokus steht die einfache Erstellung von Modellierungswerkzeugen für den Nutzer, sodass die Komplexität hinter dem Werkzeug verborgen bleibt.

Cinco verwendet die Eclipse *Rich Client Platform* als Grundlage (Unterabschnitt 2.5.1). Für die Datenverwaltung und für die Spezifikation von Metamodellen wird EMF mit dem Metamodell Ecore verwendet. Xtext und Graphiti werden für die Realisierung von Text- und Grafikeditoren benutzt. Dabei ist Graphiti eine Programmier-API, die zur Entwicklung von graphischen Editoren für Domänenmodelle verwendet wird.

Im Fokus von Cinco stehen komplexe graphbasierte Modelle, die unterschiedliche Arten von Knoten und Kanten mit einer individuellen Repräsentation und verschiedenen Beziehungen zueinander umfassen. Für die Umsetzung der strukturellen und visuellen Eigenschaften des Werkzeugs werden drei textuelle Spezifikationssprachen *Meta Graph Language* (MGL), *Meta Style Language* (MSL) und *Cinco Product Definition* (CPD) benötigt. Diese Spezifikationssprachen werden in den nächsten Abschnitten in detaillierter Form betrachtet. Anhand der Spezifikation, welche durch die Definition der MGL und MSL beschrieben wird, erfolgt die Beschreibung eines Modells mithilfe von Metamodellierungssprachen. Die CPD stellt den Einstiegspunkt in die Generierung dar, sodass aus der Spezifikation mithilfe des Cinco Produktgenerators das dazugehörige Cinco-Produkt generiert wird. Das entstandene Produkt beinhaltet das spezifizierte Modellierungswerkzeug, welches unter anderem aus einem Graphiti-Editor besteht. An dieser Stelle ist es dem Nutzer möglich die Funktionalität zu modellieren, welche zuvor in der MGL und MSL definiert wurde (siehe Abbildung 2.2) [40].

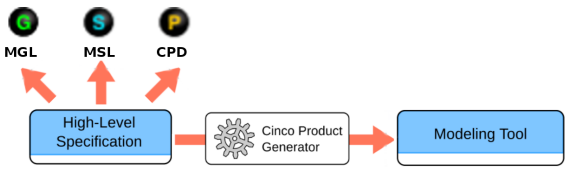


Abbildung 2.2: Generierung des Modellierungswerkzeugs aus einer abstrakten Werkzeugspezifikation [29]

In Unterabschnitt 2.5.2 und Unterabschnitt 2.5.3 werden die erwähnten Spezifikationssprachen in detaillierter Form betrachtet, wobei nur auf die wichtigsten Eigenschaften eingegangen wird.

2.5.1 CINCO als Teil der ECLIPSE Rich Client Platform

Im Rahmen der Projektarbeit werden CINCO und DIME durch neue Funktionalitäten erweitert. Da CINCO und DIME auf der ECLIPSE *Rich Client Platform* (RCP) basieren, ist dies sehr simpel. Die RCP dient zur Erweiterung und Anpassung der ECLIPSE *Platform* mithilfe von Plug-Ins.

ECLIPSE

ECLIPSE wurde ursprünglich von der ECLIPSE *Foundation* als Entwicklungsumgebung (*Integrated Development Environment*, IDE) für JAVA, C/C++ oder auch PHP entwickelt und bereitgestellt.

Durch den modularen und erweiterbaren Aufbau wird die ECLIPSE *Platform* jedoch auch verstärkt dafür eingesetzt auf der technologischen Grundlage der Software eigene JAVA-Programme zu entwickeln, die auf die grafische Benutzeroberfläche sowie weiteren Komponenten zugreifen. Die IDE sowie weitere Projekte finden sich auf der Webseite der ECLIPSE *Foundation* [24].

ECLIPSE ist ein kostenloses, plattformunabhängiges Open-Source-Projekt, das sich insbesondere durch seine starke Erweiterbarkeit und Anpassbarkeit auszeichnet. Viele der Erweiterungen lassen sich aus dem in ECLIPSE integrierten *Marketplace* herunterladen und installieren. ECLIPSE sowie seine Erweiterungen sind in JAVA geschrieben. Darum enthält das ECLIPSE SDK (*Software Development Kit*) neben der ECLIPSE *Platform* und dem *Plug-In Development Environment* auch die *Java Development Tools*.

Außerdem ist ECLIPSE eine „Plattform“, weil diese keine abgeschlossene Anwendung ist, sondern darauf ausgerichtet ist, unbegrenzt erweitert werden zu können [26]. Die ECLIPSE *Platform* kann mithilfe von beliebig vielen, gleichzeitig aktiven Plug-Ins angereichert werden. Dadurch kann ECLIPSE zu einer beliebigen IDE gestaltet werden, die auch für initial nicht angedachte Sprachen, wie beispielsweise *Scala*, verwendet werden kann. Es ist jedoch auch möglich mittels der *Rich Client Platform* eine Anwendung auf der ECLIPSE *Platform* aufzusetzen, die nicht den Charakter einer IDE hat.

Rich Client Platform

Die *Rich Client Platform* (RCP) dient zur Erweiterung und Anpassung der ECLIPSE *Platform* mithilfe von Plug-Ins. Mit der RCP ist es möglich *Rich Client Applications* zu erstellen. Die Plattform ist so ausgelegt, dass mit ihr eine Vielzahl von Client-Anwendung erstellt werden kann. Diese benötigten Plug-Ins sind:

- ECLIPSE Runtime
- Standard Widget Toolkit (SWT)
- JFace
- Workbench
- Sonstige Abhängigkeiten für die Workbench

Der Vorteil eine Anwendung auf Basis der RCP zu entwickeln, liegt darin, dass mittels der *Widgets* aus dem *Standard Widget Toolkit* (SWT) das Design der Anwendung dem des Betriebssystems entspricht und bereits eine Unterstützung für verschiedene Systeme gegeben ist. Darüber hinaus können bestehende Komponenten, wie eine Hilfeoberfläche oder ein Update-Manager, wiederverwendet werden, anstatt diese selber implementieren zu müssen. Ein weiterer Vorteil ist, dass die offene Architektur es Dritten erlaubt diese Anwendungen ebenfalls zu erweitern, wie es im Zuge dieser Projektgruppe mit CINCO und DIME erfolgt [25].

2.5.2 Meta Graph Language

Die *Meta Graph Language* (MGL) legt fest, welche Typen von Komponenten ein CINCO-Produkt enthalten darf und welche Funktionalitäten die einzelnen Typen von Komponenten haben. Der wichtigste Modelltyp ist dabei das `GraphModel`, welches die Wurzel des graphbasierten Modells darstellt. Das `GraphModel` selbst kann weitere Modelltypen wie `Node`s, `Container` und `Edge`s besitzen.

Mithilfe von `Edge`s können Kantentypen definiert werden, welche die Beziehungen zwischen anderen Modellelementen herstellen. `Node`s repräsentieren Knotentypen, welche ein- und ausgehende Kanten besitzen können sowie Attribute oder Annotationen. Als eine Erweiterung von Knotentypen dienen Containertypen. Diese können andere Modellelemente durch sogenannte `containableElements` beinhalten. Neben einfachen Attributen, welche Typen wie `EString` oder `EInt` besitzen, ist es möglich eigene Typen (*user defined types*)

zu definieren und als Attribute zu verwenden. Des Weiteren können auch Aufzählungen (*enumerations*) angelegt und verwendet werden. Außerdem lässt sich Vererbung zwischen einzelnen Modelltypen mittels `extends` realisieren. Hinzu kommt, dass die Modellelemente mittels Annotationen weitere Funktionalitäten erhalten können. Durch die Annotation `@style` erhält ein Modellelement ein Aussehen, welches in der *Meta Style Language* definiert ist, worauf in Unterabschnitt 2.5.3 eingegangen wird.

Durch Annotationen sind auch *Actions* und *Hooks* für ein Modellelement möglich. Dabei beschreibt eine Action eine Aktion, die aktiv von einem Nutzer ausgeführt wird, beispielsweise eine `@doubleClickAction`. Eine Aktion wird gestartet, sobald der Nutzer einen Doppelklick auf das annotierte Modellelement ausgeführt hat. Für passive Aktionen werden sogenannte *Hooks* verwendet, die ohne aktives Eingreifen des Nutzers ausgeführt werden. Beispielsweise wird bei einem `postCreate`-Hook eine Aktion automatisch nach dem Erstellen eines Elementes ausgeführt. Zuletzt existieren sogenannte *Prime Referenzen*, welche es erlauben andere Modelle zu referenzieren. Dies geschieht durch ein Attribut eines Knotentypen oder Containertypen. Dieses Attribut wird automatisch gesetzt, sobald der Knoten beziehungsweise Container, welcher Repräsentant für ein anderes Modell ist, erstellt wird. Diese *Prime*-Knoten/-Container agieren äquivalent zu normalen Knoten/Containern, aber besitzen zusätzlich ein *Prime-Statement* [33].

2.5.3 Meta Style Language

Nachdem in der MGL die Typen von Komponenten für ein CINCO-Produkt festgelegt wurden, wird nun die *Meta Style Language* (MSL) betrachtet, in der das Aussehen für Typen von Komponenten festgelegt werden kann.

Um das Aussehen von `Node`s oder `Container` zu definieren, wird ein `nodeStyle` verwendet. Mithilfe eines `nodeStyle` kann nach Belieben Form, Aussehen, Größe und Position angepasst werden. Auch für Kanten kann ein spezifisches Aussehen definiert werden, weshalb dafür in der MSL ein `edgeStyle` verwendet wird. Eine Kante besteht grundlegend aus einer einfachen Linie, dessen Farbe, Stil und Breite modifiziert werden kann. Außerdem kann diese Linie um sogenannte *Dekoratoren* erweitert werden, sodass eine Kante beispielsweise mit einer Pfeilspitze endet [41]. Um das Aussehen für eine Komponente festzulegen, gibt es den Begriff `appearance`. Damit können beispielsweise Vorder- sowie Hintergrundfarbe oder auch der Schriftstil einer Komponente festgelegt werden. Solche appearances können vordefiniert und innerhalb eines `nodeStyle` oder `edgeStyle` referenziert werden [41]. Das Festlegen des Erscheinungsbilds von Knoten und Kanten kann nicht nur statisch erfolgen, sondern auch dynamisch zur Laufzeit. Mithilfe von *Appearance Providern* kann das Aussehen beispielsweise abhängig von einem Attributwert verändert werden.

2.5.4 CINCO Product Definition

Sind MGL und MSL erstellt worden, so kann mithilfe der CINCO *Product Definition* (CPD) das CINCO-Produkt generiert werden. Dafür werden die erstellten MGLs in der CPD angegeben und CINCO generiert mithilfe des CINCO Produktgenerators daraus das endgültige CINCO-Produkt [41]. Dieses Vorgehen wurde bereits in Abbildung 2.2 veranschaulicht. Hinzukommen noch einige weitere Features, sodass beispielsweise Abhängigkeiten zwischen ECLIPSE Bundles/Features, Branding-Informationen oder auch ein *splashScreen* definiert werden können.

2.5.5 Validierung

Bis zum jetzigen Zeitpunkt wurden die Spezifikationssprachen MGL, MSL und CPD beschrieben, mit denen ein eigenes CINCO-Produkt spezifiziert werden kann. Damit der Nutzer auf etwaige Unstimmigkeiten bei der Modellierung aufmerksam gemacht werden kann, wird dafür das *Model Compare and Merge Framework* verwendet, welches nun ausführlicher beschrieben wird [43].

Das *Model Compare and Merge Framework* (MCaM) wurde ursprünglich für die Versionskontrolle entwickelt. Dadurch wird es möglich, Änderungen zwischen Versionen desselben Modells zu erkennen und Änderungen von Entwicklern zu einer Version zusammenfassen zu lassen. Anders als andere Versionskontrollsysteme wie GIT oder APACHE SUBVERSION (SVN), können mit MCaM auch die Entitäten eines Modells behandelt werden. Dies bedeutet, dass nicht nur die Zeilen einer Textdatei betrachtet werden, sondern dass beispielsweise bei einem graphbasierten Modell Knoten und Kanten als Entitäten aufgefasst und somit behandelt werden können [43].

Konzept der Validierung

Für die Validierung der statischen Semantik von Modellen in einem CINCO-Produkt wird der Check-Prozess von MCaM verwendet, sodass dieser noch in CINCO eingebunden werden muss. Dazu wird dem `GraphModel` eine `@mcam("check")`-Annotation und eine `@mcam_checkmodule()`-Annotation angehangen. Dabei aktiviert die erste Annotation MCaM und durch den Parameter `"check"` wird die Validierung verwendet. Mithilfe der zweiten Annotation wird die Anmeldung eines handgeschriebenen Checkmoduls, dessen *fully-qualified name* einer JAVA-Klasse als Parameter gesetzt wird, ermöglicht [41]. Letztendlich wird die eigentliche Validierung innerhalb einer `check`-Methode implementiert, welche Fehler-, Warn- und Informationsmeldungen bereitstellt. Diese Meldungen werden dem Nutzer in der *Model Validation View* angezeigt, um ihn auf etwaige Unstimmigkeiten hinzuweisen.

2.5.6 Codegenerierung

Neben der Modellierung und der Validierung ist es außerdem möglich, in CINCO eine Codegenerierung einzubinden. Dabei kann der Nutzer Informationen aus dem erstellten Modell entnehmen und in eine beliebige Sprache ausgeben lassen.

CINCO enthält ein Codegenerator-*Meta-Plugin*, welches eine `@generatable`-Annotation zur Verfügung stellt, die an das `GraphModel` annotiert werden kann [33]. Die Annotation besitzt zwei Parameter, wobei der Erste den *fully-qualified name* der Codegenerator-Klasse und der Zweite den Zielordner für den zu generierenden Code angibt. Der Generator enthält eine `generate`-Methode, die bei dem Generierungsprozess ausgeführt wird.

2.6 DyWA Integrated Modeling Environment

Das DyWA *Integrated Modeling Environment* (kurz Dime) ist ein Cinco-Produkt, welches das *Dynamic Web Application Framework* (kurz DyWA) verwendet[16, 40]. Dime bietet eine Familie von graphischen DSLs, die es ermöglicht eine umfassende Webanwendung mit Prozessen, Nutzeroberflächen und Zugriffskontrolle zu modellieren. Im Folgenden sollen die Grundkonzepte von Dime beschrieben werden, welche die wichtigsten Bausteine zur Erstellung einer kompletten Dime-Applikation darstellen.

2.6.1 Prozesse

Hauptbestandteil der Dime-Funktionalitäten sind die Prozesse. Sie bestehen aus einem Start- und mindestens einem Endknoten. Dazwischen können verschiedene Knotentypen liegen. Diese sind beispielsweise sogenannte SIBs, die wiederum Unterprozesse oder GUI-Modelle enthalten können [16]. Da ein solcher Prozess als ein Kontrollfluss agiert, können durch unterschiedliche Abzweigungen verschiedene Logiken implementiert werden. Um Daten zu verarbeiten, können Knoten außerdem Daten-Input- und Daten-Output-Ports enthalten. Diese wiederum können mit Datenobjekten des *Data Context* verbunden werden. Output-Ports sind über Update-Kanten und Input-Ports über Read-Kanten mit Datenobjekten verbindbar.

2.6.2 Datenstrukturen

Datenmodelle können primitive und komplexe Datentypen enthalten. Primitive sind:

`Text:` Eine Zeichensequenz

`Integer:` Eine Ganzzahl

`Real:` Eine reelle Zahl

`Boolean:` Ein logischer Wert (`true` oder `false`)

`Timestamp:` Ein Zeitpunkt

`File:` Eine Datei (Dateityp, Dateiname und Referenz zu ihrem eigentlichen Inhalt)

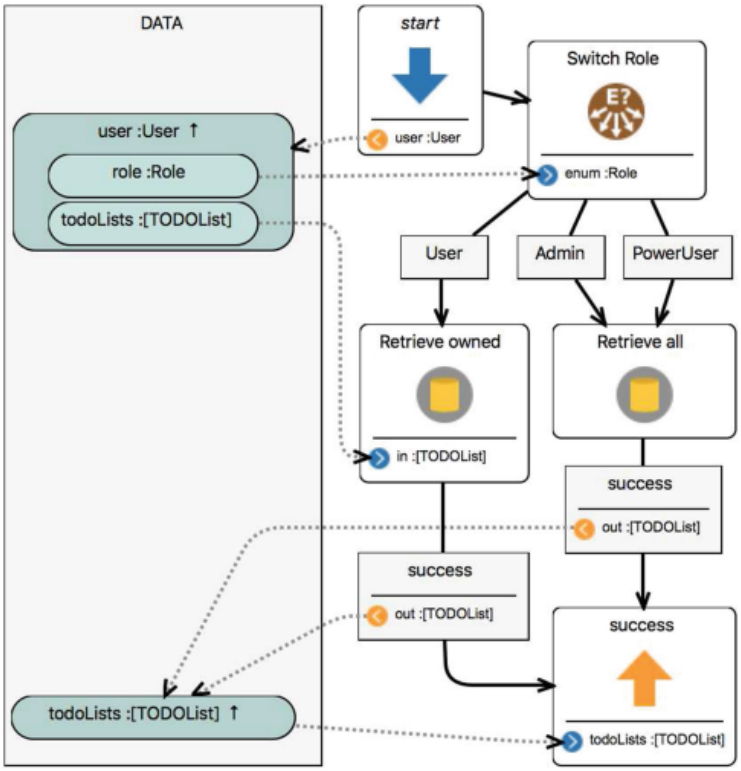


Abbildung 2.3: Beispiel eines DIME-Prozesses [16]

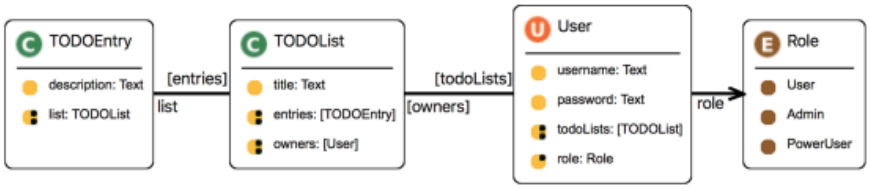


Abbildung 2.4: Beispiel eines DIME-Datenmodells [16]

Komplexe Datentypen werden ebenfalls durch eine GDSL modelliert. Sie können Attribute oder Variablen und Referenzen auf andere komplexe Datentypen enthalten.

Abbildung 2.3 zeigt beispielhaft einen DIME `TODOList` Prozess. Zunächst wird im Startknoten ein User über den Output Port im `Data Context` der User gesetzt. `Switch Role` entscheidet aufgrund des Wertes von `Role`, der anhand der `Role` des zuvor angegeben `Users` festgelegt ist, ob der Prozess `Retrieve owned` oder `Retrieve all` angestoßen wird. Beim Ersten der beiden Prozesse wird die dem Benutzer zugehörige `TODOList` über den Input Port geladen. Danach werden die entsprechenden `TODOLists` in den jeweiligen `Success` Knoten in dem `Data Context` gespeichert. Zum Schluss wird im darauf folgenden `Success` Knoten über den Input Port die Liste für einen nächsten Prozess zur Verfügung gestellt.

In Abbildung 2.4 ist `TODOList` als ein Beispiel für einen komplexen Datentyp aufgeführt. Er hat drei Attribute: `title`, `entries` und `owners`. Dabei ist `title` vom Typ `Text`, `entries` ist eine Liste von `TODOEntiy`-Referenzen und `owners` ist eine Liste von `User`-Referenzen, zu erkennen an den Referenz-Kanten.

2.6.3 GUI

GUI-Modelle sind deklarative Strukturen, welche die User Interfaces darstellen. Diese enthalten *Control Edges* und *Data Bindings*. Auch hier gibt es eine Datenbasis, die mit den GUI-Elementen für ein *Data Binding* verknüpft werden. Ähnlich wie bei den Prozessen werden hier die Daten initial injiziert [16]. Im Gegensatz zu den Prozessen gibt es hier allerdings keinen Start-Knoten [16]. Neben den Zuweisungen von Variablen durch Kanten kann der Benutzer auch sogenannte *template expressions* verwenden. In Abbildung 2.5 wird der Ausdruck `{{currentUser.username}}` verwendet, um auf das *username*-Attribut vom *currentUser* zuzugreifen. Mögliche GUI-Elemente:

`Tables` können benutzt werden, um zum Beispiel Einträge von Listen anzuzeigen. Diese können dann mit Listen-Typen verknüpft werden.

`Forms` kapselt Komponenten, um dem Benutzer eine Manipulation von Variablen zu erlauben. Dabei gibt es zwei Typen: *field components*, um primitive Variablen zu ändern und *selection components*, um aus einer Liste von Optionen auszuwählen.

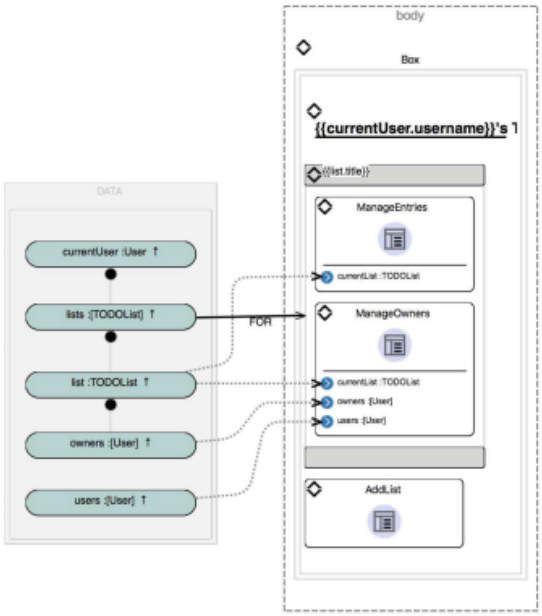


Abbildung 2.5: Beispiel eines DIME-GUI-Modells [16]

2.6.4 Generierung

Wurden sowohl die Prozesse als auch die Datenstrukturen und die GUI modelliert, so kann die Generierung ausgeführt werden. Die Generierung erzeugt die modellierte Webanwendung in Form von Dateien im sogenannten `target`-Ordner. Die im `target`-Ordner generierten Dateien können auf einen für DIME-Anwendungen konfigurierten Webserver deployed und zwei theoretischen Bereichen einer Webanwendung zugeordnet werden. Innerhalb des `target`-Ordners werden zwei Ordner stellvertretend für diese beiden Bereiche des Frontends und des Backends einer Webanwendung automatisch angelegt. Folgende Ordner sind somit nach Generierung vorhanden:

- `dywa-app`, beinhaltet die Backend-Logik
- `webapp`, beinhaltet Frontend-Logik und Designstruktur

In den Ordnern befinden sich die zu den Modellen generierten Dateien, die die Webanwendung darstellen. Sollten Änderungen am Modell vorgenommen werden, so muss die Generierung erneut angestoßen und die Webanwendung neu deployed werden.

2.7 Amazon Alexa APIs und SDKs

Neben CINCO als Meta Tooling Framework sowie den dazugehörigen Technologien, wird auch Wissen über *Amazon Alexa APIs* benötigt. Es werden die Hauptschritte beschrieben, um einen simplen Skill zu entwickeln. Die verwendeten Begriffe sind aus der offiziellen Amazon Alexa-Skill Entwicklungsanleitung übernommen, um möglichst einheitlich und verständlich zu bleiben [13].

2.7.1 Skill Entwicklung

Alexas Fähigkeiten werden als *Skills* bezeichnet, von denen es mehrere Arten gibt. Sie werden anhand ihrer Funktion unterschieden. Soll Alexa beispielsweise *Smart-Home*-Geräte steuern, so muss dazu ein *Smart-Home-Skill* entwickelt werden. Um auf Geräten mit Bildschirm ein Video abzuspielen, wird entsprechend ein *Video-Skill* verwendet.

Damit die Skills möglichst vielseitig eingesetzt werden können, wird hier die Entwicklung sogenannter *Custom Skills* betrachtet. Nur diese können mit Web-Diensten verbunden werden. Custom Skills haben keine vordefinierten Kontrollsequenzen oder Anwendungseinschränkungen. Mit ihnen kann der Entwickler die gesamte Interaktion zwischen Alexa und dem Nutzer bestimmen.

Im Folgenden wird das typische Beispiel eines Quiz-Skills zur Erläuterung verwendet. Alexa stellt mit diesem Skill zufällige Fragen, die der Nutzer beantworten kann. Dabei zählt Alexa die Anzahl der richtigen Antworten und gibt diese am Ende einer vom Nutzer festgelegten Anzahl an Fragen wieder. Der Nutzer kann Alexa auch neue Informationen beibringen.

Ein Skill besitzt bestimmte Bestandteile, welche von Amazon wie folgt beschrieben werden:

Intent: Ein *Intent* ist eine Funktionalität des Skills, die der Nutzer verwenden kann. Im Quiz wären die Intents die Möglichkeiten des Nutzers, ein Quiz zu starten, die Anzahl der Runden festzulegen, Fragen zu beantworten, Alexa neue Informationen für das Quiz bereitzustellen oder das Quiz zu beenden.

Sample Utterances: Bei den *Sample Utterances* handelt es sich um Wörter und Sätze, die der Nutzer verwenden kann, um den Skill zu nutzen. Dabei werden diese einem bestimmten Intent zugeordnet. Diese Zuordnung wird im *Interaction Model* festgelegt. Für das Quiz könnte zum Beispiel „Alexa, starte ein Quiz“ dem Intent, das Quiz zu starten, zugeordnet werden.

Invocation Name: Der *Invocation Name* ist der Name des Skills. Der Nutzer sagt diesen, um den Skill zu starten. Der Quiz-Skill hat beispielsweise den Invocation Name „Quiz“.

Entwicklungsablauf

Nun soll im Detail beschrieben werden, wie ein Custom Skill entwickelt wird. In der offiziellen Dokumentation zur Erstellung von Alexa-Skills wird die Entwicklung eines Custom Skills in die folgenden Schritte aufgeteilt [13].

1. Voice User Interface designen
2. Skill aufsetzen
3. Interaction Model erstellen
4. Code schreiben und testen

Dabei empfiehlt Amazon die Entwicklung eines Skills mit seiner *Amazon Developer Console* [4]. Dies ist eine Webseite über die der Skillentwickler seine Skills verwalten und testen kann. Die wichtigsten Schritte dieses Prozesses sind das Erzeugen eines sogenannten *Interaction Models*, das beschreibt, wie Alexa mit dem Nutzer kommuniziert, und das Schreiben und Bereitstellen des Codes, der die Logik des Skills enthält. Auf diese Punkte wird später noch genauer eingegangen. Zunächst soll jedoch das Interaction Model vorgestellt werden.

2.7.2 Interaction Model

In dem *Interaction Model* ist die Zuordnung von Anfragen des Nutzers zu Intents beschrieben. Diese Zuordnung verbindet die bereits beschriebenen Intents und Sample Utterances mit den *Custom Slot Types*:

***Custom Slot Types*:** Hierbei handelt es sich um eine Liste von möglichen Belegungen der Slots des Intents. Diese werden genutzt, wenn der Slot keinen Standardwert erwartet.

„Alexa, starte das Quiz mit fünf Fragen.“[2] Dies ist ein Beispiel einer möglichen Zuordnung im Interaction Model des Quiz-Skills. Hier besteht der Intent darin, das Quiz zu starten. Dabei füllt „fünf“ den dazu gehörenden Slot. Dieser benötigt keinen Custom Slot Type, da hier natürliche Zahlen erwartet werden. Der Gesamtausdruck ist dabei eine Sample Utterance.

Zusätzlich kann, wenn gewünscht, noch ein sogenanntes *Dialog Model* erzeugt werden. Dieses wird genutzt, um Konversationen zu erzeugen. Für solche Dialoge kann angegeben werden, ob Alexa den Nutzer fragen soll, ob er sich sicher ist, dass der Intent ausgeführt werden soll oder ob der Slot richtig belegt ist. Im Rahmen des Dialogs können ebenfalls die Werte gewünschter Slots erfragt werden, wenn der Nutzer diese nicht angegeben hat, obwohl sie notwendig sind.

Es ist noch zu beachten, dass ein Skill stets bestimmte Standard-Intents beinhalten muss. Diese dienen der allgemeinen Funktion des Skills. Beispiele hierfür sind der Beenden-Intent und der Hilfe-Intent.

JSON-Repräsentation des Interaction Models

Amazon benötigt das Interaction Model in Form einer JSON-Datei. In Listing 2.13 wird eine allgemeine Darstellung des entsprechenden Teils des Interaction Models aufgeführt.

Zunächst beinhaltet diese ein sogenanntes *Language Model*. Sie beginnt mit dem *Invocation Name*, also dem Namen des Skills, auf den Alexa reagieren soll. Darauf folgt die Liste der Intents. Darin wird für einen Intent, der Name, eine Liste von Sample Utterances und eine Liste von Slots angegeben. Eine Sample Utterance ist ein einfacher String, während Slots wiederum einen Namen, einen Typen und mehrere Beispielwerte haben. Bei dem Typen kann es sich um einen vordefinierten Typen von Amazon oder einen selbst definierten handeln. Nach der Liste aller Intents, die auch Amazons Standard-Intents beinhalten muss, folgt eine Liste von *Slot Types*. Diese beinhaltet jeden selbst definierten Slot Type. Dabei besteht solch einer aus einem Namen und einer Liste von Werten.

Nach dem Language Model folgt der *Dialog*. Dieser ist nur erforderlich, wenn ein oder mehrere Intents mit zusätzlichem Dialog, wie einer Bestätigungsanfrage, verbunden sind. Hier müssen nur Intents angegeben werden, für die solch ein Dialog gewünscht ist. Dabei wird der Name des Intents wiederholt und angegeben, ob dieser bestätigt werden muss. Für einen Slot muss wieder der Name und der Typ genannt werden. Zusätzlich muss angegeben werden, ob der Slot bestätigt oder erfragt werden soll, falls der Nutzer keinen Wert angegeben hat. Soll Alexa nach einem Slot-Wert oder einer Bestätigung von Slots oder Intents fragen, folgt nach dem Dialog die Liste der *Prompts*. Darin hat ein Prompt eine eindeutige ID und eine Liste möglicher Variationen. Jede Variation hat einen Typen und einen Wert, der den von Alexa gesagten Text beinhaltet. Nicht jede dieser Angaben ist notwendig. Braucht beispielsweise kein Intent oder Slot einen zusätzlichen Dialog, wird keine Dialog- und keine Prompts-Liste benötigt.

```
{ "interactionModel": {
    "languageModel": {
      "invocationName": "<Name>",
      "intents": [
        { "name": "<Intent Name>",
          "samples": [ <Sample Utterances> ],
          "slots": [
            { "name": "<Slot Name>",
              "type": "<Slot Type Name>",
              "samples": [ <Example Slot Values> ] },
            ...
          ] },
        ...
        ],
        "types": [
          { "name": "<Slot Type Name>",
            "values": [
              { "name": { "value": "<Slot Type Value>" } },
              ...
            ] },
          ...
        ] },
  "dialog": {
      "intents": [
        { "name": "<Zuvor definierter Intent Name>",
          "confirmationRequired": <true\false>,
          "prompts": { "confirmation": "<Prompt ID>" },
          "slots": [
            { "name": "<Slot Name>",
              "type": "<Slot Type Name>",
              "confirmationRequired": <true/false>,
              "elicitationRequired": <true/false>,
              "prompts": {
                "confirmation": "<Prompt ID>",
                "elicitation": "<Prompt ID>" } },
            ...
          ] },
        ...
      ] },
      "prompts": [
        { "id": "<Prompt ID>",
          "variations": [
            { "type": "PlainText",
              "value": "<Text>" },
        ...
      ] } }
```

Listing 2.13: Gesamter Aufbau einer JSON-Datei für ein Interaction Model

Mit solch einer JSON-Datei ist definiert, was der Nutzer sagen kann und welche Informationen er weiter geben möchte. Nun muss Alexa diese Informationen nutzen, um die eigentliche Funktion des Skills umzusetzen. Diese wird mit Hilfe der von Amazon zur Verfügung gestellten *Alexa Skill Development Tools* implementiert – nicht zu verwechseln mit dem von der PG entwickelten Alexa Skill Development Tool.

2.7.3 Alexa Skill Development Tools

Die verschiedenen *Alexa Skill Development Tools* bieten viele Möglichkeiten Skills zu entwickeln. Sie vereinfachen die Entwicklung von Skills für die entsprechenden Programmiersprachen. Es ist jedoch auch möglich mit einer beliebigen Programmiersprache zu arbeiten, auch wenn es für diese kein SDK gibt, das die JSON-Kommunikation vereinfacht.

Die Alexa Skill Development Tools bestehen aus mehreren SDKs in verschiedenen Programmiersprachen und verschiedenen Programmen zum Verwalten von Skills [3]. In Unterabschnitt 2.7.5 wird auf diese genauer eingegangen. Im Folgenden wird das ASK SDK für Java weiter betrachtet [2].

Alexa Skills Kit SDK für Java

Nun soll vorgestellt werden, wie mithilfe des SDKs für Java das Programm hinter einem Skill basierend auf einem Interaction Model entwickelt wird. Für die Entwicklung eines Skills wird Java 8 benötigt. Des Weiteren wird in den offiziellen Tutorials Maven verwendet [2, 14]. Startet der Nutzer den Skill und äußert einen Intent, sendet Alexa eine Anfrage (*Request*) an den Endpoint, auf dem ein Java-Programm läuft, das die ASK SDK verwendet. Um auf solch eine Request zu reagieren, existiert das `RequestHandler`-Interface.

2.7.4 Funktionalität einbinden

Nun muss der vorhandene Endpoint mit Alexa verbunden werden. Wie zuvor erwähnt, muss er dafür online verfügbar sein. Um dies umzusetzen, gibt es zwei Möglichkeiten. Der Code kann zum *Amazon Web Services Lambda* (AWS Lambda) hochgeladen oder als selbst entwickelter Web-Service gehostet werden [8, 9]. Im Folgenden wird nur die letzte Variante näher betrachtet.

Selbst gehostete Web-Services

Um einen Web-Service für einen Skill nutzen zu können, muss dieser bestimmte Anforderungen erfüllen. Der Service muss

- online zugreifbar sein,
- den Regeln des *Alexa Skills Kit Interfaces* folgen,
- HTTPS über SSL/TLS verwenden,
- Anfragen auf Port 443 akzeptieren,
- ein Zertifikat mit alternativen Namen, der dem Domänennamen des Endpoints entspricht, aufweisen und
- überprüfen, ob eingehende Anfragen von Alexa stammen.

Wird ein selbst gehosteter Web-Service verwendet, muss von der abstrakten Klasse `SkillServlet` des ASK SDKs geerbt werden. Diese dient als Zugriffspunkt für Alexa. Das SDK enthält noch weitere Funktionalitäten wie spezielle `ExceptionHandler` oder *Request* und *Response Interceptors*. Doch diese sind für die grundlegende Funktion des Skills nicht notwendig. Deshalb wird auf diese auch nicht weiter eingegangen.

Sowohl ein selbst gehosteter Skill als auch AWS Lambda nutzen das *Alexa Skills Kit Interface*. Es schreibt die Form der Kommunikation zwischen Alexa und dem Programm vor. Daher wird dies als nächstes erläutert.

Alexa Skills Kit Interface

Die Kommunikation zwischen Alexa und dem Programm erfolgt über HTTPS mit SSL/TLS Verschlüsselung. Alexa sendet eine POST Anfrage mit einer JSON-Datei, die alle nötigen Informationen enthält, an den Endpoint. Das Programm erstellt daraufhin eine Antwort, die wiederum in Form einer JSON-Datei verschickt wird. Beide Dateien haben ein bestimmtes Format.

Das Antwortformat sieht nach der Alexa Dokumentation wie folgt aus [10]:

`version`: Die Versionsnummer hat immer den Wert `1.0`.

`sessionAttributes`: Bei den `sessionAttributes` handelt es sich um eine Map mit Informationen, die innerhalb einer Session, also zum Beispiel während einer Quizrunde, gespeichert werden sollen. Diese Informationen entsprechen der `attributes`-Map im `session`-Objekt von Alexas Request. Falls solche Informationen nicht benötigt werden, können die `sessionAttributes` ausgelassen werden.

`response`: Das `response`-Objekt schreibt vor, wie Alexa sich verhalten soll. Es kann eine Antwort an den Nutzer senden oder den Skill beenden. Das Objekt hat die folgenden Bestandteile:

- `outputSpeech`: Das `outputSpeech`-Objekt beinhaltet Alexas zu sprechende Antwort. Darin wird angegeben, ob die Ausgabe als Text oder in *Speech Synthesis Markup Language* (SSML) geschrieben wird (für SSML siehe Unterabschnitt 2.7.6). Optional kann noch ein `playBehaviour`-String angegeben werden. Dieser bestimmt, wann der Text gesprochen werden soll, also ob er an eine Warteliste gehängt werden soll oder andere Sprachanfragen abgebrochen und der Text direkt ausgesprochen werden soll. Standardmäßig wird die Sprachanfrage an die Warteliste gehängt.
- `card`: Das `card`-Objekt beschreibt, was auf Echo-Geräten mit Display oder in Amazons Alexa-App angezeigt werden soll. Dies wird als *Karte* bezeichnet. Dabei wird zwischen verschiedenen Typen unterschieden. Je nach Typ der Karte folgt deren Inhalt.
- `reprompt`: Das `reprompt`-Objekt beschreibt, was passieren soll, wenn der Nutzer eine gewisse Zeit inaktiv ist. Es kann nur in Antworten auf `CanFullfillIntentRequest`s, `LaunchRequest`s, `IntentRequest`s oder `InputHandlerRequest`s verwendet werden. Das Objekt beinhaltet ein `outputSpeech`-Objekt, das denselben Aufbau hat wie das `outputSpeech`-Objekt im `response`-Objekt.
- `directives`: Die `directives` sind ein Array, das Befehle für die verschiedenen Interfaces beinhaltet.
- `shouldEndSession`: Der `shouldEndSession` Wert gibt an, ob Alexa die Session beenden soll, nachdem sie ihren Text gesagt hat.

2.7.5 Alexa Skill Management API

Bisher wurden die einzelnen Schritte für die Generierung eines Alexa-Skills vorgestellt. Um diese Teile miteinander zu verbinden, bietet Amazon ein Online-Werkzeug auf ihrer Webseite. Mit diesem werden die Skills verwaltet, erzeugt, getestet und veröffentlicht. Neben dieser Website bietet Amazon auch die sogenannte *Alexa Skill Management API* (SMAPI) an [7]. SMAPI bietet ein *RESTful HTTP Interface*. Dieses ermöglicht es, Alexa-Skills mit eigenen Werkzeugen zu verwalten. Auch das *Alexa Skills Kit Command Line Interface* (ASK CLI, siehe Abschnitt 2.8) basiert auf dieser API. Dabei handelt es sich um ein Werkzeug, mit dem die Skills über die Konsole verwaltet werden können. Um SMAPI in einem eigenen Werkzeug zu nutzen, muss OAuth 2.0 für den Login zum Amazon Developer Console Account verwendet werden [35].

Um einen Skill zu verwalten, werden folgende Informationen über diesen Skill benötigt:

- Skill Manifest
- Interaction Model
- Account Linking Information

Das Interaction Model wurde bereits in Abschnitt 2.7.2 vorgestellt. Daher genügt es, im Folgenden die beiden anderen Bestandteile zu erläutern.

Skill Manifest

Das *Skill Manifest* beinhaltet notwendige Informationen zur Verwaltung und Veröffentlichung des Skills in Form einer JSON-Datei [1]. Mit dem Skill Manifest werden Alexa alle nötigen Metadaten übergeben. Das Manifest besteht aus mehreren Objekten.

`publishingInformation`: Dieses Objekt beinhaltet Informationen, die bestimmen, wie der Skill dem Nutzer im Store und in der Alexa-App präsentiert wird.

`apis`: Dieses Objekt beinhaltet die Informationen für die vom Skill genutzten Interfaces.

`manifestVersion`: Hierbei handelt es sich um die Versionsnummer des Skill Manifests.

`permissions`: Dieses Array beinhaltet die Rechte den Nutzer nach speziellen Informationen zu fragen.

`privacyAndCompliance`: Dieses Objekt beinhaltet Informationen über den Datenschutz.

`events`: Dieses Objekt bestimmt welche Ereignisse dem Skill mitgeteilt werden sollen. Dies könnte beispielsweise das Starten oder Beenden des Skills sein.

2.7.6 Speech Synthesis Markup Language

Möchte man nicht nur die Standardaussprache von Alexa nutzen, sondern beispielsweise die Stimme von Alexa variieren oder die Aussprache ändern, so hilft die *Speech Synthesis Markup Language* (SSML). Diese beschreibt eine Vielzahl von Optionen, die Sprachausgabe von Alexa-Geräten an den Alexa-Nutzer zu variieren.

Folgende beispielhafte Eigenschaften können im Rahmen von SSML verwenden werden:

`language`: Durch die `language`-Option kann die Aussprache von gesprochenen Texten von Alexa-Geräten auf den Akzent einer Sprache gesetzt werden. Somit kann ein deutscher Text auch mit einem französischen Akzent gesprochen werden.

`voice`: Alexa-Geräte können Response-Texte auch in anderen Stimmen sprechen als nur der Standardstimme. So ist es beispielsweise über die `voice`-Option möglich, Alexa-Geräte eine männliche Stimme für die Ausgabe der Response-Texte nutzen zu lassen. Hierbei existiert eine Vielzahl von möglichen Stimmen.

`whisper`: `whisper` ist die Möglichkeit Texte von Alexa-Geräten leise flüstern zu lassen. Hierbei ist eine Kombination mit beispielsweise der `language`-Option möglich, so dass die Standard-Alexa-Stimme ersetzt werden kann durch eine französische flüsternde Stimme.

`pause`: Die `pause`-Option bekommt eine Zeitangabe in Millisekunden übergeben. Bei Nutzung der `pause`-Option pausiert die Alexa-Stimme zwischen den Sätzen einer Response für den angegebenen Zeitraum. Nach der Pause wird die nächste spezifizierte Aktion ausgeführt.

`emphasis`: Die `emphasis`-Option gibt an, mit welcher Betonung die Alexa-Stimme die Response-Texte vorlesen soll. Hierbei wird zwischen drei unterschiedlichen Stufen der Betonung unterschieden.

2.7.7 Weitere Funktionalitäten

Damit wurden die wichtigsten Aspekte der Erstellung eines Alexa-Skills in Kurzform vorgestellt. Doch Alexa hat noch weitere Funktionalitäten, die hier nicht vorgestellt wurden, aber im Folgenden kurz aufgezählt werden. Diese sollen einen Eindruck über Alexas Funktionalität sowie mögliche Erweiterungen der entwickelten DSL bieten.

Alexa kann sogenannte Interfaces verwenden, die der Skill nutzen kann [10]. Bei diesen handelt es sich um Funktionalitäten der Echo-Geräte, die genutzt werden können. Das sind beispielsweise der `AudioPlayer`, die `VideoApp` oder der `GadgetController`.

Alexa hat einige Ausdrücke, die besonders ausgesprochen werden können. Diese werden als *Speechcons* bezeichnet. Sie sind in mehreren Sprachen verfügbar, darunter Englisch und Deutsch, und können auf der Webseite angehört werden [12]. Um sie einzubinden, muss Alexas Antworttext im SSML-Format angegeben werden. Dann kann der SSML-Tag `<say-as interpret-as="interjection">` verwendet werden. Zusätzlich dazu ist es mit SSML möglich, auf eine Bibliothek verschiedener Töne zuzugreifen. Dabei handelt es sich beispielsweise um Tier-, Natur- oder SciFi-Töne.

Es ist auch möglich Zahlungsmöglichkeiten in einen Skill einzubinden [1]. Dies kann in Form eines Abonnements oder einer einmaligen Zahlung erfolgen. Amazon erhält einen Anteil von 30 % an allem, was über den Skill gekauft wird.

2.8 Alexa Skills Kit Command Line Interface

Dieses Kapitel befasst sich mit den Möglichkeiten, die es zum Erstellen und Hochladen von Alexa-Skills gibt. Zunächst wird die *Skill Management API* und im weiteren Verlauf das *Alexa Skills Kit Command Line Interface* vorgestellt.

Die meistgenutzte Möglichkeit zum Verwalten eines Skills ist die *Amazon Developer Console*. Diese bietet eine GUI über den Browser an, um die eigenen Skills zu administrieren und zu testen. In CINCO kann dieser Ablauf integriert werden ohne die Amazon Developer Console nutzen zu müssen. Die benötigten Voraussetzungen werden im Folgenden beschrieben.

Eine Möglichkeit ist die *Skill Management API*, kurz SMAPI, von Amazon. Diese bietet ein *RESTful HTTP Interface* an, wodurch Anfragen zur Skill-Verwaltung an den Amazon-Server gesendet werden können. SMAPI verwendet *Login with Amazon*, einen weiteren von Amazon gestellten Service. Dieser dient der Nutzerauthentifizierung und ermöglicht, die erstellten Skills einem Amazon-Konto zuzuweisen. Doch *Login with Amazon* ist nur für Webseiten oder Apps auf Android, iOS oder Amazon-Geräten gedacht. Um also SMAPI auf einem PC zu verwenden, muss dies über einen Browser erfolgen, der eine Login-Webseite öffnet, auf der der Nutzer sich einloggen muss. Hat der Nutzer dies getan, erhält er von *Login with Amazon* ein Authentifizierungs-Token, das den *HTTPS Requests* angehangen werden muss. Diese Token sind nur für einige Minuten gültig und müssen dementsprechend regelmäßig durch Requests erneuert werden. Um dieses Vorgehen in CINCO zu integrieren, ist es also nötig, ein Browser-Fenster zu öffnen, den Nutzer sich anmelden zu lassen und alle fünf Minuten einen neuen HTTPS Request zu senden.

Eine Alternative zu diesem aufwendigen Verfahren mittels SMAPI bietet das *Alexa Skills Kit Command Line Interface* (ASK CLI). ASK CLI ist ein eigenständiges vollständiges Programm zur Skill-Verwaltung. Wie der Name sagt, wird es über die Kommandozeile bedient. Dies ermöglicht die Integration in JAVA-Code. Im Gegensatz zu SMAPI genügt es für ASK CLI, einmalig die Daten des gewünschten Amazon-Accounts anzugeben. Durch die Anmeldung wird ein `auth-Token` erstellt und ASK CLI verwendet dieses, wenn nötig, zur Authentifizierung. Da hier also nur eine einmalige Aktion vom Nutzer gefordert ist, wurde sich in diesem Projekt für ASK CLI entschieden.

2.8.1 Installation von ASK CLI

Vor der Installation von ASK CLI muss Git und Node.js 4.5 installiert sein. Um ASK CLI zu verwenden, muss der Nutzer einen *Amazon Developer Account* haben. Dieser wird unter anderem dazu verwendet den entwickelten Skill zu testen und zu veröffentlichen.

Der Nutzer muss selbst ASK CLI installieren und einrichten, damit dieses genutzt werden kann. Über den *Node Package Manager* kann das ASK CLI installiert werden. Hat der Nutzer dies getan, muss er ASK CLI noch über den `init`-Befehl initialisieren. Dabei wird nach dem Amazon Developer Account und einem *Amazon Web Service Account* (AWS-Account) gefragt. Der AWS-Account muss allerdings nicht angegeben werden, da ein eigener Endpoint aufgebaut wird.

2.8.2 ASK CLI Projekte

ASK CLI verwaltet für jeden Skill ein Projekt. Dieses wird als Ordner auf dem System angelegt. Ohne den entsprechenden Ordner mit der passenden Struktur und den richtigen Dateien kann der Skill nicht verwaltet werden. Die Ordnerstruktur ist fest vorgegeben. Sie muss jedoch nicht vom Entwickler selbst erstellt werden, da ASK CLI eine Funktion dafür besitzt. Der Projektordner besteht aus folgenden Bestandteilen:

`hooks`: In diesem Ordner befinden sich Skripte, die nach Ausführen eines ASK-CLI-Befehls gestartet werden. Diese müssen nicht verändert werden.

`lambda`: Dieser Ordner enthält den Quellcode für den entsprechenden *AWS Lambda Skill*. Er ist für das Projekt nicht relevant, da ein eigener Endpoint verwendet wird.

`models`: Anders als die Vorherigen muss dieser Ordner bearbeitet werden. In ihm befinden sich die Interaction Models des Skills. Hier muss für jede Region, in der veröffentlicht werden soll, eine JSON-Datei mit dem Namen der Region und dem entsprechenden Interaction Model vorhanden sein. Für Deutschland wäre dies eine Datei mit dem Namen `de-DE.json`. In diese Datei muss also das durch die Intents-DSL erzeugte Interaction Model eingefügt werden.

`README` und `LICENSE`: Auch diese Dateien müssen nicht bearbeitet werden.

`skill.json`: Hierbei handelt es sich um das sogenannte *Skill Manifest*. Dieses beinhaltet benötigte Informationen für die Veröffentlichung des Skills, wie zum Beispiel eine Skill-Beschreibung. In ihr werden auch die unterstützten Regionen so wie die Endpoints festgelegt. Für das Skill Manifest wird eine sogenannte Manifest-DSL verwendet. Diese wird in Abschnitt 3.4 genauer beschrieben.

2.8.3 Funktionalität

Zur Verwaltung eines Skills gehören viele Funktionen. ASK CLI beherrscht alle von Amazon unterstützen Arten, einen Skill zu verwalten, da durch ASK CLI auch reine SMAPI-Anfragen möglich sind. Doch ASK CLI selbst hat die wichtigsten Funktionalitäten deutlich vereinfacht.

Projekt anlegen: Um ein Projekt anzulegen, genügt es den `init`-Befehl „`ask init`“ aufzurufen. Dieser legt einen Projektordner in dem Verzeichnis an, in dem man sich befindet. Dabei muss ein Name sowie eine Projektart angegeben werden. Auch eine Vorlage muss gewählt werden.

Skill deployen: Um einen Skill auf dem Amazon-Account zu erzeugen oder zu updaten, gibt es den `deploy`-Befehl „`ask deploy`“. Dies ist notwendig, um den Skill zu testen oder zu veröffentlichen. Dabei wird der Skill auch auf Konsistenz mit dem Skill auf dem Amazon-Account überprüft. Auch wenn dies nicht übereinstimmt, kann das Update erzwungen werden.

Skill testen: Es ist ebenfalls möglich, einen Skill auf dem Amazon-Account mit dem `dialog`-Befehl „`ask dialog`“ zu testen. Durch diesen entsteht ein Dialog mit Alexa in der Konsole.

Skill updaten: Ein vorhandener Skill kann mithilfe des „`ask api update-skill`“ und den Parametern „`-s`“ mit der `Skill-Id` und „`-f`“ mit dem *Skill Manifest* aktualisiert werden. Dies ist allerdings nur möglich, wenn ein vorhandener Skill mit der `Skill-Id` existiert und das *Skill Manifest* gültig ist.

Weitere Funktionalitäten: Weitere Befehle sind zum Beispiel das Klonen von Projekten. Dies kann hilfreich sein, um zu überprüfen, ob der Skill bereit ist, veröffentlicht zu werden.

Kapitel 3

Modellierung

Im Folgenden wird beschrieben, wie die Modellierung eines Alexa-Skills mit dem Alexa Skill Development Tool (ASDT) erfolgt und aus welchen Bestandteilen die Modellierung besteht. Im Zuge dessen werden die einzelnen Modellierungsbestandteile und das Zusammenspiel selbiger erläutert.

Ein Alexa-Skill und auch die Alexa-Skill-Modellierung haben dabei zwei Hauptbestandteile. Diese beschreiben zum einen den Ablauf des Skills, der auf einem eigenen Server laufen soll, und zum anderen die von Amazon benötigten Informationen über das Interaction Model des Skills. Beide Bestandteile sollen durch DSLs modelliert werden.

Um den Skill-Ablauf darzustellen, muss es möglich sein, verschiedene Entscheidungen und Zustände zu modellieren. Dabei sollte leicht erkennbar sein, ob es sich um einen Intent des Nutzers oder um eine Antwort des Skills handelt. Dafür wurde eine graphische DSL (Skill-DSL) zur Modellierung des Skill-Ablaufs entwickelt. Diese ermöglicht eine intuitive Modellierung von Abläufen, Entscheidungen und Beziehungen. Es kann leicht dargestellt werden, was der Nutzer sagt und wie Alexa darauf in welcher Situation reagiert. Aus der Skill-DSL soll ein Programm generiert werden, das über einen gewünschten Zugangspunkt erreichbar sein soll und Amazons Anforderungen erfüllt, um als Skill zu funktionieren.

Wie in den Grundlagen bereits erläutert, werden dem Alexa-Service die nötigen Informationen über ein sogenanntes Interaction Model übertragen (siehe Unterabschnitt 2.7.2). Dabei handelt es sich um eine JSON-Datei, die beschreibt aus welchen Intents der Skill besteht und durch welche Begriffe ein Nutzer diese aufrufen kann. Diese Datei ist allerdings sehr unübersichtlich, da sich viele Informationen wiederholen und zusammenhängende Informationen durch die Datei verteilt sind. Deshalb ist es nötig auch das Interaction Model durch die DSL zu modellieren, so dass eine JSON-Datei daraus generiert werden kann.

Dafür eignet sich eine graphische DSL nicht. Hier müssen keine Abläufe oder Beziehungen beschrieben werden, sondern nur Informationen. Demnach bietet sich hier eine textuelle DSL (*Intents-DSL*) zur Modellierung an. Diese liefert eine simple Grammatik zur Repräsentation des Interaction Models.

Ein weiterer Bestandteil des Skills, den der Skill-Entwickler anpassen muss, sind nach der Codegenerierung die Implementierungsklassen. Die Implementierungsklassen stellen die Geschäftslogik hinter den Intents dar, falls kein DIME-Prozess verwendet wird.

Amazon erwartet bei Veröffentlichung eines Skills zugehörige spezifische Informationen. Mit der im ASDT enthaltenen *Manifest-DSL* können diese definiert und beim Veröffentlichen eines Skills mitgeliefert werden. Die mit der Manifest-DSL festgehaltenen Informationen werden verwendet, um eine JSON-Datei zu generieren, die bei der Veröffentlichung notwendige Skill-Informationen bereitstellt.

Insgesamt ergeben sich so vier Themengebiete, die in den folgenden Kapiteln genauer beschrieben werden. Dabei wird als ein einheitliches Beispiel ein Quiz-Skill verwendet. In diesem stellt Alexa dem Nutzer nacheinander Fragen, die dieser richtig beantworten soll. Nach der Antwort des Nutzers wird diese überprüft und der Nutzer gefragt, ob er fortfahren möchte. Möchte er das Quiz beenden, wird dem Nutzer sein aktueller Punktestand genannt, den er auf Wunsch in der Highscore-Liste eintragen lassen kann.

Integration in DIME-App oder alleinstehendes Skill-Projekt

Es existieren grundsätzlich zwei Möglichkeiten ein *Alexa Skill Workflow* (ASW) Projekt zu entwickeln. Die erste Möglichkeit ist die Integration des ASW-Projekts per Projekt-Wizard in eine bestehende DIME-App. So wäre es denkbar, dass der Entwickler mit DIME eine Webseite aufbaut, auf der ein Quiz administriert und über die Alexa-Geräte gespielt werden kann. Dieser Ansatz, der Integration von dem ASW-Projekt in eine bestehende DIME-App erweitert die Möglichkeiten der graphischen Modellierung um die DIME-Prozesse und DIME-Daten.

Die Alternative zu dem Ansatz der Integration des ASW-Projekts in eine DIME-App ist ein alleinstehendes ASW-Projekt. Mit einem alleinstehenden ASW-Projekt ohne Integration in eine DIME-App ist es möglich, Alexa-Skills zu entwickeln, welche keine Webseite, eine persistente Datenbasis oder DIME-Prozesse benötigen. Auch hier steht dem Benutzer zum Erstellen eines solchen alleinstehenden Projekts ein Projekt-Wizard zur Verfügung, um leicht eine ASW-Projektstruktur mit vorgefertigten Beispielen erstellen zu können (siehe Abschnitt 3.1).

3.1 Projekt-Wizard

Unabhängig davon, ob eine DIME-Integration stattfinden soll, kann der Skill-Entwickler auf einen Wizard zugreifen. Mit dem Wizard können einerseits im generierten CINCO-Produkt vollständig neue ASW-Projekte angelegt werden, um den Alexa-Skill mit der Skill-DSL und der Intents-DSL zu programmieren oder, andererseits bei Integration in eine DIME-Infrastruktur nur die benötigten Projektdateien in die DIME-App generiert werden ohne ein komplett neues Projekt anzulegen. Der Wizard fragt nach dem gewünschten Projektnamen und generiert daraufhin folgende Dateien:

Intents-Datei: Eine Datei mit dem Suffix `.intents`, in der die verwendbaren Intents definiert werden

ASW-Datei: Eine Datei mit dem Suffix `.asw`, in der das Graphmodell definiert wird

Skill-Manifest-Datei: Eine Datei mit dem Suffix `.skillmanifest`, in der Daten für die Generierung der JSON-bezogenen Manifest-Datei eingetragen werden können

Generell besteht auch die Möglichkeit, ein Projekt manuell zu erstellen. Hierfür müssen alle notwendigen Dateien manuell erzeugt und benannt werden. Der gleiche manuelle Weg kann auch bei Einbettung eines Alexa-Skills in eine DIME-App gegangen werden.

Um den Wizard aufzurufen, muss auf den Projekt-Browser des generierten Produktes oder die DIME-Infrastruktur, in der das ASW-Projekt eingebunden werden soll, ein Rechtsklick ausgeführt und der Eintrag „New Alexa Skill Workflow Project“ ausgewählt werden (siehe Abbildung 3.1).

Nachdem der Eintrag „New Alexa Skill Workflow Project“ ausgewählt wurde, öffnet sich ein Fenster zum Festlegen des Projektnamens (siehe Abbildung 3.2). Wenn ein Projektname eingegeben und bestätigt wurde, werden alle für das Projekt benötigten Dateien automatisch angelegt. Es öffnet sich die generierte ASW-Datei automatisch im Editor und der Benutzer kann mit der Modellierung des Skills beginnen.

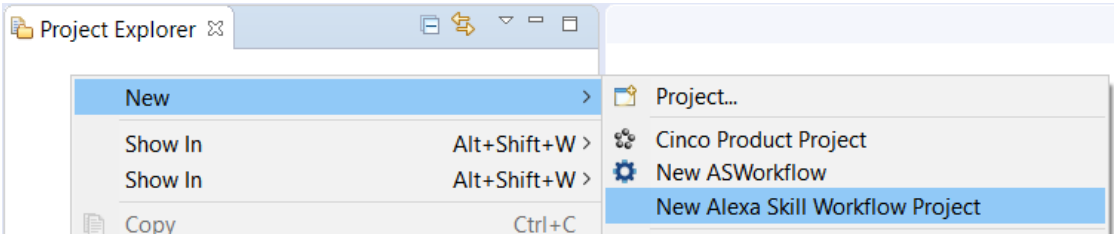


Abbildung 3.1: Wizard-Eintrag im Kontextmenü des Projekt-Browsers

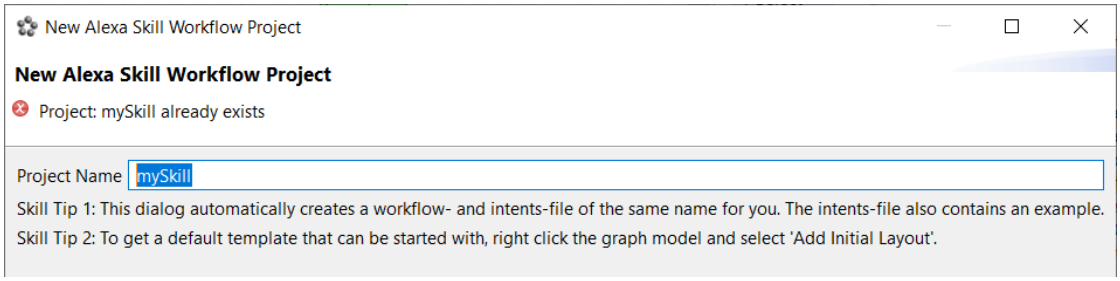


Abbildung 3.2: Aufbau des Wizard-Dialogs

3.2 Graphische DSL für den Skill-Ablauf

Wie bereits erwähnt, wurde sich für den Skill-Ablauf für eine graphische Skill-DSL entschieden. Durch diese Darstellungsweise kann der Skill-Entwickler leicht die verschiedenen Pfade in dem entwickelten Skill nachvollziehen. Insgesamt liegt der Fokus der graphischen Modellierung auf der Einfachheit und Übersichtlichkeit, sodass die Bedienung möglichst intuitiv und benutzerfreundlich ist.

Nach der graphischen Modellierung des Skills sowie dem Erstellen der nötigen Intents in der Intents-DSL erfolgt im nächsten Schritt die Codegenerierung. Bei der Entwicklung eines Alexa-Skills wird mehrfach derselbe Code benötigt. Damit der Skill-Entwickler nicht immer wieder denselben Code schreiben muss, wird dies durch Codegenerierung vereinfacht.

Der Skill-Entwickler hat grundsätzlich zwei Möglichkeiten die Geschäftslogik für einen Intent zu implementieren. Entweder er benutzt einen bereits vorhandenen DIME-Prozess oder er kann vorgefertigte JAVA-Implementierungs-Templates individuell anpassen.

Im Folgenden erfolgt die detaillierte Beschreibung aller Modellelemente der graphischen Modellierungssprache. Abschließend werden ausführlich die einzelnen Validierungsfälle erläutert, durch die die Erstellung von validen Modellen gewährleistet ist.

3.2.1 Modellierungssprache

Wie bereits beschrieben, wird für die weitere Beschreibung der graphischen Modellierungssprache als Beispiel-Skill ein Quiz betrachtet. Für den Überblick ist in der Abbildung 3.3 ein Ausschnitt der graphischen Modellierung des Quiz-Skills dargestellt.

In dem Alexa-Skill-Workflow gibt es verschiedene Zustände, die unterschiedliche Funktionen repräsentieren. Die Zustände an denen Alexa beziehungsweise die dahinter stehende Implementierung entscheidet, wie es im Ablauf weitergeht, werden dabei in Blau dargestellt. Alle grünen Zustände im Ablauf stehen für den Skill-Nutzer, sodass dieser an diesen

Zuständen den Ablauf beeinflussen kann, indem er beispielsweise verfügbare Intents auswählt. Ansonsten gibt es noch graue Zustände, wie den Start- und End-Knoten, welche den Start beziehungsweise das Ende des Skills symbolisieren. Während eines Skills können Daten verwendet und im SessionCache abgespeichert werden, um an beliebiger Stelle, erneut verwendet werden zu können. Auf die einzelnen Modellelemente, die bereits in der Abbildung 3.3 zu sehen sind, wird im weiteren Verlauf in detaillierter Form eingegangen.

ASWorkflow

Zu Beginn steht dem Skill-Entwickler ein leerer Canvas mit einer Palette zur Verfügung. Der Canvas selbst ist dabei ein von der Projektgruppe entwickelter *ASWorkflow*, der verschiedene Modellelemente aus der Palette, aus der Intents-DSL oder aus DIME, wie DIME-Prozesse und DIME-Daten, aufnehmen kann. In dem Editor kann der Skill-Entwickler nun einen Skill als eine Art Flussdiagramm mithilfe der Modellelemente modellieren.

Als kleine Hilfestellung wird dem Skill-Entwickler ein initiales Layout vorgegeben, wenn dieser einen Rechtsklick auf den Canvas ausübt und in dem Kontextmenü „Add initial Layout“ auswählt. Dieses Layout ist in Abbildung 3.4 dargestellt. Es wird ein *SessionCache*, ein Start-Knoten, ein Server-Knoten und ein Response-Knoten initial angelegt.

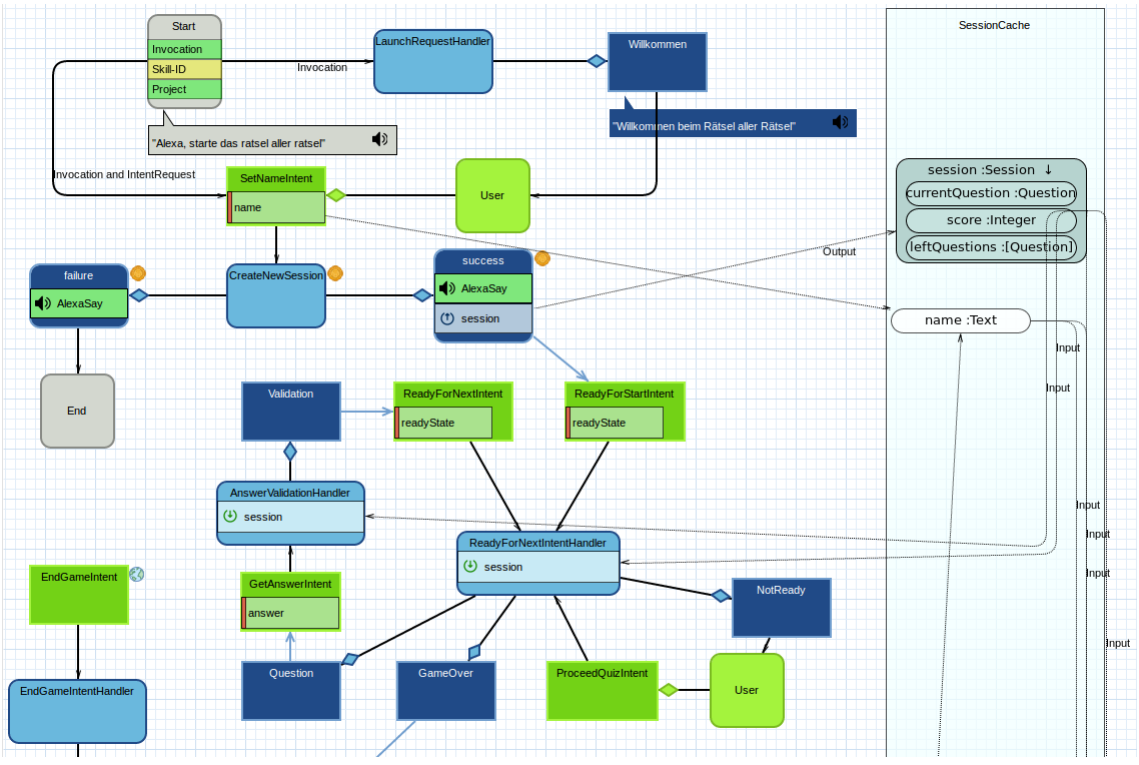


Abbildung 3.3: Ausschnitt des modellierten Quiz-Skills

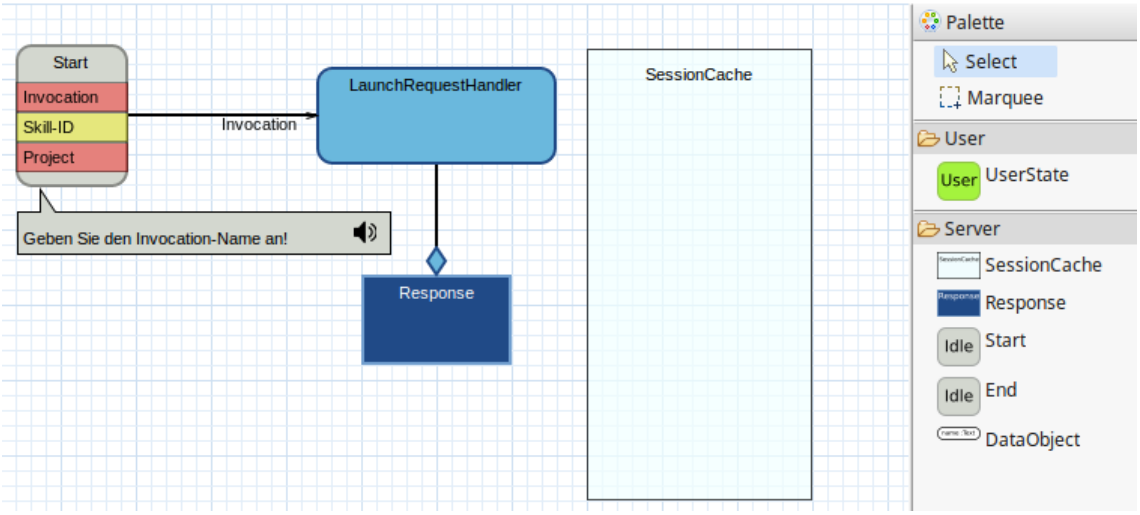


Abbildung 3.4: Initiales Layout

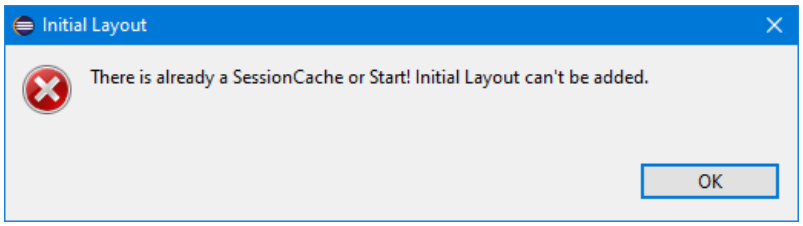


Abbildung 3.5: Fehlermeldung beim Ausüben des „Add initial Layout“-Befehls

Sollte ein SessionCache oder ein Start-Knoten bereits erstellt worden sein, wird eine Fehlermeldung beim Ausüben des „Add initial Layout“-Befehls ausgegeben, wie es in Abbildung 3.5 zu sehen ist. Einige weitere Funktionalitäten, die über das Kontextmenü erreichbar sind, werden im späteren Verlauf beschrieben.

Start

Ein ASWorkflow kann genau einen Start-Knoten besitzen und dieser kennzeichnet den Start des Skills. Wie bereits in dem initialen Layout in Abbildung 3.4 und 3.6 (siehe ①) zu sehen, verfügt der Start-Knoten über ein visuelles Ampelsystem. Wenn eine Angabe zwingend notwendig ist und noch nicht von dem Skill-Entwickler getätigt wurde, wird dies warnend in Rot dargestellt. Ist eine Angabe optional und fehlt, wird dies durch die Farbe Gelb signalisiert. Eine grüne Farbe visualisiert, dass der Skill-Entwickler alle Angaben getätigt hat. Folgende Angaben können beziehungsweise müssen im Start-Knoten getätigt werden:

Invocation Name: Zwingend notwendig ist die Angabe eines *Invocation Name*, damit der Skill gestartet werden kann. Durch eine Sprechblase wird entsprechend visualisiert, wie der Alexa-Nutzer seinen Skill starten kann. Somit wird, wie in Abbildung 3.3 zu sehen, das Quiz durch „Alexa, starte das ratsel aller ratsel“ gestartet.

Skill-ID: Die *Skill-ID* kann von dem Skill-Entwickler als erforderlich selektiert werden. Dadurch kann nur dieser Skill Anfragen an der Server stellen. Alle anderen Anfragen kommen zwar auch am Server an, aber werden jedoch von dem Servlet ignoriert, da diese die falsche Skill-ID besitzen. Die Skill-ID wird hierbei automatisch aus der Konfigurationsdatei des ASK CLI ausgelesen, siehe Unterabschnitt 5.3.2.

Project: Wird ein alleinstehender ASW-Skill entwickelt ohne Integration in eine DIME-App, so ist die Angabe eines Projektnamen zwingend notwendig. Bei der Generierung wird ein neues MAVEN-Projekt abhängig von dem anzugebenden Projektnamen angelegt. Nähere Informationen sind in dem entsprechenden Unterabschnitt 4.2.2 zu finden.

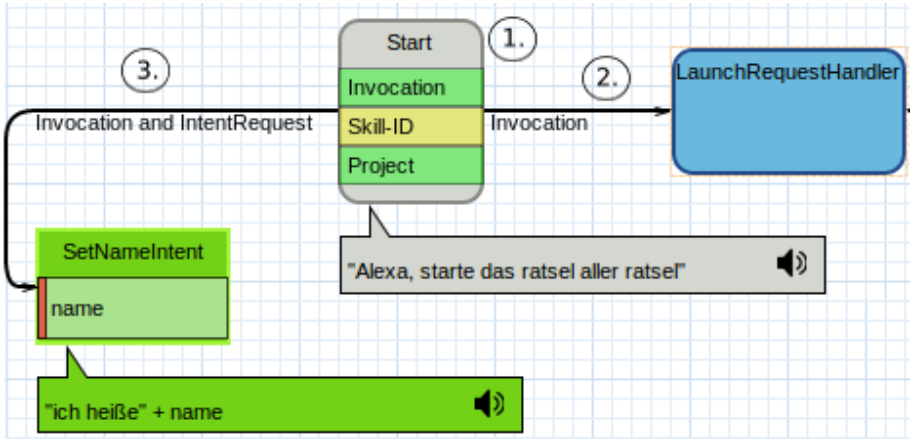


Abbildung 3.6: Start-Knoten

Ausgehend von dem Start-Knoten lassen sich zwei Pfade modellieren, welche nun einzeln beschrieben werden.

Invocation-Pfad: Der Skill kann mit dem Invocation Name, beispielsweise „Alexa, starte das ratsel aller ratsel“, gestartet werden, wie es auch in der Sprechblase dargestellt ist. Wird dieser Weg von dem Alexa-Nutzer ausgewählt (siehe (2.) in Abbildung 3.6), folgt daraufhin automatisch ein sogenannter *Launch Request*, hier dargestellt als ein Server-Knoten. Dieser Launch Request kann dabei nicht übersprungen werden.

Invocation- und Intent-Request-Pfad: Eine weitere Möglichkeit einen Skill zu starten, ist in der Abbildung 3.6 bei (3.) zu sehen. Demnach ist es möglich, beim Starten eines Skills direkt den ersten Intent anzusprechen. In diesem Fall sagt der Alexa-Nutzer neben dem Invocation Name auch direkt die Keyphrase, um den jeweiligen Intent zu verwenden. Für das obige Beispiel würde der Alexa-Nutzer „Alexa, starte das ratsel aller ratsel, ich heiße Sebastian“ sagen, um so den Skill an sich und den *SetNameIntent* zu starten. Dieser Pfad kann in dem Modell beliebig oft modelliert werden.

Intent

Intent-Knoten können entweder direkt von einem Start-Knoten erreicht werden (siehe (3.) in Abbildung 3.6) oder folgen auf einen User-Knoten (siehe Abbildung 3.7 bei (1.)). Die Intent-Knoten basieren auf der Intents-DSL, sodass diese aus der Intents-Datei auf den Canvas gezogen werden können. Intents können Slots besitzen, welche graphisch in einem hellgrünen Rechteck dargestellt werden, so wie der Slot `name` in Abbildung 3.7. Falls ein Slot als *mandatory* gekennzeichnet ist, wird dies durch eine rote Markierung an der linken Seite dargestellt. Für den Fall, dass die Slots in der Intents-Datei modifiziert werden, gibt es die Möglichkeit diese Änderungen im graphischen Modell zu übernehmen. Dazu kann der Skill-Entwickler über einen Rechtsklick auf das ASWorkflow in einem Kontextmenü „Update Intent Slots“ auswählen. Durch das Ergänzen und Entfernen von Slots wird die Größe des Intent-Knotens entsprechend angepasst. Zusätzlich kann der Skill-Entwickler mithilfe eines Doppelklicks auf den Intent die dazugehörige Intents-Datei öffnen. Auf einen Intent folgt immer genau ein dazugehöriger Server-Knoten oder DIME-Prozessknoten.

Die Keyphrases, mit denen ein Intent aufgerufen werden kann, können mithilfe einer Sprechblase angezeigt werden. Dazu gibt es in der Properties View eine Combobox, die alle möglichen Keyphrases beinhaltet, sodass der Skill-Entwickler eine beliebige Auswahl treffen kann.

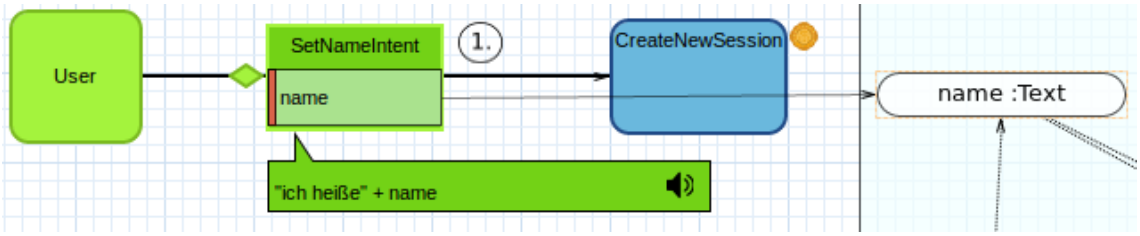


Abbildung 3.7: Intent-Knoten

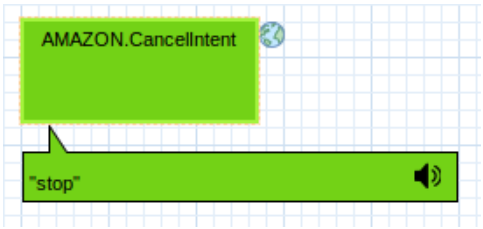


Abbildung 3.8: Globaler Intent-Knoten

Neben den oben vorgestellten Intents sind auch globale Intents verfügbar. Globale Intents besitzen keine eingehende Kante und sind von überall aus ansprechbar. Zur Visualisierung werden globale Intents mit einer kleinen Weltkugel kenntlich gemacht. Amazon selbst stellt einige Intents bereits zur Verfügung. Dazu gehören beispielsweise der in Abbildung 3.8 dargestellte *CancelIntent* oder ein *HelpIntent*.

Intent-Handler

Prinzipiell existieren zwei Möglichkeiten, was auf einen Intent in der Skill-DSL folgen kann. Einerseits kann nach einem Intent ein Server-Knoten folgen oder, andererseits ein DIME-Prozess.

Server-Knoten In den vorherigen Abschnitten wurde bereits erwähnt, dass auf jeden Intent ein Server-Knoten folgen kann. Jedoch können auch mehrere Intents als Nachfolger denselben Server-Knoten besitzen. Der Server-Knoten besitzt dabei einen Namen, der jederzeit vom Benutzer auf dem Knoten editiert werden kann. Dabei muss der Name eindeutig gewählt sein. Auf die Funktionalität eines Handlers wird in dem Unterabschnitt 4.2.2 in detaillierter Form eingegangen. Ein Server-Knoten kann auf spezifische Datenobjekte zugreifen, wenn diese wie in Abbildung 3.9 bei (1.) mit einer Kante referenziert werden. Dabei kann der Skill-Entwickler durch ein- und ausgehende Kanten Datenobjekte aus dem SessionCache benutzen, um so Input, Output oder Update des jeweiligen Datenobjektes zu realisieren. Dies wird jeweils durch ein entsprechendes Icon visualisiert. Wie auch bei den Intents wird die Größe des Servers

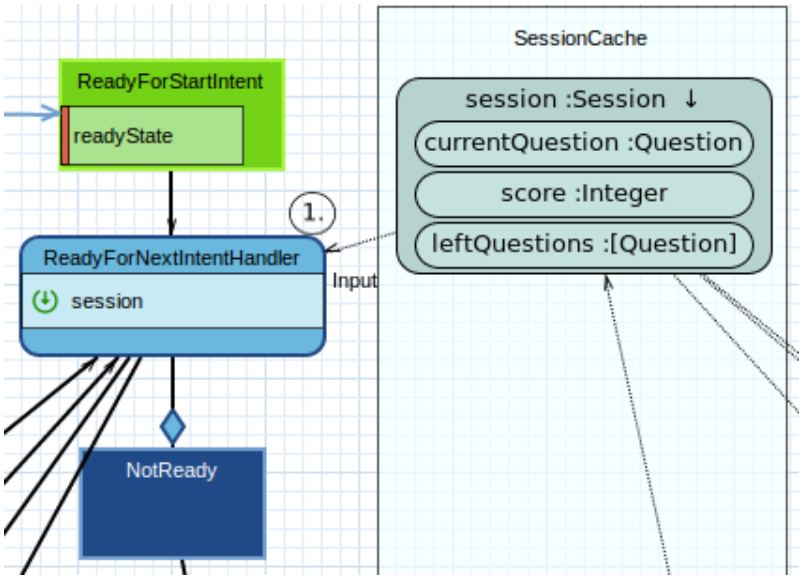


Abbildung 3.9: Server-Knoten

durch die darin enthaltenen Datenobjekte bestimmt. Außerdem besitzt jeder Server-Knoten einen Boolean `reprompt`, den der Skill-Entwickler in der Properties View ankreuzen kann. Ist `reprompt` aktiviert, so kann Alexa noch einmal nachfragen und beispielsweise die gestellte Frage des Quiz wiederholen. Bei Deaktivierung des Booleans tritt dieses Verhalten nicht auf. Auf einen Server-Knoten können beliebig viele *Response*-Knoten folgen. Dabei agiert der Server-Knoten als eine Art Entscheidungsknoten, der später in der Implementierung bestimmt, welcher der Response-Knoten im Dialog mit Alexa ausgewählt wird. Je nach gewählter Response werden die weiteren Pfade festgelegt, sodass sich die Pfade an einem Server-Knoten abzweigen können. Im Falle des Quizbeispiels könnte im Server-Knoten überprüft werden, ob die Antwort *richtig* oder *falsch* ist und dementsprechend wird eine Response ausgewählt.

DIME-Prozess Sollte das ASW-Projekt im Kontext einer DIME-App angelegt worden sein, so können DIME-Prozesse anstelle der Server-Objekte benutzt werden. Hierbei greift der Skill-Entwickler dann auf bereits vorhandene Prozesse aus der DIME-App zu, anstelle diese manuell in einer Implementierungsklasse programmieren zu müssen.

Um einen DIME-Prozess in der graphischen Modellierung benutzen zu können, muss der Skill-Entwickler zunächst einen Intent auf das Graphmodell platzieren. Es besteht die Möglichkeit, dass der Skill-Entwickler einen DIME-Prozess auf das Graphmodell zieht und dort per Kante mit einem Intent verbindet. Somit wird anstelle einer Implementierungsklasse beim Ausführen des Alexa-Skills ein DIME-Prozess angestoßen. Der DIME-Prozess nimmt hierbei den Platz des Server-Knotens ein und auf ihn folgen die EndSIBs aus dem DIME-Prozess, die sich analog zu den Responses nach

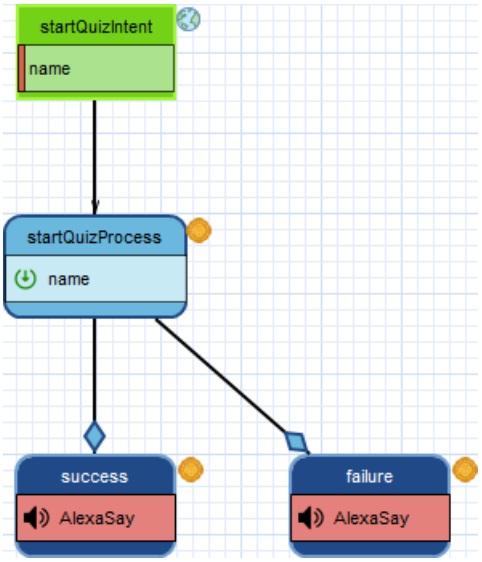
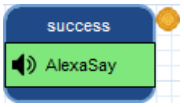
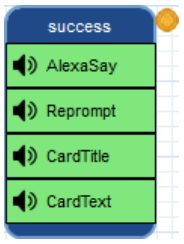


Abbildung 3.10: DIME-Prozess nach einem Intent mit zwei EndSIBs



(a) Ausschließlich AlexaSay-Port



(b) Weitere mögliche Ports

Abbildung 3.11: Daten-Ports im EndSIB mit besonderen Funktionen

Server-Knoten verhalten und ebenfalls Branching ermöglichen. Abbildung 3.10 zeigt einen Intent mit einem darauf folgenden DIME-Prozess und zwei EndSIBs, welche in dem referenzierten DIME-Prozess vorhanden sind. Ein DIME-Prozess wird auf der Skill-DSL mit einem DIME-Icon markiert.

Der AlexaSay-Port in Abbildung 3.10 im referenzierten DIME-Prozess entspricht der eigentlichen Ausgabe des Alexa-Geräts. Wie der Name schon andeutet, wird Alexa dem Benutzer das sagen, was der DIME-Prozess in diesen Port schreibt, daher auch die Notwendigkeit des Ports im referenzierten DIME-Prozess.

Wie in Abbildung 3.10 zu sehen, ist in den beiden EndSIBs der AlexaSay-Port rot markiert. Dieser immer vorhandene Knoten der EndSIBs symbolisiert, ob auf dem referenzierten EndSIB im DIME-Prozess ein AlexaSay-Datenport vorhanden ist. Sollte dies nicht der Fall sein, so wird ein rot markierter AlexaSay Hinweis in den EndSIB auf dem ASW-Graphmodell generiert. Sollte ein solcher Port hingegen vorhanden sein, erkennt dies das ASW-Graphmodell und markiert den AlexaSay Knoten grün (siehe 3.11(a)).

Neben dem AlexaSay-Datenport, welcher immer vorhanden sein muss, existieren noch drei weitere Arten von Ports, welche eine Sonderrolle einnehmen. Neben AlexaSay existieren auch *Reprompt*, *CardTitle* und *CardText*. Diese drei sind jedoch optional und müssen nicht vorhanden sein. Sollten diese im EndSIB des referenzierten DIME-Prozess aufgeführt sein, werden sie mit einer grünen Markierung im Alexa-Graphmodell visualisiert (siehe 3.11(b)).

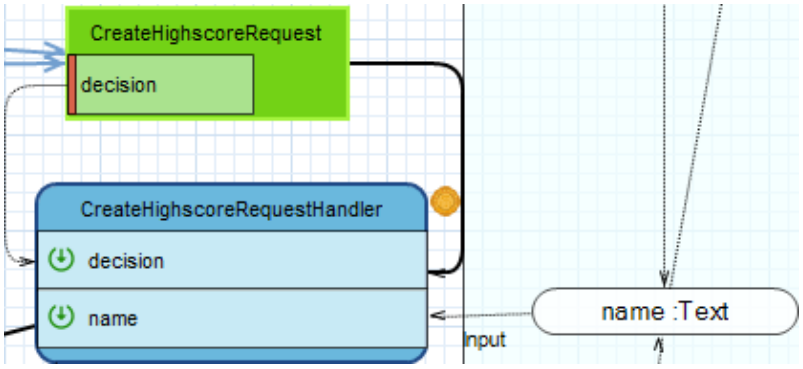


Abbildung 3.12: Dieser DIME-Prozess benötigt Daten, um ausgeführt werden zu können

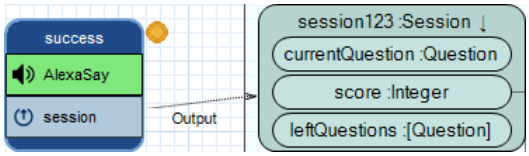


Abbildung 3.13: EndSIB mit Ausgabedaten für den SessionCache

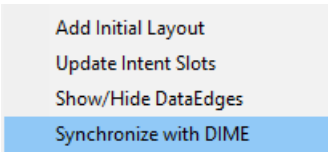


Abbildung 3.14: Synchronisation von DIME und ASW-Graphmodell

Eine Besonderheit von DIME-Prozessen ist, dass die referenzierten DIME-Prozesse Input-Ports besitzen können, die mit Daten befüllt werden müssen, bevor der Prozess ausgeführt werden kann (siehe Abbildung 3.12).

Auf die unterschiedlichen Daten-Typen die ein Daten-Port besitzen kann, soll an dieser Stelle nicht weiter eingegangen werden. Generell gilt, dass alle Ports mit Daten bedient werden müssen. In Abbildung 3.12 kommen die Daten für die Ports einerseits aus dem SessionCache und andererseits aus Slots des vorangegangenen Intents.

Ähnliche Thematik ergibt sich bei den EndSIBs eines referenzierten DIME-Prozesses. Auch die EndSIBs können Daten-Ports besitzen, die über die oben genannten Sonderfälle hinaus gehen. Hierbei handelt es sich dann um Daten, die typischerweise im Rahmen des DIME-Prozesses erstellt werden und zur weiteren Verarbeitung an die EndSIBs übergeben werden. Der Skill-Entwickler hat auch im ASW-Graphmodell die Möglichkeit, diese Ausgabedaten der EndSIBs zu nutzen (siehe Abbildung 3.13).

Sollte ein EndSIB Ausgabedaten besitzen, so können, aber müssen diese nicht, im ASW-Graphmodell weiter verarbeitet werden. In Abbildung 3.13 wird ein Highscore von einem EndSIB zurückgegeben, welcher anschließend in einem Element im SessionCache gespeichert wird.

Ein weiterer wichtiger Punkt ist die Synchronisation der im ASW-Graphmodell dargestellten Prozesse und EndSIBs mit den DIME-Prozessen und dortigen EndSIBs. Sollte ein DIME-Prozess bereits auf dem ASW-Graphmodell benutzt und referenziert worden sein und anschließend etwas an der Struktur der Input-Ports des Prozesses oder der Output-Ports der EndSIBs geändert werden, so muss eine Synchronisierung zwischen DIME und ASW-Graphmodell stattfinden. Hierfür kann über ein per Rechtsklick auf dem ASW-Graphmodell geöffnetes Kontextmenü die Funktion „Synchronize with DIME“ verwendet werden, siehe Abbildung 3.14. Diese Funktion aktualisiert unter anderem alle referenzierten DIME-Prozesse auf dem Graphmodell. Dabei werden EndSIBs und deren Ports, wenn nötig, gelöscht, neu angelegt oder umbenannt.

Response

Eine *Response* folgt immer nach einem Server-Knoten. Mit Response-Knoten wird Branching nach einem Handler-Knoten realisiert. Die von Alexa ausgegebene Antwort, wird vom Skill-Entwickler implementiert und kann sich auch bei demselben Response-Knoten unterscheiden. Für diese Antwort kann auf Datenobjekte via Inputs zugegriffen werden. Durch das Hinzufügen und Entfernen von Datenobjekten wird auch hier die Größe der Response individuell angepasst. Der Skill-Entwickler kann zusätzlich eine beispielhafte Antwort von Alexa im Response-Knoten angeben, sodass diese in Form einer Sprechblase visualisiert wird (siehe Abbildung 3.15 bei (1.)). Anhand einer speziellen graphischen Kantenrepräsentation wird verdeutlicht, dass Server und Response eng miteinander verbunden sind. Auf eine Response folgt entweder ein User- oder ein End-Knoten. Zusätzlich besteht die Möglichkeit des sogenannten *Intent-Chaining*, auf das im Folgenden eingegangen wird.

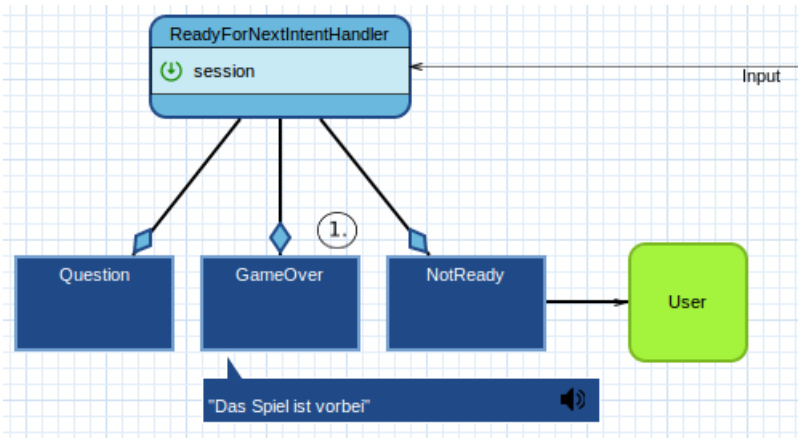


Abbildung 3.15: Response-Knoten

Intent-Chaining

Intent-Chaining ist die Möglichkeit, nach Ausführen einer Response direkt die Slots des nächsten, per Intent-Chaining verbundenen Intents abzufragen. Hierbei ist es möglich, dass auf eine Response oder einen DIME-EndSIB direkt wieder ein Intent folgt, ohne den Umweg über einen UserState. Hierbei gibt Alexa die Antwort der Response aus und fragt anschließend die Slots des per Intent-Chaining verbundenen Intents ab (siehe Abbildung 3.16).

UserState

Ein *UserState* schließt sich einer Response oder einen DIME-EndSIB an und darauf folgt mindestens ein Intent (siehe (1.) in Abbildung 3.17). Er visualisiert, dass sich der Alexa-Nutzer in einem Zustand in der Konversation mit Alexa befindet, in dem er eine Aktion ausüben kann. Dabei wartet Alexa auf den Alexa-Nutzer bis dieser eine Aktion beendet hat. Demnach können von hier aus weitere Intents gestartet werden. Diese Verbundenheit von UserState und Intent-Knoten wird durch eine besondere Darstellung der Kante zum Ausdruck gebracht.

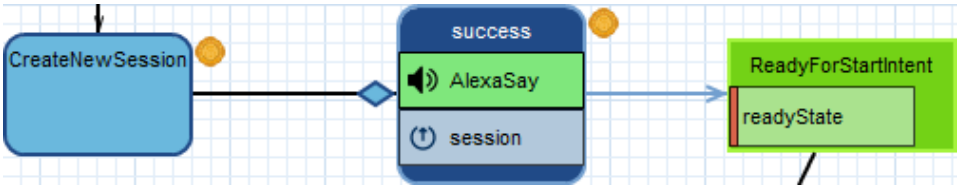


Abbildung 3.16: Intent-Chaining

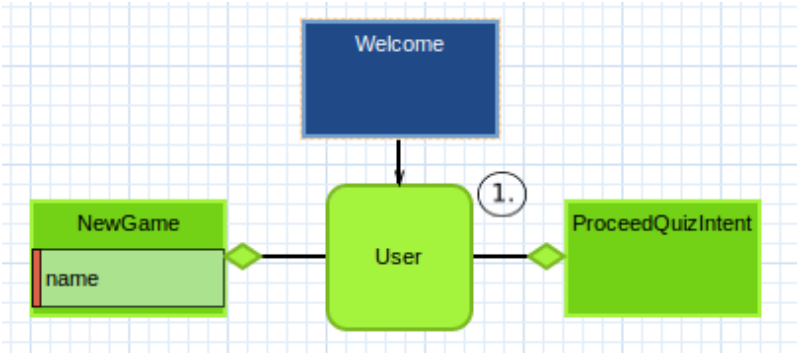


Abbildung 3.17: UserState

End

Der *End*-Knoten symbolisiert das Ende des Skills und folgt immer auf eine Response oder einen DIME-EndSIB (siehe (1.) in Abbildung 3.18). Wenn die Response von Alexa gesprochen wurde, werden die Session und der Skill beendet.

SessionCache

Die Session ist eine Server-seitige Funktionalität, die der Skill-Entwickler verwenden kann, um Informationen während eines Skill-Ablaufs zu speichern. Die Lebensdauer einer Session eines Skills ist gleichzusetzen mit dem Beginn und dem Ende der Alexa-Nutzer-Interaktion mit dem Skill. In der Session kann der Entwickler mittels seiner Implementierung Daten hinterlegen. So kann beispielsweise der Name eines Alexa-Nutzers, der in einem Slot in einem Intent angegeben wurde, gespeichert werden. Dadurch kann der Name auch in einem später folgenden Intent über die Session verwendet werden.

Repräsentiert wird die Session in der Skill-DSL durch den *SessionCache*. Damit kann der Skill-Entwickler für die Speicherung von Daten den *SessionCache* nutzen, um primitive oder komplexe Datenobjekte zwischenzuspeichern. In Abbildung 3.19 ist ein SessionCache-Objekt abgebildet, in dem zwei Datenobjekte vorhanden sind. Der SessionCache kann auf dem Graphmodell vom Skill-Entwickler beliebig oft angelegt und frei umbenannt werden.

Wie in Abbildung 3.20 zu sehen, können Slots direkt mit Datenobjekten im SessionCache verbunden werden. Dies hat zur Folge, dass die vom Benutzer angegebenen Antworten auf die Abfrage von Slots automatisch in der Session gespeichert werden, ohne dass der Skill-Entwickler sich weiter darum kümmern müsste.

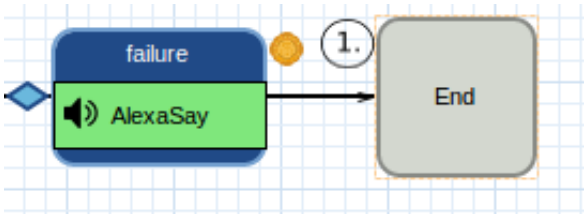


Abbildung 3.18: End-Knoten

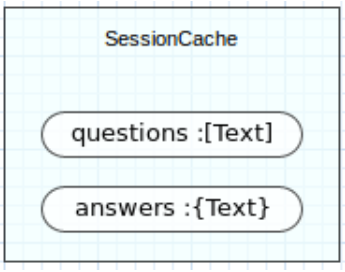


Abbildung 3.19: SessionCache

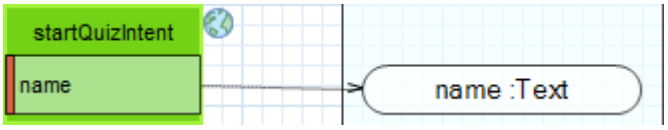


Abbildung 3.20: Automatische Speicherung eines Slots im SessionCache

Datenobjekte

Es wurde bereits erwähnt, dass einige Modellelemente Datenobjekte verwenden können. Dabei ist es möglich primitive oder auch komplexere Datentypen zu verwenden. Dazu werden im Folgenden DataObject und DIME-Daten vorgestellt.

DataObject Primitive Datentypen oder auch Listen und Maps werden durch ein *DataObject* realisiert und können in einem SessionCache abgelegt werden. Als Datentypen stehen Text, Integer, Boolean und Real zur Verfügung. Abhängig von einer Checkbox in der Properties View kann ein *DataObject* als Liste oder Map definiert werden. Je nach Auswahl wird der dargestellte Typ in eckige oder geschweifte Klammern gesetzt, um eine Liste beziehungsweise eine Map zu visualisieren (siehe ① und ② in Abbildung 3.21). Wird beides ausgewählt, so werden sowohl geschweifte als auch eckige Klammern gesetzt (siehe ③))Das gewählte Typsystem gibt vor, dass Maps als Schlüssel immer einen String besitzen.

Zugriffe auf ein Datenobjekt werden durch ein- und ausgehende Kanten dargestellt. Dabei wird ein Input, Output oder Update in dem jeweiligen Modellelement mit einem entsprechenden Icon angelegt. Damit der Nutzer keine Daten mehrfach anlegt, wird dies beim Anlegen direkt verhindert. Falls der Skill-Entwickler mit demselben Datenelement sowohl Input als auch Output anlegt, werden diese automatisch in ein Update umgewandelt. Viele Datenzugriffe hemmen die Übersichtlichkeit (siehe Abbildung 3.22(a)), sodass dem Skill-Entwickler eine weitere Funktionalität geboten wird. Durch einen Rechtsklick auf das ASWorkflow können mithilfe von „Show/Hide DataEdges“ die Kanten ein- und ausgeblendet werden (siehe Abbildung 3.22).

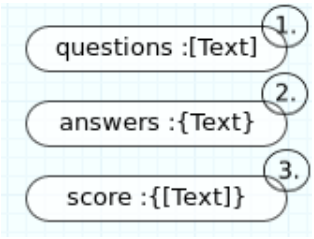
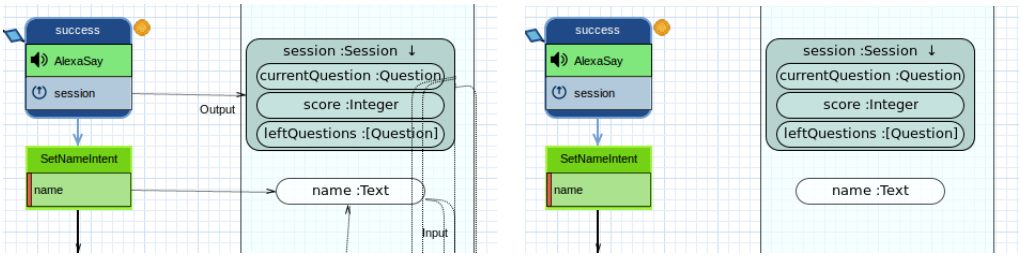


Abbildung 3.21: DataObject als Liste, Map und als Map mit Listen



(a) Ein (b) Aus

Abbildung 3.22: Kantensichtbarkeit

DIME-Daten Damit nicht nur primitive Datentypen beziehungsweise Listen und Maps verwendet werden können, ermöglicht das ASDT auch die Verwendung von DIME-Datentypen. Dafür kann ein bereits bestehendes DIME-Datenmodell verwendet werden, in dem dieses im *Project Explorer* ausgeklappt wird. Die aufgeklappten Elemente können danach in das Modell gezogen werden, um ein DIME-Datenelement zu erstellen (siehe Abbildung 3.23).

Durch das Erstellen wird eine *ComplexVariable* angelegt (siehe ①. in 3.24(a)), welche wiederum primitive (siehe ②. in 3.24(a)) oder komplexe Attribute (siehe ③. in 3.24(a)) beinhalten kann.

Die *ComplexVariable* besitzt einen Namen, der vom Nutzer angepasst werden kann, sowie einen Typen, welcher auch als Liste deklariert werden kann. Dafür wird eine Checkbox in der CINCO Properties View verwendet. Wurde eine Liste definiert, so wird der Typ in eckige Klammern gesetzt und die *ComplexVariable* beinhaltet primitive Listenattribute wie *size* (siehe①. in 3.24(b)) oder auch komplexe Listenattribute wie *first* und *last* (siehe ②. in 3.24(b)).

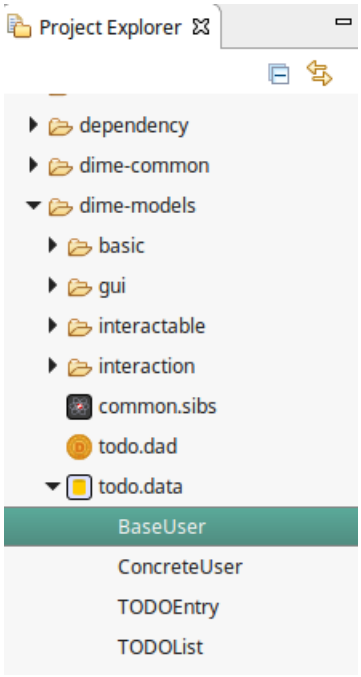
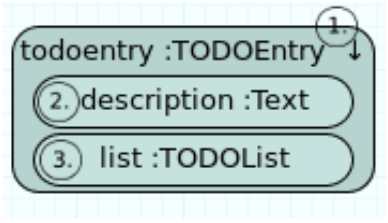
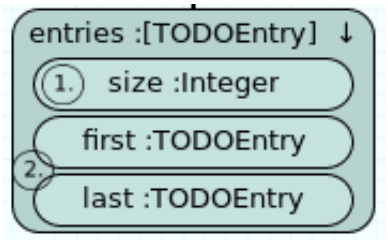


Abbildung 3.23: Verwendung von DIME-Daten



(a) mit primitiven und komplexen Attributen



(b) als Liste mit primitiven und komplexen Listenattributen

Abbildung 3.24: Darstellung der *ComplexVariable*

Durch einen Doppelklick auf die erstellte *ComplexVariable* ist es möglich, diese aufzuklappen (zu sehen in Abbildung 3.25), sodass alle Attribute des Datentyps sichtbar sind und durch Datenkanten ebenfalls referenziert werden können. Mit einem weiteren Doppelklick ist es möglich die Attribute wieder einzuklappen, dabei werden allerdings nur die Attribute eingeklappt, die nicht von einem Knoten für den Datenfluss referenziert wurden.

Unterschiede bei der Nutzung von DIME-Datenelementen sind die Namensgebung eines Input-, Output- oder Update-Knotens, sowie die Verhinderung von Datenzugriffen auf Attribute von Dime Datenobjekten durch Server-Knoten. Zuvor wurde lediglich der Name des referenzierten *DataObject*s verwendet, dies kann aber zu Uneindeutigkeiten führen. Wenn beispielsweise zwei primitive Attribute `description` innerhalb von einer *ComplexVariable* A beziehungsweise B referenziert werden, so reicht es nicht aus nur den Namen des Attributs zu verwenden, da nicht eindeutig ist, ob nun das Attribut von *ComplexVariable* A oder B gemeint ist. Daher wird im Folgenden immer der *fully-qualified name* (FQN) eines Modellelementes benutzt (siehe Abbildung 3.26).

Es ist zudem möglich komplexe Attribute einer *ComplexVariable* aus der Variable zu ziehen. Somit wird die Referenzierung von Attributen einer *ComplexVariable* gestaltet und zugleich das Referenzieren von Attributen eines komplexen Attributes ermöglicht. Abbildung 3.27 stellt dar, wie dies im ASDT aussieht. Durch eine entsprechende Kante wird die Zugehörigkeit der herausgezogenen *ComplexVariable* (in der Abbildung `attributeVariable`) als Attribut einer anderen *ComplexVariable* (in der Abbildung `parentVariable`) deutlich gemacht.

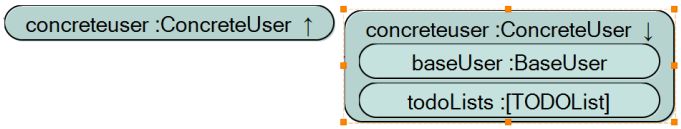


Abbildung 3.25: *ComplexVariable* zugeklappt (links) und aufgeklappt, mit Attributen des Datentyps (rechts)

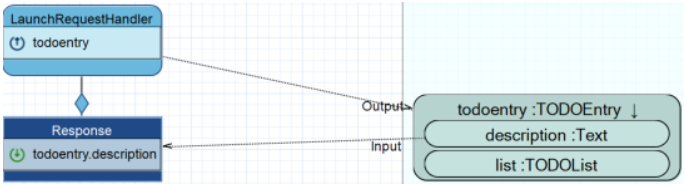


Abbildung 3.26: DIME-Datenzugriffe

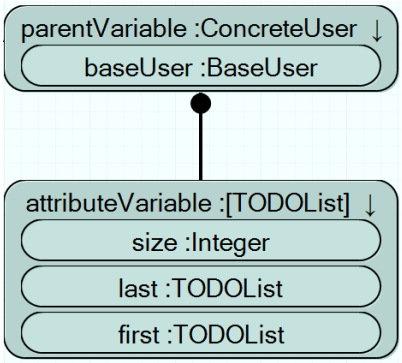


Abbildung 3.27: *ComplexVariable*, dessen Attribut `todoLists` aus der Variable herausgezogen wurde und zur Variablen `attributeVariable` umbenannt wurde

Zusätzlich bietet das ASDT Funktionalitäten, welche die Nutzung und Integration von DIME-Daten in ASW-Modellen erleichtert. Dazu zählt die Möglichkeit die genutzten DIME-Datentypen mit dem entsprechenden DIME-Datenmodell zu synchronisieren, um Änderungen an diesem zu integrieren. Diese Synchronisation kann durch einen Kontextmenü-Eintrag ausgeführt werden. Dafür muss der Modell-Canvas ausgewählt sein und nach einem Rechtsklick, der entsprechende Eintrag „Synchronize with DIME“ ausgewählt werden. Ebenso ist es über den Kontextmenü-Eintrag „Open corresponding DIME Data model“ möglich, dass entsprechende DIME-Datenmodell zu öffnen. Hierzu muss die entsprechende *ComplexVariable* ausgewählt sein, um den entsprechenden Kontextmenü-Eintrag zur Auswahl zu haben.

In Abbildung 3.26 wird die *ComplexVariable* `todoentry` verwendet sowie das dazugehörige primitive Attribut `description`. Für den FQN in dem Input-Knoten ergibt sich für das Attribut *description* daher `todoentry.description`. Falls die Namen der Datenelemente angepasst werden, so werden die Änderungen auch in den referenzierten Input-, Output- oder Update-Knoten übernommen, sodass der FQN dahingehend geändert wird.

3.2.2 MCaM-Validierung

Um den Skill-Entwickler auch während der Modellierung zu unterstützen, erfolgt mittels MCaM die Validierung, welche bereits in Unterabschnitt 2.5.5 vorgestellt wurde. Da es vorkommen kann, dass dem Skill-Entwickler bei der Modellierung Fehler unterlaufen, soll dieser mittels Fehlermeldungen unterstützt werden, um valide Modelle erstellen zu können.

In den folgenden Abschnitten werden die einzelnen Validierungen kurz vorgestellt. Daraufhin wird eine fehlerhafte Modellierung mit den dazugehörigen Fehlermeldungen aufgezeigt.

1. Der Skill-Entwickler kann den Responses individuelle Namen vergeben. Da basierend auf den Namen im nächsten Schritt zugehörige Klassen generiert werden, sollten die vergebenen Namen der JAVA-Konvention für Klassennamen entsprechen. Dies bedeutet, dass der Name mit einem Großbuchstaben beginnen muss. Danach können beliebige Buchstaben, Zeichen und auch Unterstriche folgen (siehe ①. in Abbildung 3.28).

2. Für den Invocation Name stellt Amazon einige Bedingungen auf, die zu erfüllen und somit auch zu validieren sind. Dies bedeutet, dass der Invocation Name nur Kleinbuchstaben, Leerzeichen zwischen Wörtern, Apostrophe und Punkte enthalten darf (siehe ②. in Abbildung 3.28). Intuitiv sollte der Skill-Entwickler den Invocation Name so wählen, dass dieser leicht auszusprechen und eindeutig zu verstehen ist.

3. Unter eindeutiger Namensgebung ist zu verstehen, dass Modellelemente mit derselben Bezeichnung nicht existieren dürfen. Da zu jedem Server-Knoten eine gleichnamige Handler-Klasse generiert wird, welche die erlaubten Pfade in dem Skill-Ablauf überprüft, muss die Benennung eindeutig sein. Eine doppelte Benennung führt zur Generierung einer Klasse, deren Pfadprüfung inkorrekt wäre (siehe (3.) in Abbildung 3.28).

   Hinzu kommt, dass die Namen von Responses, welche nach demselben Server-Knoten folgen, eindeutig gewählt sein müssen. Da basierend auf den gewählten Namen die Generierung der dazugehörigen Klassen erfolgt. Besonders im Fall von Abzweigungen nach einem Server-Knoten endet eine nicht eindeutige Benennung in einem Fehlerfall. Der Skill-Entwickler muss selbst in der Implementierung angeben, welche Response er zurückgeben möchte. Bei gleicher Benennung kann hier also keine Unterscheidung erfolgen.

   Hinzu kommt, dass sowohl DataObject als auch *ComplexVariable*s eindeutig benannt werden müssen, da sonst der FQN uneindeutig ist. Lediglich die Attribute innerhalb einer komplexen Variablen sind bereits im referenziert DIME-Datenmodell eindeutig definiert und müssen nicht zusätzlich validiert werden.

4. Wie bei der Beschreibung der Modellierungssprache bereits vorgestellt wurde, wird im Start-Knoten der Projektname für das zu generierende MAVEN-Projekt angegeben. Dieser darf nicht dem Namen des ASWorkflow-Projekts entsprechen (siehe (4.) in Abbildung 3.28). Diese Prüfung und ob überhaupt ein Projektname gesetzt wurde, greift jedoch nur bei alleinstehenden ASW-Projekten und nicht bei solchen, die in eine DIME-App integriert wurden, da hier kein MAVEN-Projekt in der Generierung erforderlich ist.

5. Bei der Modellierung kann es vorkommen, dass einige Modellelemente erstellt, aber nie verwendet werden, sodass dann eine Warnung ausgegeben wird (siehe (5.) in Abbildung 3.28). Ausgenommen davon sind globale Intents, da diese nicht in den Ablauf durch Kanten integriert werden müssen.

6. Zu jeder referenzierten *ComplexVariable* wird das dazugehörige Objekt aus der Datenbank geholt. Für Attribute einer *ComplexVariable* wird ebenfalls das gesamte Objekt entnommen, um an das enthaltene Attribut zu gelangen. Daher ist es nicht ratsam ein Attribut zu referenzieren, wenn bereits der *parent*, also die *ComplexVariable* verwendet wird. In dem Fall wird eine Warnung ausgegeben, dass der *parent* des Attributs bereits referenziert wird (siehe (6.) in Abbildung 3.28).

   Des Weiteren darf ein Server-Knoten bei den DIME-Datentypen nur *ComplexVariable*s referenzieren, also sogenannte *root*-Datenelemente. Zu jedem Server-Knoten wird bei der Generierung eine Implementierungs-Klasse generiert, sodass dem Skill-Entwickler dort das referenzierte Objekt zur Verfügung steht. Von diesem Objekt

können die einzelnen Attribute abgefragt werden, sodass eine Referenzierung der Attribute in dem Modell selbst nicht notwendig ist (siehe (7.) in Abbildung 3.28).

Wenn eine *ComplexVariable* eine Liste ist, besitzt diese das Listenattribut *size*. Allerdings kann dieses durch eine Output- oder Update-Kante gesetzt werden, wobei dies nicht sinnvoll ist, da die Größe einer Liste fest und daher unveränderbar ist. In dieser Situation wird der Nutzer mithilfe einer Warnung darüber informiert, dass seine Modellierung effektlos ist (siehe (8.) in Abbildung 3.28).

7. Durch die Verwendung von DIME-Prozessen stehen dem Nutzer Ports zur Verfügung, welche mit Datenelementen aus dem SessionCache verbunden werden können. Dabei können nur komplexe Ports mit komplexen Daten verbunden werden sowie der analoge Fall mit primitiven Daten. Allerdings muss zusätzlich überprüft werden, ob die Typen übereinstimmen oder ob die verbundenen Elemente Listen beziehungsweise Maps sind. Für den Fall, dass die Typen nicht übereinstimmen, wird eine entsprechende Fehlermeldung mit einem *type mismatch* ausgegeben. Ist nur eins der verbundenen Elemente eine Liste oder Map, wird ein *list mismatch* beziehungsweise *map mismatch* in Form einer Fehlermeldung ausgegeben (siehe (9.) in Abbildung 3.28).

8. Da es möglich ist Slots eines Intents direkt mit den Input-Ports eines DIME-Prozesses zu verbinden, existiert eine MCaM-Abfrage, dass mit einem Prozess-Input-Port nur die Slots des direkt vorangegangenen Intents verbunden werden dürfen.

9. In dem referenzierten DIME-EndSIB des ASW-Graphmodells muss immer ein AlexaSay-Port vorhanden sein. Sollte dies nicht der Fall sein, so wird der AlexaSay-Port auf dem ASW-Graphmodell einerseits rot markiert und andererseits erzeugt dies eine MCaM-Fehlermeldung (siehe (10.) in Abbildung 3.28). Darüber hinaus muss der AlexaSay-Port im referenzierten DIME-EndSIB vom Typ Text sein, auch dies erzeugt sonst eine MCaM-Fehlermeldung.

10. IntentChaining ist nicht immer erlaubt. Drei Fälle werfen eine MCaM-Fehlermeldung. Zunächst darf IntentChaining nicht mit einem Amazon-Standard-Intent verwendet werden, da diese nur vom Alexa-Benutzer selbst ausgeführt werden dürfen. Zudem darf IntentChaining auch nur mit solchen Intents agieren, die über mindestens einen Slots verfügen, der von Amazon abgefragt werden kann (siehe (11.) in Abbildung 3.28). Als letztes ist es auch nicht erlaubt, IntentChaining mit der Response des gleichen Intents zu verwenden, von dem das IntentChaining gestartet wurde, sodass ein Zyklus entstehen würde.

11. Sollten bei einem DIME-Prozess Input-Ports vorhanden sein, so müssen diese mit Datenkanten belegt werden, also entweder Daten aus dem SessionCache oder den vorangegangenen Slots des Intents befüllt werden. Sollte ein Input-Port eines DIME-Prozesses existieren, welcher keine Daten bekommt, so führt dies zu einer MCaM-Fehlermeldung (siehe (12) in Abbildung 3.28).

Beispiel der Validierung

Um die Validierung abzuschließen, sollen die zuvor beschriebenen Fälle von Validierungen an einem Beispiel gezeigt werden. Daher ist in Abbildung 3.28 ein fehlerhaftes Modell mit den dazugehörigen MCaM-Fehlermeldungen zu sehen.

Damit der Skill-Entwickler auch während der Modellierung kontinuierlich unterstützt wird, werden alle Fehlermeldungen in der *Model Validation View* angezeigt.

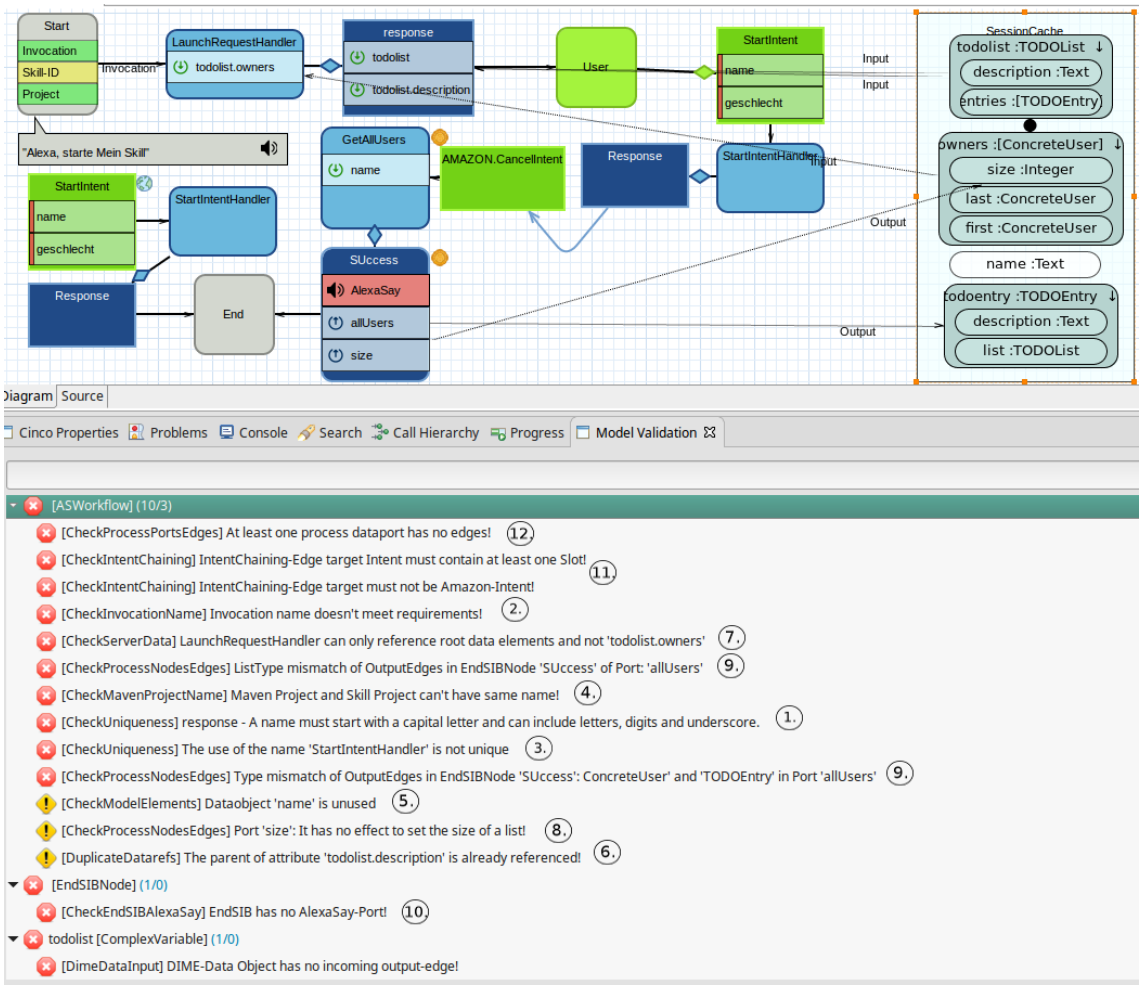


Abbildung 3.28: Beispiel einer fehlerhaften Modellierung

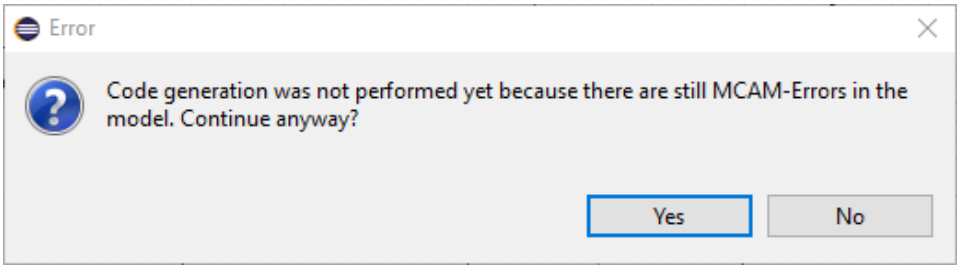


Abbildung 3.29: Warnung, dass trotz Beginn der Generierung noch MCaM-Fehler vorhanden sind

Im Beispiel aus Abbildung 3.28 erfüllt zunächst der Invocation Name des Start-Knotens nicht die Bedingungen, da er Großbuchstaben beinhaltet. Außerdem haben das MAVEN-Projekt und das Skill-Projekt denselben Namen. In beiden Fällen führt dies zu einer Fehlermeldung (siehe ②. und ④.). Der *LaunchRequestHandler* erhält als Input ein Attribut vom root-Datenelement *todolist*, weshalb darauf hingewiesen wird, dass ein Server nur root-Datenelemente referenzieren darf (siehe ⑦.).

Der Name des Response-Knotens *response* beginnt fälschlicherweise nicht mit einem Großbuchstaben und der Knoten selbst beinhaltet sowohl das root-Datenelement als auch dessen Attribut (siehe ①. und ⑥.). Nach dem Response-Knoten *Response* erfolgt unerlaubterweise IntentChaining zu einem Amazon-StandardIntent, welcher zusätzlich keine Slots besitzt (siehe ⑪.). Im Prozess-Knoten *GetAllUsers* befindet sich ein Port, welcher keine eingehenden Kanten besitzt (siehe ⑫.). Im EndSIB *SUccess* wurde der AlexaSay-Parameter nicht gesetzt und der Port *allUsers* wurde einem falschen Typen zugeordnet (siehe ⑩. und ⑨.). Hinzu kommt, dass versucht wird die *size* einer Liste zu setzen, wodurch eine Warnung ausgegeben wird (siehe ⑧.). Des Weiteren wird der Intent *StartIntent* mehrfach verwendet (siehe ③.). Im SessionCache befindet sich das *DataObject name*, welches nicht benutzt wird (siehe ⑤.).

Sollte der Skill-Entwickler trotz vorhandener MCaM-Fehlermeldungen die Generierung anstoßen, so bekommt er einen Warnhinweis angezeigt. Dieser Warndialog gibt dem Skill-Entwickler die Möglichkeit die Generierung abzubrechen und zunächst die vorhandenen Fehler in der Modellierung zu beheben oder alternativ die Generierung trotz einer fehlerhaften Modellierung weiter fortzuführen (siehe Abbildung 3.29).

3.3 Textuelle DSL für Intents

Um die Intents des Alexa-Skills zu definieren, wird eine textuelle DSL entwickelt, die Intents-DSL. Diese hat zwei grundlegende Funktionen: Dadurch können einerseits Intents und deren Slots für die Skill-DSL zur Verfügung gestellt werden und andererseits dient sie zur Generierung der von Amazon geforderten JSON-Datei für das Interaction Model (siehe Unterabschnitt 2.7.2). Dabei ist es wichtig, die Intents-DSL von der relativ unleserlichen JSON-Syntax zu distanzieren. Die Intents-Sprache sollte leserlich und einfach zu verstehen sein sowie Funktionserweiterungen, wie beispielsweise Codevervollständigung, Validierung und Codewiederverwendbarkeit, bieten.

Im Folgenden wird zunächst auf das Metamodell der Intents-DSL eingegangen. Daraufhin wird ihre mittels Xtext definierte konkrete Syntax erläutert. Anschließend wird anhand eines Beispiels die Anwendung der Sprache vorgestellt. Zudem wird auf die Aspekte der semantischen Validierung und der Generierung des Interaction-Model-JSON (siehe Listing 2.13) eingegangen.

3.3.1 Ecore-Modell

Die Anforderungen an das Metamodell (siehe Abbildung 3.30) beinhalten die Erfassung aller notwendigen Informationen für die Interaction-Model-JSON und die Sicherstellung ausreichender Abstraktion. Den Einstiegspunkt bildet wie in der Interaction-Model-JSON das `Interaction Model`. Im Gegensatz zur JSON sind im `Interaction Model` auf oberster Ebene alle wichtigen Bestandteile aufgeführt:

`name:` repräsentiert den Invocation Name des Alexa-Skills.

`intents:` beinhaltet alle Intents des Skills.

`slots:` beinhaltet Slots, die global definiert sind und von allen Intents referenziert werden können.

`types:` beinhaltet alle Types, die im `Interaction Model` verwendet werden.

`prompts:` beinhaltet sowohl Nachfragen (*Elicitation*), als auch Bestätigungsabfragen (*Confirmation Prompts*), die global definiert sind und von allen Intents und Slots referenziert werden können.

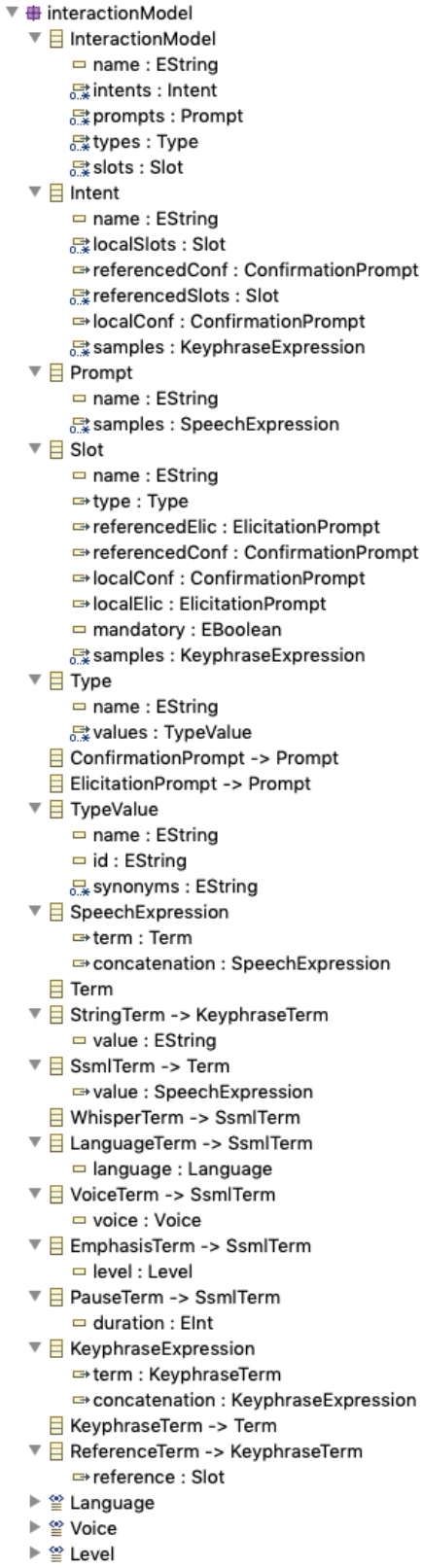


Abbildung 3.30: Ecore-Modell der Intents-DSL

Ein `Intent` besteht aus einem Namen (`name`) zur Identifizierung und einer Liste von Sprachbefehlen (`samples`), mit denen ein Alexa-Nutzer den Intent ansprechen kann. Diese werden im Folgenden auch *Keyphrases* genannt. Außerdem kann ein `Intent` Slots beinhalten, welche entweder direkt im Intent definiert werden (`localSlots`) oder solche, die global definiert sind und im Intent referenziert werden (`referencedSlots`). Zuletzt kann ein `Intent` eine Confirmation besitzen, die wie bei den Slots entweder lokal definiert wird (`localConf`) oder eine Referenz auf eine global definierte Confirmation ist (`referencedConf`).

Ein `Slot` enthält wie ein Intent einen Namen (`name`) sowie eine Liste von Sprachbefehlen (`samples`). Zusätzlich besitzt ein `Slot` einen Type (`type`), welcher referenziert wird. Wie auch bei einem `Intent` können Prompts definiert oder referenziert werden. Dabei wird zwischen einem `ElicitationPrompt` (`localElic`, `referencedElic`) und einem `ConfirmationPrompt` (`localConf`, `referencedConf`) unterschieden. Von beiden Arten von Prompts kann ein `Slot` jeweils maximal einen definieren oder referenzieren.

Ein `Type` setzt sich zusammen aus einem Namen (`name`) und einer Liste von möglichen Werten für diesen `Type` (`values`). Ein `Type` im Alexa-Kontext ist somit als eine Menge von Strings definiert. Dabei besitzt jedes `TypeValue` eine eindeutige ID zur programmatischen Identifizierung dieses spezifischen Werts. Die Bezeichnung, mit der ein Alexa-Nutzer diesen Wert identifizieren kann, wird in `name` gespeichert. Weitere Synonyme für den Wert können in der Liste `synonyms` hinterlegt werden.

Ein `Prompt` besteht aus einem Namen (`name`) und einer Liste von Nachfragen (`samples`), die Alexa an den Nutzer stellen kann. Dabei kann ein `Prompt` entweder ein `ElicitationPrompt` oder ein `ConfirmationPrompt` sein. Auf Ebene des Metamodells unterscheiden sich diese beiden jedoch nicht. Die Aufteilung in zwei verschiedene Unterklassen ist nötig, um auf Ebene der konkreten Syntax eine semantische Unterscheidung vornehmen zu können.

3.3.2 Konkrete Syntax in Xtext

Die konkrete Syntax der Sprache wird mithilfe von Xtext (siehe Abschnitt 2.4) entwickelt. Im Folgenden soll die konkrete Syntax der einzelnen Teile des Metamodells vorgestellt werden und insbesondere auf Unterschiede zwischen dem Metamodell, also der abstrakten Syntax, und der konkreten Syntax eingegangen werden. Zur Veranschaulichung der konkreten Syntax wird zudem ein Beispiel zur Definition eines Interaction Models für einen Quiz-Skill herangezogen. Relevante Ausschnitte werden nach den jeweiligen Regeln erläutert. Zur Übersicht findet sich ein Auszug aus der Intents-Datei in Listing 3.1.

```
intent SetNameIntent {
  mandatory slot name : AMAZON.FirstName {
    elicitation {
      "Wie lautet der Spielername?",
      "Mit welchem Spielernamen wird gespielt?"
    }
    keyphrases {
      "Ich heiße" + name,
      "Mein Name ist" + name,
      name
    }
  }
  keyphrases {
    "Ich heiße" + name,
    "Mein Name ist" + name,
    name
  }
}
type Answer {
  "Dummy Type Value"
}
elicitation answerPrompt {
  "Was ist die richtige Antwort?",
  "Wie lautet die Antwort?",
  "Weißt dus?"
}
intent GetAnswerIntent {
  mandatory slot answer : Answer {
    elicitation answerPrompt
    keyphrases {
      "Die Antwort heißt " + answer,
      "Die Antwort lautet " + answer,
      answer
    }
  }
  keyphrases {
    "Dummy Keyphrase Eins"
  }
}
type Decision {
  "Ja" : ["Okay", "In Ordnung", "Na gut"],
  "Nein" : ["Ne", "Nö", "Nope"]
}
intent CreateHighscoreRequest {
  mandatory slot decision : Decision {
    elicitation {
      "Soll ein Highscore-Eintrag angelegt werden?"
    }
    keyphrases {
      decision
    }
  }
  keyphrases {
    "Dummy Keyphrase Zwei"
  }
}
```

Listing 3.1: Beispiel einer Intents-Datei (gekürzt)

Für das `Interaction Model` (siehe Listing 3.2) wurde beinahe eine Eins-zu-Eins-Repräsentation in der konkreten Syntax gewählt. Einzig der Name eines `Interaction Model`s entfällt in der konkreten Syntax, da dieses für den Nutzer nicht relevant ist. Im Laufe der Generierung wird dieser allerdings noch gesetzt (siehe Abschnitt 4.1). Ansonsten besteht eine Intents-Datei aus Intents, Slots, Confirmations, Elicitations und Types, welche in beliebiger Reihenfolge definiert werden können.

Die Regel zur Definition eines Intents (siehe Listing 3.3) wird eingeleitet durch das Schlüsselwort `intent` gefolgt von dem Namen des Intents. Der Name muss dabei der Konvention einer qualifizierten ID folgen (einzelne IDs, welche durch Punkte getrennt werden). Eine ID, wie sie Amazon definiert, darf dabei auch Bindestriche (`-`) enthalten. In einem Intents-Block können in beliebiger Reihenfolge `Slots`, Keyphrases oder ein `ConfirmationPrompt` angegeben werden. Ein `Slot` beziehungsweise ein `ConfirmationPrompt` kann dabei angegeben werden, indem entweder ein global definierter referenziert oder lokal ein eigener definiert wird. Ein referenzierter `Slot` oder `ConfirmationPrompt` unterscheidet sich von einer lokalen Definition nur darin, dass nach dem Keyword `slot` lediglich der Name des `Slots` oder `ConfirmationPrompts` referenziert wird und kein Definitionsblock mehr folgt. Lokal bedeutet in diesem Fall, dass der `Slot` oder `ConfirmationPrompt` außerhalb des Intents nicht referenziert werden kann. Keyphrases werden dabei in einem eigenen Block definiert und sind, wie auch `Slots` und `ConfirmationPrompts`, optional. Außerdem müssen alle `Slots`, die referenziert oder definiert werden, direkt nacheinander angegeben werden und dürfen nicht durch einen `ConfirmationPrompt` oder den Keyphrase-Block getrennt werden.

In Listing 3.4 ist ein Beispiel für die `IntentDefinition`-Regel aufgeführt. Der `SetNameIntent` dient der Erfassung des Spielernamens und wird beispielsweise durch die Keyphrase „Ich heiße Bob“ angesprochen. Der `Slot name` wird in diesem Fall mit dem Wert „Bob“ gefüllt.

```
1 InteractionModel returns interactionmodel::InteractionModel:
2   {interactionmodel::InteractionModel}
3   (
4     intents+=IntentDefinition |
5     slots+=SlotDefinition |
6     prompts+=(ConfirmationDefinition | ElicitationDefinition) |
7     types+=TypeDefinition
8   )* ;
```

Listing 3.2: `Interaction Model`-Regel in Xtext

```
IntentDefinition returns interactionmodel::Intent:
  {interactionmodel::Intent}'intent' name=KeywordOrID '{'
    (
      (
        'slot' referencedSlots+=[interactionmodel::Slot|KeywordOrID]
        |
        localSlots+=SlotDefinition
      )*
      &
      ('keyphrases' '{'
        samples+=KeyphraseExpression (',' samples+=KeyphraseExpression)*
      '}')?
      &
      (
        ('confirmation' referencedConf=[interactionmodel::ConfirmationPrompt|QID])
        |
        localConf=LocalConfirmationDefinition
      )?
    )
  '}' ;

KeywordOrID :
  'confirmation' | 'elicitation' | 'slot' | 'intent' | QID ;

terminal OWNID:
  ID ( ( '-' )* ID )* ;

terminal QID:
  OWNID ( '.' OWNID )* ;
```

Listing 3.3: `IntentDefinition`-Regel in Xtext

```
intent SetNameIntent {
  mandatory slot name : AMAZON.FirstName {
    elicitation {
      "Wie lautet der Spielername?",
      "Mit welchem Spielernamen wird gespielt?"
    }
    keyphrases {
      "Ich heiße" + name,
      "Mein Name ist" + name,
      name
    }
  }
  keyphrases {
    "Ich heiße" + name,
    "Mein Name ist" + name,
    name
  }
}
```

Listing 3.4: Beispiel für die `IntentDefinition`-Regel

Die Definition eines `Slot`s (siehe Listing 3.5) kann mit dem Schlüsselwort `mandatory` starten. Dies impliziert, dass dieser `Slot` beim Ansprechen des Intents unbedingt einen Wert zugewiesen bekommen muss. Nach dem optionalen `mandatory` muss das Schlüsselwort `slot` und der Name des `Slot`s folgen. Angelehnt an Programmiersprachen, wie beispielsweise C# oder SWIFT, wird der Type eines `Slot`s mit einem Doppelpunkt (`:`) gefolgt vom Namen des Types angegeben. Bei dem Namen handelt es sich um die Referenz auf einen bereits definierten `Type` oder auf Amazon-Types, die im Hintergrund zur Laufzeit in das Modell geladen werden und vom Entwickler referenziert werden können. Amazon-Types sind von Amazon zur Verfügung gestellte Types und umfassen unter anderem Zahlen, Tiere, Personen- und Ortsnamen. Der Type eines Slots spezifiziert dabei die erlaubten möglichen Werte für einen Slot. Ähnlich wie bei der Definition eines Intents können im Definitionsblock Keyphrases, ein `ConfirmationPrompt` und ein `ElicitationsPrompt` angegeben werden. Auch hier können `ConfirmationPrompt` und `ElicitationsPrompt` entweder referenziert oder definiert werden und sind beide optional, genauso wie Keyphrases.

Der in Listing 3.6 definierte `Slot answer` fordert die Angabe einer Antwort auf eine Quizfrage. Er hat den Type `Answer`. Die Angabe dieses `Slot`s ist notwendig (`mandatory`) und benötigt daher auch einen Elicitation-Prompt. Dieser wird außerhalb des `Slot`s definiert und hier referenziert (`elicitation answerPrompt`). Der `Slot` kann mittels der definierten Keyphrases angesprochen und direkt befüllt werden.

Die Definition eines `Type`s (siehe Listing 3.7) erfolgt durch das Keyword `type` gefolgt von dem Namen des Types. Anschließend folgen die Definitionen der einzelnen `TypeValue`s, wobei ein `Type` mindestens einen `TypeValue` haben muss. Ein `TypeValue` besteht dabei aus dem Namen, welcher gleichzeitig der Standardwert ist, und beliebig vielen Synonymen für diesen Wert.

Als Beispiel für einen `Type` wird in Listing 3.8 der `Decision`-Type aufgeführt. Ein `Slot` dieses Typs kann mit einem der beiden `TypeValue`s „Ja“ oder „Nein“ gefüllt werden. Weitere Synonyme für die `TypeValue`s stehen jeweils hinter ihrer Definition in eckigen Klammern.

```
SlotDefinition returns interactionmodel::Slot:
  {interactionmodel::Slot} (mandatory?='mandatory')? 'slot' name=KeywordOrID ':'
      type=[interactionmodel::Type|QID] '{'
    (
      ('keyphrases' '{'
        samples+=KeyphraseExpression (',' samples+=KeyphraseExpression)*
      '}')?
      &
      (
        ('confirmation' referencedConf=[interactionmodel::ConfirmationPrompt|QID])
        |
        localConf=LocalConfirmationDefinition
      )?
      &
      (
        ('elicitation' referencedElic=[interactionmodel::ElicitationPrompt|QID])
        |
        localElic=LocalElicitationDefinition
      )?
    )
  '}' ;
```

Listing 3.5: `SlotDefinition`-Regel in XTEXT

```
mandatory slot answer : Answer {
  elicitation answerPrompt
  keyphrases {
    "Die Antwort heißt " + answer,
    "Die Antwort lautet " + answer,
    answer
  }
}
```

Listing 3.6: Beispiel für die `SlotDefinition`-Regel

```
TypeDefinition returns interactionmodel::Type:
  {interactionmodel::Type}'type' name=QID '{'
    values+=TypeValueDefinition (',' values+=TypeValueDefinition)*
  '}'
;

TypeValueDefinition returns interactionmodel::TypeValue:
    name=STRING (':' '[' synonyms+=STRING (',' synonyms+=STRING)* ']')?
;
```

Listing 3.7: `TypeDefinition` und `TypeValueDefinition`-Regel in XTEXT

```
type Decision {
  "Ja" : ["Okay", "In Ordnung", "Na gut"],
  "Nein" : ["Ne", "Nö", "Nope"]
}
```

Listing 3.8: Beispiel für die `TypeDefinition`-Regel

Die Definition eines `ConfirmationPrompts` (siehe Listing 3.9) kann entweder global oder lokal erfolgen. Diese Unterscheidung macht die abstrakte Syntax nicht. Eine globale `ConfirmationDefinition` unterscheidet sich dabei von der lokalen darin, dass die Confirmation global einen Namen zugewiesen bekommt, lokal allerdings nicht. Der Grund hierfür liegt darin, dass globale Confirmations über ihren Namen referenzierbar sein müssen, lokale Confirmations allerdings nicht referenzierbar sind und somit auch keinen Namen brauchen. Der benötigte Name für das zu generierende JSON wird dabei im Laufe des Generierungsvorgangs berechnet (siehe Abschnitt 4.1). Eine `ConfirmationDefinition` startet immer mit dem Schlüsselwort `confirmation`, auf welches entweder der Name oder der Definitionsblock folgt. Im Definitionsblock werden dabei die einzelnen Samples, die Alexa in zufälliger Auswahl zu dem Nutzer sprechen kann, angegeben.

Analog zu der Definition von `ConfirmationPrompts` erfolgt die Definition der `ElicitationPrompts` (siehe Listing 3.10). Der einzige Unterschied hierbei ist die Verwendung des Schlüsselworts `elicitation` anstelle von `confirmation`. Wie in allen Regeln zu sehen ist, werden in der Intents-DSL geschwungene Klammern `{ }` genutzt, um logisch zusammenhängende Blöcke kenntlich zu machen und voneinander zu trennen. Ebenso werden Kommata verwendet, um eine Aufzählung kenntlich zu machen, sowie im Rahmen der `TypeValueDefinition`-Regel eckige Klammern `[ ]`, um eine Liste von Werten zu signalisieren.

Als Beispiel eines Prompts ist in Listing 3.11 die `elicitation answerPrompt` aufgeführt. Sie enthält eine Rückfrage, die Alexa dem Nutzer stellt, wenn ein notwendiger `Slot` nicht befüllt wurde. Dieser Block wird außerhalb eines `Slots` oder `Intents` (global) definiert und kann daher mittels des Namens `answerPrompt` referenziert werden.

Es gibt zwei verschiedene Arten von Expressions in der konkreten Syntax (siehe Listing 3.12). `KeyphraseExpressions` werden in `Intents` und `Slots` verwendet. Sie definieren Keyphrases, die der Alexa-Nutzer sagen kann, um einen Intent oder Slot anzusprechen. Sie können Strings und Referenzen auf etwaige Slots enthalten. `SpeechExpressions` werden im Gegensatz dazu in Confirmation- und Elicitation-Prompts eingesetzt. Da es sich bei den Prompts um Sprachausgaben von Alexa handelt und nicht um Keyphrases, die der Alexa-Nutzer spricht, ist es möglich zusätzlich SSML (siehe Unterabschnitt 2.7.4) zu nutzen. Strings, `Slot`-Referenzen und SSML-Bausteine können beliebig mit dem Plusoperator (`+`) konkateniert werden.

Als simples Beispiel eines SSML-Ausdrucks ist in Listing 3.13 die `elicitation namePrompt` mit dem `whisper(...)`-Ausdruck aufgeführt. Das Wort „Spielername“ wird bei der Rückfrage von Alexa mit flüsternder Stimme gesprochen.

```
ConfirmationDefinition returns interactionmodel::ConfirmationPrompt:
  {interactionmodel::ConfirmationPrompt}'confirmation' name=QID '{'
    samples+=SpeechExpression (',' samples+=SpeechExpression)*
  '}' ;

LocalConfirmationDefinition returns interactionmodel::ConfirmationPrompt:
  {interactionmodel::ConfirmationPrompt}'confirmation' '{'
    samples+=SpeechExpression (',' samples+=SpeechExpression)*
  '}' ;
```

Listing 3.9: ConfirmationDefinition-, LocalConfirmationDefinition-Regel in XTEXT

```
ElicitationDefinition returns interactionmodel::ElicitationPrompt:
  {interactionmodel::ElicitationPrompt}'elicitation' name=QID '{'
    samples+=SpeechExpression (',' samples+=SpeechExpression)*
  '}' ;

LocalElicitationDefinition returns interactionmodel::ElicitationPrompt:
  {interactionmodel::ElicitationPrompt}'elicitation' '{'
    samples+=SpeechExpression (',' samples+=SpeechExpression)*
  '}' ;
```

Listing 3.10: ElicitationDefinition-, LocalElicitationDefinition-Regel in XTEXT

```
elicitation answerPrompt {
  "Was ist die richtige Antwort?",
  "Wie lautet die Antwort?",
  "Weißt dus?"
}
```

Listing 3.11: Beispiel für die ElicitationDefinition-Regel

```
KeyphraseExpression returns interactionmodel::KeyphraseExpression:
  term=KeyphraseTerm ('+' concatenation=KeyphraseExpression)? ;

SpeechExpression returns interactionmodel::SpeechExpression:
  term=Term ('+' concatenation=SpeechExpression)? ;

Term returns interactionmodel::Term:
  KeyphraseTerm | SsmlTerm ;

KeyphraseTerm returns interactionmodel::KeyphraseTerm:
  StringTerm | ReferenceTerm ;

SsmlTerm returns interactionmodel::SsmlTerm:
  WhisperTerm | LanguageTerm | VoiceTerm | EmphasisTerm | PauseTerm ;

StringTerm returns interactionmodel::StringTerm:
  value=STRING ;

ReferenceTerm returns interactionmodel::ReferenceTerm:
  reference = [interactionmodel::Slot|QID] ;
```

Listing 3.12: SpeechExpression- und KeyphraseExpression-Regeln in XTEXT

```
elicitation namePrompt {
  "Wie lautet der" + whisper("Spielername?")
}
```

Listing 3.13: Einsatz einer SSML-Expression

3.3.3 Validierung

Zusätzlich zu den in der XTEXT vorgegebenen syntaktischen Regeln werden auch statisch-semantische Regeln mittels Validierungen festgelegt. Diese prüfen die Intents-Datei schon während der Bearbeitung. Darunter zählen beispielsweise durch Amazon vorgegebene Ausnahmen, die nicht mittels der konkreten Syntax überprüft werden können. Im Folgenden wird näher auf die implementierten Validierungen eingegangen.

1. Slots, die mit dem Schlüsselwort `mandatory` deklariert sind, benötigen einen entsprechenden Elicitation Prompt. So wird sichergestellt, dass Alexa bei einem fehlenden Wert für einen Slot noch einmal den Alexa-Benutzer nach der benötigten Information fragt.

2. Jeder Slot kann Sprachbefehle (`samples`) beinhalten. Mit ihrer Hilfe kann ein Alexa-Nutzer den Slot mit Informationen füllen. Dazu muss in jedem Sprachbefehl dieses Slots jedoch sein Bezeichner (`name`) enthalten sein (siehe Listing 3.6). Es wird daher geprüft, ob jeder Sprachbefehl eines Slots seinen Namen beinhaltet.

3. Es ist ebenfalls erlaubt, andere Slots desselben Intents in einem Slot-Sprachbefehl zu referenzieren. Daher muss geprüft werden, ob tatsächlich ein weiterer Slot mit dem angegebenen Namen innerhalb desselben Intents existiert. Dies kann nicht automatisch durch XTEXT geprüft werden, da die Sprachbefehle lediglich Strings sind.

4. Ähnlich wie bei Validierung 3, müssen auch die Sprachbefehle der Intents überprüft werden. Diese können ebenfalls Referenzen zu Slots beinhalten.

5. Wenn ein Slot einen Confirmation Prompt besitzt, dürfen die Samples des Prompts hingegen maximal eine Referenz zu dem dazugehörigen Slot beinhalten.

6. Die Sprachbefehle (`samples`) von Intents und Slots werden zusätzlich auf folgende Eigenschaften überprüft: Alle Sprachbefehle dürfen nicht leer sein. Sie dürfen nur Buchstaben, Punkte, Bindestriche, Unterstriche und Referenzen enthalten. Intent-Sprachbefehle müssen eindeutig unter allen Intents und Slot-Sprachbefehle unter allen Slots innerhalb desselben Intents sein. Zusätzlich müssen Intents mindestens einen Sprachbefehl beinhalten, Slots müssen dies hingegen nicht.

7. Die von Amazon zur Verfügung gestellten Standard-Intents können vom Skill-Entwickler mit weiteren Sprachbefehlen erweitert werden. Andere Werte, wie beispielsweise die Slots oder Prompts des Intents, dürfen nicht verändert oder erweitert werden. Wenn der Name eines Intents mit einem Amazon-Intent übereinstimmt, darf der Intent demnach nur das `samples`-Attribut enthalten.

8. Alle Prompts benötigen eindeutige Namen zur Identifizierung. Es wird daher geprüft, ob ihre Namen paarweise verschieden sind. Dazu kommt die Regel, dass Prompt-Namen nicht auf `_ElicitationPrompt` oder `_ConfirmationPrompt` enden dürfen. Prompt-Namen mit diesem Suffix sind für automatisch generierte Namen reserviert, beispielsweise für anonyme lokale Prompts.

9. Wie auch Prompts benötigen Intents und Slots eindeutige Namen. Hier werden also ebenfalls die Namen aller Intents beziehungsweise Slots paarweise verglichen. Gleichzeitig schreibt die Alexa-API vor, dass die Namen nur Buchstaben und Unterstriche enthalten dürfen.

10. Intents dürfen globale Slots nicht mehr als ein Mal referenzieren. Das bedeutet, dass keine Referenz in `referencedSlots` mehrfach enthalten sein darf.

11. In der Alexa-Dokumentation wird ein Ausnahmefall aufgeführt, der ebenfalls validiert werden muss. Ein Intent darf maximal einen Slot besitzen, der vom Typ `AMAZON.SearchQuery` ist. Dies hat den Hintergrund, dass dieser besondere Typ jegliche Spracheingaben akzeptiert und nicht auf vordefinierte Werte verweist. Dies ist vor allem für allgemeine Suchanfragen nützlich. Wären mehrere Slots des Typs zulässig, so könnte Amazon nicht mehr zweifelsfrei bestimmen, welcher Teil einer Spracheingabe zu welchem Slot gehört.

12. Die Namen jedes Werts eines Types dürfen nicht leer sein und müssen eindeutig innerhalb desselben Types sein. Dies liegt unter anderem daran, dass aus den Namen der Werte eindeutige IDs generiert werden müssen.

13. Alle Prompt-Samples, die SSML nutzen, werden auf die Kompatibilität der verwendeten SSML-Bausteine überprüft. So ist es beispielsweise nicht erlaubt innerhalb eines `whisper(...)`-Blocks einen weiteren `whisper(...)`-Block zu verwenden.

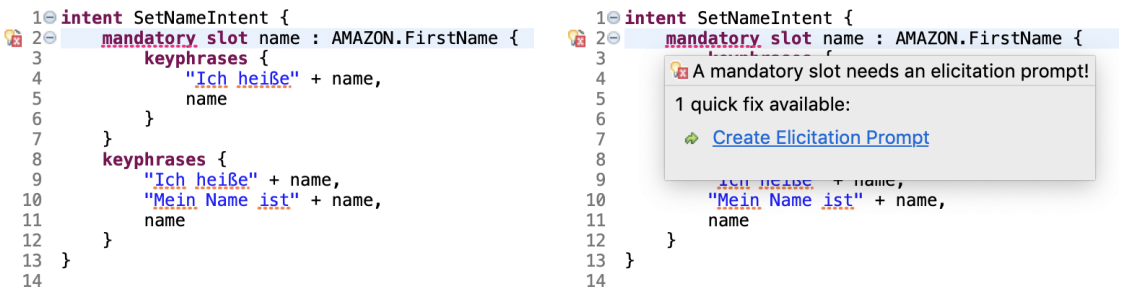
3.3.4 Editor

Im Folgenden wird auf die Funktionen eingegangen, die dem Skill-Entwickler während der Bearbeitung einer Intents-Datei zur Verfügung stehen. Dabei werden lediglich die zusätzlichen, selbst implementierten Funktionen vorgestellt, die nicht automatisch durch das Xtext-Framework bereitgestellt werden (siehe dafür Abschnitt 2.4).

Validierungsregeln (siehe Unterabschnitt 3.3.3) werden während der Codeeingabe kontinuierlich überprüft. Sollte ein Fehler erkannt worden sein, wird die entsprechende Stelle im Quellcode markiert. In Abbildung 3.31 wird ein Fehler durch die Validierungsregel 1 erkannt und dem Entwickler eine Lösung vorgeschlagen.

Codevervollständigung ist eine weit verbreitete, oft genutzte Funktion in Codeeditoren. Sie verhindert falsche Schreibweisen von Schlüsselwörtern sowie Referenzen und beschleunigt die Eingabe. Dem Entwickler werden bei der Bearbeitung von Keyphrases erlaubte Referenzen auf Slots vorgeschlagen. In Abbildung 3.32 findet sich dazu ein Beispiel: Im Slot `name` werden bei der Eingabe von Sprachbefehlen Referenzen auf alle Slots innerhalb des gleichen Intents vorgeschlagen (`name` und `quizName`).

Des Weiteren können auch von Amazon bereitgestellte Standard-Intents und -Types vervollständigt werden. Dabei wurde die Auswahl zunächst auf die in Deutsch verfügbaren Intents und Types beschränkt. Es wurde ein statisches Interaction Model als XMI-Datei erstellt, das alle (deutschen) Standard-Intents und -Types beinhaltet (siehe Abbildung 3.33). Im Hintergrund wird es zur Laufzeit des Editors in den Scope geladen, sodass bei entsprechenden Eingaben Vervollständigungen angeboten werden können.



(a) Quellcode mit Fehlermarkierung

(b) Fehlermeldung mit Lösungsvorschlag

Abbildung 3.31: Beispiel für Validierungsregel 1

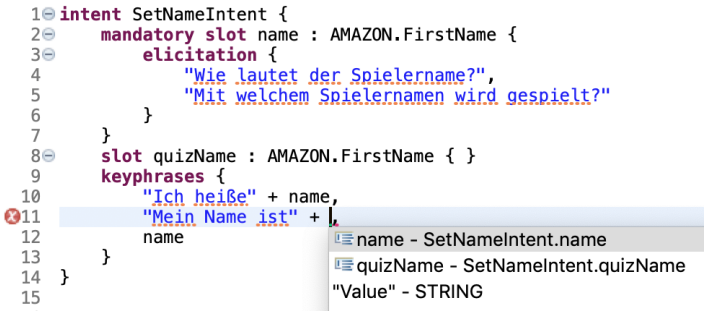


Abbildung 3.32: Codevervollständigung einer Slot-Referenz

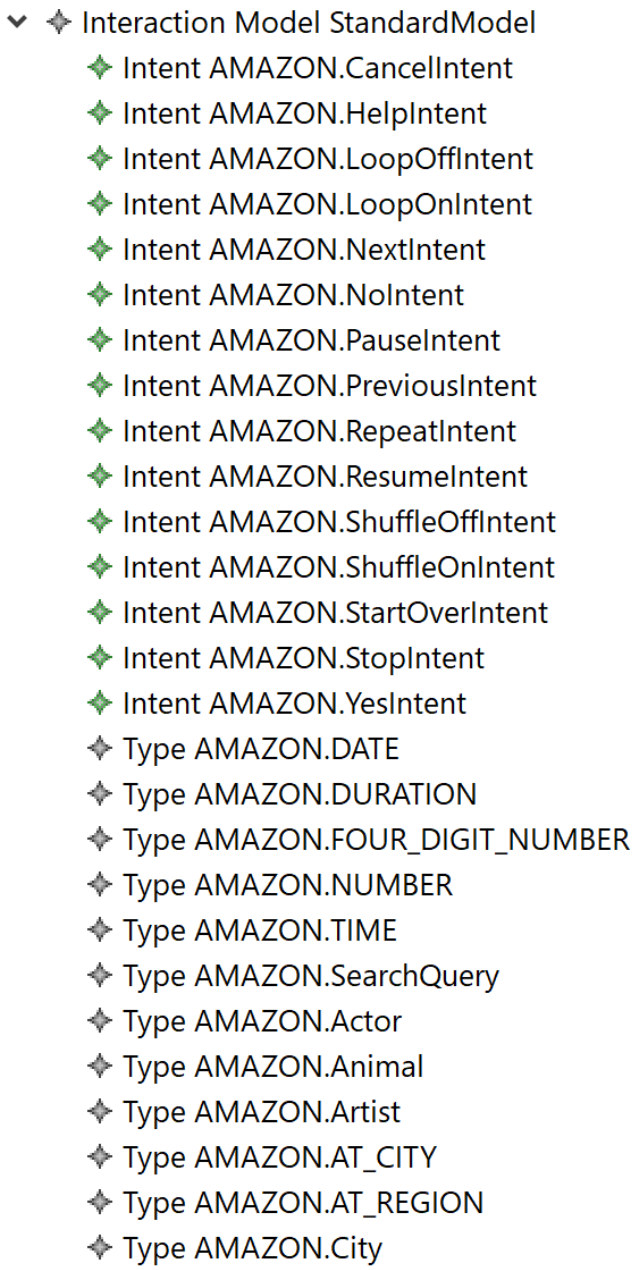


Abbildung 3.33: Statisches Interaction Model

3.4 Textuelle DSL für das Skill-Manifest

Für die Konfiguration eines Skills wird ein Manifest verwendet (siehe Abschnitt 2.7.5), in dem allgemeine Angaben über den Alexa-Skill hinterlegt werden. Diese sind zum einen technisch, wie der Verweis auf den Endpoint, an dem letztendlich die Verarbeitung der Intents erfolgt, und zum anderen den Skill beschreibend, wie der Name sowie eine Beschreibung des Skills.

Dieses Manifest wird mittels einer textuellen DSL, der Manifest-DSL, und einem JSON-Generator erstellt, die sich auf zwei Metamodelle beziehen. Das von der Manifest-DSL definierte Metamodell beschriebt einen Teilbereich des Manifestes und ist dabei so aufgebaut, dass es von der vorgegebenen Struktur abstrahiert in einen intuitiveren Aufbau überleitet. Die Modelle, welche vom Benutzer durch die DSL erstellt werden, sind Dateien mit der Endung `.skillmanifest`. Aus diesen Modellen werden dann mittels des JSON-Generators die `skill.json` Dateien erstellt, die das eigentliche Manifest des Skills darstellen. Mit diesem Ansatz wird eine möglichst hohe Wiederverwendbarkeit der Informationen ermöglicht, die um fehlende Informationen im ersten Metamodell ergänzt werden.

3.4.1 Ecore-Modell

Das Metamodell der Manifest-DSL bildet zum aktuellen Zeitpunkt nahezu alle benötigten Informationen ab, die für das Manifest benötigt werden. Die Unterschiede zwischen dem ersten auf die manuelle Eingabe fokussierten und dem zweiten auf die letztendliche Generierung des Manifestes ausgelegte Modell (siehe Abbildung 3.34) liegen in ihren Strukturen. Das `.skillmanifest` Modell unterscheidet sich strukturell vom Aufbau des Manifestes sowie dem `skill.json` Modell und bietet somit eine Abstraktion zur JSON-Syntax. Zudem sind für einige Bestandteile nicht veränderbare Standardwerte voreingestellt, weil diese bisher noch nicht in der Alexa-Skill-Entwicklungsumgebung unterstützt werden. Ein Beispiel dafür ist das Feld `allowsPurchases`, das signalisiert, ob innerhalb des Skills Käufe möglich sind. Da dies bisher seitens der Implementierung nicht möglich ist, kann dieser Wert standardmäßig auf `false` gesetzt werden (siehe Abbildung 3.35).

Den Einstieg in das Manifest bietet die `ManifestJson`. Ihr sind folgende Bestandteile untergeordnet:

`manifestVersion:` Gibt die aktuelle Version des Alexa-Skills an.

`publishingInformation:` Beinhaltet Informationen für den Store.

`apis:` Listet die verwendeten APIs inklusive ihrer Endpoints auf.

`permissions`: Beinhaltet alle benötigten Berechtigungen für die Ausführung des Alexa-Skills.

`privacyAndCompliance`: Beinhaltet Informationen bezüglich des Daten- und Jugendschutzes.

`events`: Beinhaltet die Events, auf die der Skill reagiert.

Die `manifestVersion` ist ein String mit der Version des Manifestes nach der gebräuchlichen Schreibweise und dient zur eigenen Übersicht. Die `publishingInformation` enthalten die global gültigen Attribute, die angeben, ob der Alexa-Skill weltweit verfügbar ist (`isAvailableWorldwide`), wie der Skill getestet werden kann (`testingInstructions`) und welcher der von Amazon vorgegebenen Kategorie dieser zugeordnet werden kann (`category`). Sollte ein Skill nicht weltweit verfügbar sein, werden darüber hinaus die Länder explizit angegeben, in denen der Alexa-Skill verfügbar ist (`distributionCountries`). Zusätzlich werden regionsspezifisch im `Locales` weitere Inhalte definiert. Dazu zählen eine kurze Beschreibung des Skills (`summary`), eine Liste von beispielhaften Äußerungen (`examplePhrases`) und Schlagwörtern (`keywords`) sowie URLs zu einem Vorschau- (`smallIconUri`) und einem Anzeigebild (`largeIconUri`). Zuletzt wird noch der Name (`name`) des Alexa-Skills für die jeweilige Region und eine Beschreibung (`description`) angegeben.

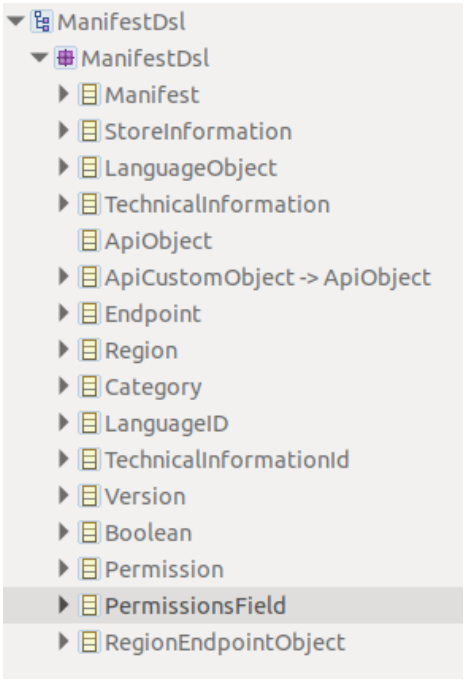


Abbildung 3.34: Ecore-Modell der Manifest-DSL

Property	Value
Changeable	false
Container	false
Containment	true
Default Value Literal	
Derived	false
EKeys	
EOpposite	
EType	Boolean
Lower Bound	0
Name	purchases
Ordered	true
Resolve Proxies	true
Transient	false
Unique	true
Unsettable	false
Upper Bound	1
Volatile	false

Abbildung 3.35: Eigenschaften des Attributes `allowsPurchases`

Darüber hinaus wird im `privacyAndCompliance` angegeben, ob Käufe innerhalb des Alexa-Skills möglich sind (`allowsPurchases`), persönliche Informationen verwendet werden (`usesPersonalInfo`), dieser für Kinder genehmigt ist (`isChildDirected`), der Skill in weiteren Regionen verwendet werden darf (`isExportCompliant`) und ob dieser Werbung enthält (`containsAds`). Des Weiteren kann für die einzelnen Regionen (`PrivacyLocale`), welche über die `id` bestimmt werden, ein Link zu den Datenschutzbestimmungen (`privacyPolicyUrl`) sowie zu den Nutzungsbedingungen (`termsOfUseUrl`) hinterlegt werden. Ausgewählte Werte, wie der zuvor genannte Indikator für Käufe innerhalb des Skill, werden in dem ersten Metamodell auf `false` gesetzt. Zu einem späteren Zeitpunkt der Entwicklung kann dann für das Metamodell des Manifestes diese Informationen auch aus dem graphischen Modell bezogen werden, indem der Generator angepasst wird.

Das `apis`-Objekt beinhaltet anstelle einer Liste von mehreren möglichen API-Objekten, als Repräsentation der von Amazon angebotenen APIs, ausschließlich ein individualisierbares `Custom`-Objekt. Zum einen kann über dieses Objekt der `endpoint` angegeben werden und zum anderen werden die weiteren API-Typen nicht benötigt, weil diese von der graphischen Modellierungssprache bisher nicht unterstützt werden.

Zuletzt können die benötigten Berechtigungen (`Permissions`) und auf die zu reagierenden Ereignisse (`Events`) angegeben werden. Da diese Funktionen noch nicht unterstützt werden, bleiben diese jedoch leer.

3.4.2 Syntax in Xtext

Auch für diese Sprache wird die Syntax mittels Xtext (siehe Abschnitt 2.4) erstellt. Hierbei ist insbesondere die Syntax für die Eingabe des ersten Metamodells relevant, da diese von dem Aufbau und den Bezeichnungen des Manifestes abweicht, um möglichst intuitiv gestaltet zu sein. Auf die spezifischen Eigenschaften von Xtext, die in diesem Rahmen Anwendung finden, wird in Unterabschnitt 3.3.2 eingegangen.

Der Einstieg erfolgt über das `Manifest`, in dem die Manifestversion sowie die oben genannten Bestandteile angegeben werden, soweit diese manueller Eingaben bedürfen (siehe Listing 3.14). Die `Version` (siehe Listing 3.15) besteht dabei aus zwei Teilen, der Hauptversionsnummer vor sowie der Nebenversionsnummer nach dem Punkt.

Die Informationen werden gegenüber der Struktur im Manifest so unterteilt, dass die für den Store relevanten Informationen auf der einen Seite (`StoreInformation`) und die technischen Informationen für den Betrieb des Alexa-Skills auf der anderen Seite (`TechnicalInformation`) gebündelt und in beliebiger Reihenfolge angegeben werden können.

Die `StoreInformation` (siehe Listing 3.16) können entweder für die einzelnen Regionen als Liste von `Region`-Objekten angegeben oder global definiert werden, was durch die Wertzuweisung des Attributes `global` signalisiert wird. Die Werte für `testInstructions`, `childDirected` und `exportCompliant` werden als primitive Datentypen in Form von Strings beziehungsweise Booleans angegeben. Für die sprachspezifischen Eingaben wird eine Liste von `LanguageObjects` angegeben, die mindestens einen Eintrag enthalten muss. Zuletzt kann optional für die Kategorie ein `Category`-Objekt angelegt werden.

```
1  Manifest returns manifest::Manifest:
2    'manifestversion' ':' version = Version ';'
3    (
4      ( storeInformation = StoreInformation )
5      &
6      ( technicalInformation = TechnicalInformation )
7    ) ;
```

Listing 3.14: `Manifest`-Regel in XTEXT

```
1  Version returns manifest::Version:
2    major = INT '.' minor = INT ;
```

Listing 3.15: `Version`-Regel in XTEXT

```
1  StoreInformation returns manifest::StoreInformation:
2    'storeInformation' '{'
3      (
4        ( 'regions' ':'
5          (
6            '[' regions+=[manifest::Region|ID] (',' regions+=[manifest::Region])* ']'
7            |
8            global ?= 'global'
9          )
10       )
11       &
12       ( 'testingInstructions' ':' testInstructions = STRING )
13       &
14       ( 'purchases' ':' purchases = Boolean )
15       &
16       ( 'personalInformationUsage' ':' personalInformation = Boolean )
17       &
18       ( 'childDirected' ':' childDirected = Boolean )
19       &
20       ( 'exportCompliant' ':' exportCompliant = Boolean )
21       &
22       ( 'advertisement' ':' advertisement = Boolean )
23       &
24       ( ( languageObject += LanguageObject )+ )
25       &
26       ( ( 'category' ':' category = [manifest::Category|ID] )? )
27     )
28   '}' ;
```

Listing 3.16: `StoreInformation`-Regel in XTEXT

Im `LanguageObject` (siehe Listing 3.17) sind insbesondere `examples` und `keywords` zu erwähnen. Im Ersteren müssen genau drei Strings angegeben werden, während im Letzteren eine beliebig lange Liste an Keywords erstellt werden kann, solange diese mindestens ein Element enthält.

Auf der anderen Seite gibt es die `TechnicalInformation` (siehe Listing 3.18), die eine beliebig lange Liste von `ApiCustomObject`, einen allgemeinen `Endpoint` sowie einen `Endpoint` für die Region enthält.

Das `ApiCustomObject` selbst (siehe Listing 3.19) besteht insbesondere aus einem String für den `apiName` und einem `Endpoint` (siehe Listing 3.20). Dieser wiederum enthält zwei Strings für die `uri` und den `sslCertificateType`.

Um dem Benutzer die Handhabung der Manifest-DSL zu vereinfachen, wird bei der Erstellung des ASW-Projektes eine Skill-Manifest-Datei-Vorlage generiert. Diese beinhaltet die für ein Deployen bei Amazon nötigen Informationen, welche in die `skill.json` – also die eigentliche Manifest-Datei – generiert werden. Für eine Minimallösung müssen hier lediglich die `uri` unter `endpoint` und der Zertifikatstyp unter `sslCertificateType` angegeben werden (siehe Listing 3.21).

```
1  LanguageObject returns manifest::LanguageObject:
2    'language' languageId = [manifest::LanguageID|AMAZONLANGUAGEID] '{'
3      (
4        ( 'summary' ':' summary = STRING )
5        &
6        ( 'examples' ':' '[' examples += STRING ','
7                             examples += STRING ','
8                             examples += STRING ']' )
9        &
10       ( 'keywords' ':' '[' keywords += STRING ( ',' keywords += STRING )* ']' )
11       &
12       ( 'smallIconUri' ':' smallIcon = STRING )
13       &
14       ( 'largeIconUri' ':' largeIcon = STRING )
15       &
16       ( 'name' ':' name = ID )
17       &
18       ( 'description' ':' description = STRING )
19       &
20       ( 'privacyPolicyUrl' ':' privacyPolicy = STRING )
21       &
22       ( 'termsOfUseUrl' ':' termsOfUse = STRING )
23     )
24   '}' ;
```

Listing 3.17: `LanguageObject`-Regel in Xtext

```
TechnicalInformation returns manifest::TechnicalInformation:
  'technicalInformation' '{'
    ( apiCustomObject += ApiCustomObject )* &
    ( 'endpoint' ':' endpointId = STRING ) &
    ( 'endpoint' region = [manifest::Region|ID] ':' endpointIdRegion = STRING )
  '}' ;
```

Listing 3.18: TechnicalInformation-Regel in XTEXT

```
ApiCustomObject returns manifest::ApiCustomObject:
  'api' ( apiName = STRING ) '{'
    endpoint = Endpoint
  '}' ;
```

Listing 3.19: ApiCustomObject-Regel in XTEXT

```
Endpoint returns manifest::Endpoint:
  'endpoint' ':' '{'
    'uri' ':' uriId = STRING
    'sslCertificateType' ':' sslCertificateType = STRING
  '}' ;
```

Listing 3.20: Endpoint-Regel in XTEXT

```
manifestversion : 1.0;
storeInformation {
  regions : global
  testingInstructions : "Testing Instructions Placeholder"
  language de-DE {
    summary : "Summay Placeholder"
    examples : ["Example 1", "Example 2", "Example 3"]
    description : ""
    keywords : ["Keyword Placeholder"]
    smallIconUri : ""
    largeIconUri : ""
    name : namePlaceholder
    privacyPolicyUrl : ""
    termsOfUseUrl : ""
  }
  advertisement : false
  childDirected : false
  personalInformationUsage : false
  purchases : false
  exportCompliant : false
}
technicalInformation {
  api "API Placeholder" {
    endpoint : {
      uri : "URI Placeholder"
      sslCertificateType : "Trusted"
    }
  }
}
permissions {
  alexa::alerts:reminders:skill:readwrite
}
```

Listing 3.21: Generierte Vorlage einer Skill-Manifest-Datei

3.4.3 Validierung

Für gewisse Bestandteile im Manifest werden Validierungen durch MCaM durchgeführt, um die Vorgaben an das Manifest auch auf semantischer Ebene zu erfüllen. Folgende Punkte sind dabei relevant:

1. Ein Endpoint muss stets mit `https://` beginnen und darf nicht `null` oder leer sein.
2. Das verwendete Sicherheitszertifikat muss `Trusted`, `SelfSigned` oder `Wildcard` sein.
3. Die Region des Servers muss `de-DE` sein, sofern der Skill in Deutschland eingesetzt werden soll.

3.5 Implementierungsklassen

Unabhängig davon, ob der Skill in eine DIME-App integriert wird oder nicht, stehen dem Benutzer die Implementierungsklassen nach initialer Ausführung der Generierung zur Verfügung bei Verwendung von Server-Knoten auf dem ASW-Graphmodell. In den Implementierungsklassen, kurz folgend *Impl-Klassen*, wird die Geschäftslogik der Intents programmiert als Alternative zur Verwendung von DIME-Prozessen. Damit ist gemeint, dass in den Impl-Klassen die Aktionen spezifiziert werden, welche der Server ausführen soll, wenn ein Alexa-Nutzer einen bestimmten Intent anspricht und das Servlet und der jeweilige Handler die Impl-Klasse zur Ausführung bringen. Die Impl-Klassen sind in JAVA geschrieben.

3.5.1 Intents und Impl-Klassen

Pro Server-Objekt im Graphmodell wird in einem ersten Generierungsschritt eine Impl-Klasse generiert, wodurch jeder vom Alexa-Nutzer angesprochene Intent unterschiedliche Aktionen auf dem Server ausführen und damit auch unterschiedliche Antworten an den Alexa-Nutzer über das *Amazon Voice Interface* zurückgeben. Darüber hinaus kann derselbe Handler von mehreren Intents im Modell referenziert und somit im Skill-Ablauf aufgerufen werden. In diesem Sonderfall wird nur eine Impl-Klasse generiert, welche für alle Intents die gleichen vom Skill-Entwickler spezifizierten Aktionen ausführt. An dieser Stelle muss der Skill-Entwickler mit JAVA programmieren. Dies liegt begründet in der Tatsache, dass durch das Graphmodell und die dort vorhandenen Informationen nicht durch den Generator die Geschäftslogik eines aufgerufenen Intents hervorgeht. Gemeint ist damit, dass der Generator nicht wissen kann, was bei Aufruf eines Intents auf dem Server erfolgen

soll, die sogenannte Geschäftslogik. Als Alternative steht hier weiterhin das Nutzen eines DIME-Prozesses zur Verfügung, aus der dann auch die Geschäftslogik hervorgeht.

Die Impl-Klasse erlaubt dem Skill-Entwickler unter anderem das Auslesen von Alexa-Nutzer-Angaben aus den Slots und das Speichern von Daten in einer Session. Folgend sind die wichtigsten Funktionen der Impl-Klasse aufgelistet, die dem Skill-Entwickler zur Verfügung stehen:

- Parameter `slots`
- Parameter `session`
- Parameter `responses`
- Speech-Objekt für SSML

Dem Skill-Entwickler werden in der Impl-Klasse, welche technisch eine Implementierung eines Interfaces ist, drei Parameter übergeben/bereitgestellt, dessen Inhalte er verwenden kann (siehe Listing 3.22). Als Rückgabewert wird eine der auf der Skill-DSL angelegten Responses erwartet. Auf die oben genannten Funktionen der Impl-Klassen wird im Folgenden weiter eingegangen.

3.5.2 Slots-Parameter

Sollte der Intent, zu dem die Implementierungs-Klasse gehört, über Slots verfügen, in welchen der Alexa-Nutzer Daten angeben kann, so stehen diese Daten im `slots`-Parameter. Damit kann der Entwickler, um auf die Antworten des Alexa-Nutzers zu reagieren, auf Rückfragen durch Alexa zugreifen. Die Daten in den Slots liegen standardmäßig als Strings vor. Sprachaufnahmen des Alexa-Nutzers werden von Amazon in Text übersetzt und per JSON an den Server gesendet. Dort werden diese als `slots`-Parameter der Impl-Klasse übergeben.

```
1  public AnswerIntentHandlerResponse getResponse(
2      AnswerIntentHandlerSlots slots,
3      AnswerIntentHandlerSession session,
4      AnswerIntentHandlerResponse_Handler responses) {
5    ...
6    return responses.getStartResponse1_Response()
7            .setSpeech(speech)
8            .setRepromptText(speech)
9            .setDisplayText("Welcome to my app")
10           .setDisplayTitle("Welcome");
11 }
```

Listing 3.22: Beispiel der Impl-Klassen-Parameter als Teil der Signatur

Die Daten der Slots können hier nur ausgelesen, nicht jedoch geschrieben werden, da die Slots die Antwort des Alexa-Nutzers auf eine Nachfrage repräsentieren.

In dem Beispiel (siehe Listing 3.23) wird davon ausgegangen, dass der Intent, zu dem dieses Beispiel gehört, einen Slot `Username` besitzt. Der Alexa-Nutzer wird, falls der Slot im Graphmodell als *mandatory* gekennzeichnet wurde, nach seinem gewünschten `Username` gefragt. Die vom Alexa-Nutzer gegebene Antwort befindet sich in dem Slots-Objekt und kann damit in der Impl-Klasse weiter verarbeitet werden. Sollte der Slots nicht *mandatory* gewesen und vom Benutzer nicht befüllt worden sein, so wird ein leerer String zurückgegeben. Es bietet sich jedoch an, um mögliche unerwünschte Seiteneffekte zu vermeiden, stets mit der Angabe des *mandatory* zu arbeiten.

3.5.3 Session-Parameter

Als Nächstes steht dem Entwickler der `session`-Parameter zur Verfügung. Je nachdem welche Datenobjekte der Skill-Entwickler im Graphmodell an dem Server-Objekt per Kanten annotiert hat, bekommt er hier über dieses Objekt auf eben diese Daten Zugriff. Wichtig ist hierbei, dass im Session-Objekt auch nur jene Datenobjekte vorhanden sind, welche im Graphmodell mit dem Server verbunden wurden und nicht etwa an die darauf folgenden Responses.

Auf dem Graphmodell gibt es die Möglichkeit, von einem Datenobjekt Input-, Output- oder Update-Kanten zum Server-Knoten zu ziehen. Dadurch wird das Datenobjekt im Server-Knoten/-Objekt angelegt. An dieser Stelle der Implementierung macht sich die Logik der unterschiedlichen Kantenarten bemerkbar. Sollte der Skill-Entwickler eine Input-Kante vom Datenobjekt zum Server-Knoten auf dem Graphmodell gezogen haben, so existiert im Session-Objekt lediglich eine `get`-Funktion. Dies bedeutet, dass der Skill-Entwickler bei der Programmierung der Impl-Klasse über das Session-Objekt nur lesend auf das Datenobjekt und die darin enthaltenen Daten in der Session zugreifen kann, nicht jedoch schreibend. Sollte eine Output-Kante vom Server-Objekt zu einem Datenobjekt im Graphmodell vorliegen, so bekommt der Skill-Entwickler im Session-Objekt, gemäß Interpretation der Output-Kante, auch nur eine `set`-Funktion an die Hand gegeben. Dies wiederum bedeutet, der Skill-Entwickler kann in der Impl-Klasse im Session-Objekt nur Daten in das Datenobjekt in der Session schreiben.

```
1 String username = slots.getUsername();
```

Listing 3.23: Beispiel der Verwendung eines Slots

```
1 String username = session.getUsername();
2 session.setUserLanguage(language);
```

Listing 3.24: Beispiel der Verwendung eines Session-Parameters

Sollte der Skill-Entwickler Daten über eine `set`-Funktion des Session-Objekts schreiben, so sind diese automatisch im JSON vorhanden, welches wieder zurück als Antwort an Amazon gesendet wird. Hierfür reicht die Ausführung der `set`-Funktion innerhalb der Session und damit ist keine weitere Aktion seitens des Skill-Entwicklers nötig. Sollte der Skill-Entwickler im Graphmodell die dritte und letzte Variante zum Anlegen eines Datenobjekts im Server-Objekt gewählt haben, also eine Update-Kante, so stehen ihm sowohl `get-` als auch `set`-Funktion für das entsprechende Datenobjekt zur Verfügung. Mit Update-Kanten können also sowohl vom gleichen Datenobjekt in der Session Daten gelesen als auch geschrieben werden. Es können nur solche Daten aus der Session gelesen oder in die Session geschrieben werden, die auch im Graphmodell an den Servern annotiert wurden. Dadurch ergibt sich der Mehrwert der Datenobjekte innerhalb der Server-Objekte und der verwendeten Kantenart in der Skill-DSL.

In Listing 3.24 sind beide wichtigen Funktionen des Session-Objekts abgebildet. Zunächst wird der in der Session gespeicherte Name ausgelesen, den der Alexa-Nutzer vorher im Rahmen eines Slots eines anderen Intents angegeben hat. Hat der Alexa-Nutzer in einem anderen Intent in einem Slot Daten hinterlegt, welche im weiteren Verlauf des Skills benötigt werden, so können diese Daten in der Session gespeichert und später ausgelesen werden.

Die zweite in Listing 3.24 dargestellte Möglichkeit ist das Setzen des Inhalts eines Session-Objekts. Hier wird als Beispiel die Sprache des Alexa-Nutzers in der Session zur weiteren Verwendung gespeichert. Es wäre denkbar, dass die Sprache zuvor vom Alexa-Nutzer im selben Intent innerhalb eines Slots angegeben wurde. Wie in der Beschreibung der Skill-DSL schon erwähnt, existiert die Möglichkeit, gegebene Slot-Inhalte direkt in Objekten der Session zu speichern, ohne den alternativen Umweg über die Implementierungsklasse gehen zu müssen.

Bei der Nutzung von komplexen DIME-Daten verhält es sich konstruktionsbedingt ein wenig anders mit den Kantenarten. Hier existieren zwar auch die unterschiedlichen Kantenarten, jedoch wird nicht verhindert, dass wenn ein Skill-Entwickler per get-Funktion einen komplexen DIME-Datentyp bekommt und Änderungen daran vornimmt, dass diese Änderungen automatisch persistent gespeichert werden, obwohl so nicht vorgesehen. Hierbei müsste im Ausblick als möglicher Lösungsansatz zusätzlich das gesamte DIME-Datenobjekt zunächst kopiert werden, bevor der Nutzer es in der Klasse benutzen könnte, damit Änderungen daran nicht direkt gespeichert werden.

3.5.4 Responses-Parameter

Der dritte Parameter der Funktion, die der Skill-Entwickler programmieren muss, betrifft das sogenannte Responses-Objekt. Das Responses-Objekt ist ein Verwaltungsobjekt für die im Graphmodell hinter dem Server-Objekt angelegten Responses. Für jede mögliche Response, die nach dem Server-Knoten folgt, existiert eine Methode in der Impl-Klasse.

Diese Methode, die ein Response-Objekt des Graphmodells darstellt, bietet verschiedene Optionen. Wichtig ist, dass der Entwickler eine der Responses aus dem `responses`-Parameter am Ende der Methode zurückgibt. Dabei soll es sich um die entsprechende Response handeln, die der Alexa-Nutzer erhalten soll. Ohne auf die technischen Details weiter einzugehen, existieren in einem Response-Objekt innerhalb des Response-Verwaltungs-Objektes mehrere Möglichkeiten, wie die Antwort des Alexa-Geräts an den Alexa-Nutzer aussehen soll. Die wichtigsten Funktionen hierbei sind `speech` und `repromptText`.

Darüber hinaus gibt es noch verschiedene Optionen für die graphische Darstellung auf Echo-Geräten mit Bildschirmen. Beim `speech`-Objekt handelt es sich um das, was das Alexa-Gerät dem Alexa-Nutzer akustisch sagen soll. Der Alexa-Nutzer hat nach einer Antwort von Alexa acht Sekunden Zeit, um eine nächste Aktion auszuführen. Wird die Bedenkzeit überschritten, gibt es einen Session-Timeout und der Skill wird von Amazon aus beendet. Um ein Timeout zu vermeiden, kann ein sogenannter *Reprompt* verwendet werden.

Der Reprompt, also die Nachfrage, wird von Alexa an den Alexa-Nutzer nach diesen acht Sekunden gestellt, woraufhin der Alexa-Nutzer wiederum erneut acht Sekunden Zeit hat, mit dem Skill zu interagieren. Es wird genau einmal ein Reprompt an den Alexa-Nutzer gestellt. Beide Angaben und die Angaben für die grafische Darstellung müssen vom Skill-Entwickler für jede Response angegeben werden. Es ist hierbei möglich, für unterschiedliche Response-Objekte auf dem Graphmodell auch unterschiedliche Texte zu spezifizieren, was Alexa dem Alexa-Nutzer sagen soll. Hierfür muss der Skill-Entwickler der Impl-Klasse jedoch in einer programmierten Entscheidungsfindung bestimmen, welche der zur Auswahl stehenden Responses an den Alexa-Nutzer gesendet werden soll. Je nachdem welche Response der Skill-Entwickler mit seiner Programmierung ausführen lässt, bekommen die Handler-Klassen der Intents die gewählte Response mit und erhalten dadurch die Information, welcher weitere Pfad im Graphmodell für den Alexa-Nutzer möglich ist. Ein Beispiel der Response befindet sich in Listing 3.22.

3.5.5 Speech Synthesis Markup Language

Wie bereits in den Grundlagen beschrieben, existiert die Möglichkeit der Nutzung von SSML. Im Kontext der Implementierungsklassen steht SSML dem Skill-Entwickler zur Verfügung. Folgend wird mit einem Beispiel der Nutzung von SSML in Implementierungsklassen die Nutzung von SSML veranschaulicht.

Beispiel der SSML-Nutzung

Der Großteil der SSML-Optionen befinden sich in dem Speech-Objekt vorgefertigt in der Impl-Klasse. So kann der Skill-Entwickler eine Konkatenation von unterschiedlichen SSML-Optionen in einer gesamten Aussage eines Alexa-Geräts vornehmen. Es ist beispielsweise möglich, dass der erste Teil der Antwort mit der Standard-Alexa-Stimme gesprochen wird und der Rest mit der italienischen Stimme *Giorgio*. Natürlich ist es auch möglich Teile eines Satzes mit bestimmten Effekten zu versehen. Diese und weitere SSML-Optionen befinden sich in dem Speech-Objekt. Das Speech-Objekt beinhaltet also die eigentliche Antwort des Geräts an den Alexa-Nutzer und muss dem Response-Objekt übergeben werden.

Das Listing 3.25 veranschaulicht die Nutzung der SSML-Optionen innerhalb der Implementierungsklasse. Das konkrete Beispiel hat zur Folge, dass die Alexa-Geräte in ihrer Sprache zunächst das Wort „Hello“ in ihrer Standardstimme sagen. Als Nächstes wird eine Pause von 60 Millisekunden eingelegt. Anschließend wird der Akzent durch das Setzen der `language`-Option auf Französisch gesetzt. Da die `language`-Option in diesem Beispiel keinen Text-Parameter zusätzlich übergeben bekommt, werden alle folgenden Aussagen mit französischem Akzent ausgesprochen. Die Alternative wäre hier, neben der Angabe des Akzents auch den Text mit als Parameter zu übergeben. Dann würde nur dieser Text mit dem gewählten Akzent ausgesprochen. Das Wort „World“ wird durch die Nutzung der `whisper`-Option geflüstert und durch die `language`-Option erfolgt dies mit einem französischen Akzent. Als vorletztes wird in dem Beispiel eine weitere Pause von zwei Sekunden eingelegt, um abschließend mit starker Betonung und weiterhin französischem Akzent „Whats up?“ zu fragen.

```
1 Speech speech = new Speech();
2 speech.say("Hello")
3       .pause(60)
4       .language(Language.FR)
5       .whisper("World.")
6       .pause(120)
7       .emphasis("Whats up?", Level.STRONG);
```

Listing 3.25: Beispiel der SSML-Optionen

3.5.6 Beispiel einer Impl-Klasse

In Listing 3.26 werden alle wichtigen Funktionen einmal exemplarisch in der Implementierungsklasse benutzt. Zunächst werden Daten ausgelesen, welche in der Session hinterlegt sind. Das Beispiel zeigt auch, wie Alexa-Nutzer-Angaben aus den Slots ausgelesen werden können. Durch eine Liste `questionsPlayed` wird auch verdeutlicht, wie Daten in eine Session geschrieben werden können. Da dieses Beispiel im Graphmodell zwei unterschiedliche Responses nach einem Server-Knoten besitzt, findet eine `if`-Abfrage statt, die entscheidet, welche der zur Verfügung stehenden Responses wieder zurückgegeben werden soll. Das Beispiel wurde an dieser Stelle gekürzt.

```
1  // Eine gestellte Frage aus der Session auslesen
2  Integer questionId = session.getCurrentQuestion();
3
4  // Antwort der Frage aus der Session auslesen
5  String expectedAnswer = session.getAnswers().get(questionId);
6
7  // Antwort des Alexa-Nutzers auslesen aus den Slot des Intents
8  String userAnswer = slots.getAnswer();
9
10 // Eine neue gestellte Frage merken
11 List<Integer> questionsPlayed = new LinkedList<Integer>();
12 questionsPlayed.add(questionId);
13 session.setQuestionsPlayed(questionsPlayed);
14
15 // Beispiel des Antwortobjekts
16 Speech speech = new Speech();
17 speech.say("Richtig. Aktueller Score: " + session.getCurrentHighScore().toString()
18            + " Weitere Frage: " + nextQuestion)
19
20 // Beispiel einer richtigen Antwort eines Alexa-Nutzers
21 if (expectedAnswer.toLowerCase().equals(userAnswer.toLowerCase())) {
22
23 // Highscore hochsetzen
24 session.setCurrentHighScore(session.getCurrentHighScore() + 1);
25
26 // Response-Objekt bauen
27 responses.getQuestion_Response()
28          .setSpeech(speech)
29          .setDisplayTitle("Richtig")
30          .setDisplayText("Richtig, weitere Frage: " + nextQuestion);
31 }
```

Listing 3.26: Beispiel einer Implementierung

Kapitel 4

Codegenerierung

Im folgenden Abschnitt wird, aufbauend auf der im vorherigen Kapitel vorgestellten Ebene der Modellierung von Alexa-Skills, der Generierungs-Workflow beschrieben. Dabei wird jeweils auf die Funktionsweise und den Output der Generatoren für die einzelnen vorgestellten Modelltypen eingegangen sowie das Zusammenspiel der jeweiligen Generatoren näher erläutert.

Bei der Generierung wird dabei zwischen zwei grundlegenden Ansätzen unterschieden: Der Generierung eines Alexa-Skills ohne Integration von Dime-Artefakten und der Generierung mit diesen Artefakten (Dime-Ansatz). Der Generator überprüft dafür, ob die ASW-Datei, die den Einstieg für die den Großteil der Generatoren bildet, in einem Projekt mit Dime-Dateien (Dime-Projekt) liegt oder nicht. Ist dies der Fall wird vor dem Start der Generatoren des Alexa Skill Development Tool, zunächst der Dime-Generator angestoßen, um sicherzustellen, dass Dime-Generate, welche die Grundlage für Alexa-spezifische Generate bilden, vorhanden sind. Liegt die ASW-Datei nicht in einem Dime-Projekt, legt der Generator ein Maven-Projekt an, in welches die Generatoren generieren.

Starten der Generierung

Sämtliche Generatoren des Alexa Skill Development Tools werden zentral durch einen Klick auf das Generate-Symbol (siehe Abbildung 4.1) angestoßen, einzige Ausnahme bildet der Generator für die Manifest-Datei, welcher separat vor dem Deployment des Alexa-Skills angestoßen wird.

Abbildung 4.1: Teil der ECLIPSE-Toolbar mit Generierungs-Icon „G“

Erfüllt der Skill-Entwickler die Voraussetzungen zur Generierung des Projekts auf Basis der Skill-DSL und der Intents-DSL, so kann der Generierungsprozess starten und alle benötigten JAVA-Klassen werden generiert. Im Folgenden wird der Generierungsprozess genauer beschrieben, welcher durch das Klicken des Generierungs-Icon angestoßen wird.

4.1 Modelltransformation und JSON-Generierung

Zu Beginn des Generierungsprozesses wird die JSON-Datei zur Beschreibung eines Interaction Models generiert. Diese JSON-Datei ist für das Deployen und das Starten eines Alexa-Skills notwendig. Für die Generierung dieser JSON-Datei werden die in Abschnitt Abschnitt 3.3 beschriebenen Intents-Dateien genutzt. Dabei besteht der Generierungsprozess im Wesentlichen aus zwei Schritten: Der Transformation des Interaction Models der Intents-Datei und des graphischen Modells sowie der eigentlichen Generierung der JSON-Datei.

Im Zuge des Generierungsprozesses erfolgt eine Modelltransformation (skizziert in Abbildung 4.2) auf Grundlage des Modells der Skill-DSL (siehe Abschnitt 3.2). Aus allen dort referenzierten Intents und dem angegebenen Invocation Name wird ein neues Interaction Model aufgebaut. Die referenzierten Intents im graphischen Modell können hierbei möglicherweise aus verschiedenen Intents-Dateien stammen und werden im Zuge der Transformation zu einem neuen Interaction Modell zusammenfügt.

Motivation für die Modelltransformation ist zum einen das Sicherstellen von eindeutigen Slot- und Prompt-Namen innerhalb der jeweiligen Parent-Intents, welche von Amazon verlangt werden. Zum anderen wird durch die Transformation ein Interaction Model erstellt, welches eine Eins-zu-Eins-Repräsentation des JSON-Modells darstellt und damit die Generierung der JSON-Datei vereinfacht. Dies ist möglich, da das in Unterabschnitt 3.3.1 beschriebene Metamodell für die Intents-Sprache das JSON-Modell von Amazon repräsentiert, aber es zusätzlich um neue nutzerfreundliche Konzepte, wie die Wiederverwendbarkeit von Slots und Prompts, erweitert. Im Folgenden soll nun detaillierter auf die Transformation, welche während des Generierungsprozesses erfolgt, eingegangen werden.

Für die bestehenden Instanzen der Intents-DSL und der Skill-DSL werden neue, nur für die Generierung verwendete Instanzen erstellt. Die Transformation und Neuerstellung der Modellinstanzen bietet den Vorteil, dass die ursprünglichen Instanzen nicht manipuliert werden müssen. Dadurch bleiben die Darstellungen im Editor und ihre Repräsentationen im Speicher konsistent. Die Abbildung 4.2 veranschaulicht dabei den Transformationsvorgang: Die Dateien, deren Informationen zur Transformation genutzt werden, sind dabei die Datei des graphischen Modells sowie die Intents-Dateien, aus denen Intents und Slots referenziert werden.

Auf Grundlage dieser Informationen wird ein neues Interaction Model (siehe Unterabschnitt 3.3.1) erstellt und mit den Intents und ihren Slots, die in dem graphischen Modell referenziert werden, angereichert. Dazu gehören neben Intents und Slots auch alle Prompts, Types und sonstige Abhängigkeiten, die transitiv durch die Referenzierung entstehen. Im Rahmen dessen bekommen lokale Slots auch eindeutige Namen, die sich aus dem Namen des Parent Intents und dem Namen des Slots zusammensetzen (`ParentName_SlotName`). Ebenso wird auch für Prompts ein neuer Name generiert, der die Form `IntentName_PromptName` beziehungsweise `IntentName_SlotName_PromptName` hat. Sofern ein Prompt lokal definiert ist und somit keinen benutzerdefinierten Namen hat, wird stattdessen an den Intent- beziehungsweise Slot-Namen das Präfix `_ElicitationPrompt` oder `_ConfirmationPrompt` angehangen. Sämtliche lokale Prompts werden im Zuge der Transformation als globale Prompts angelegt und entsprechend von den Slots oder Intents, in denen sie bisher lokal definiert waren, referenziert. Umgekehrt fallen globale Slots im neu erstellten Interaction Model weg. Globale Slots, die von Intents referenziert werden, werden in den entsprechenden Intents als neue lokale Slots angelegt.

Da die Namen von Slots und Prompts in der am Ende generierten JSON-Datei nicht mehr übereinstimmen, mit denen, die in der Intents-Datei definiert wurden, muss nach der Generierung der JSON-Datei noch das graphische Modell transformiert werden. Dafür wird eine neue Instanz des graphischen Modells erstellt und sämtliche Information aus dem graphischen Modell in diese neue Instanz kopiert. Auf dieser Kopie werden nun die Referenzen auf Intents und Slots durch entsprechende Referenzen auf die Intents und Slots des transformierten Interaction Models ausgetauscht. Dadurch wird sichergestellt, dass im weiteren Verlauf des Generierungsprozesses die Referenzen des graphischen Modells auf Intents und Slots korrekt sind. Auf Basis dieses transformierten graphischen Modells kann der Generierungsprozess fortgeführt werden.

Nachdem die Transformation des Modells stattgefunden hat, kann daraus die JSON-Datei generiert werden. Zur Generierung der benötigten Interaction-Model-JSON-Datei wurde das JAVA-Paket `javax.json` [37] verwendet. Es erlaubt eine objektbasierte Konstruktion von JSON-Dateien. Die Struktur der JSON-Objekte basiert auf den Vorgaben der Alexa-

API (siehe Unterabschnitt 2.7.2). Sie werden nacheinander generiert und mit Informationen aus dem transformierten Interaction Model angereichert. Anschließend wird mithilfe des `build`-Befehls aus den JSON-Objekten eine String-Repräsentation generiert, die als Interaction-Model-JSON-Datei gespeichert wird. Das Listing 4.1 zeigt dabei wie das generierte JSON für das Quizbeispiel aussieht.

```
{ "interactionModel": {
    "languageModel": {
      "invocationName": "interactionmodel",
      "intents": [
        { "name": "AMAZON.CancelIntent",
          "samples": [
            "Halt",
            "Aufhören",
            "Beende das Spiel",
            "Beende das Quiz"
          ] },
        { "name": "CreateHighscoreRequest",
          "slots": [
            { "name": "CreateHighscoreRequest_decision",
              "type": "Decision",
              "samples": [ "{CreateHighscoreRequest_decision}" ] }
          ],
          "samples": [ "Dummy Keyphrase Zwei" ] },
        { "name": "GetAnswerIntent",
          "slots": [
            { "name": "GetAnswerIntent_answer",
              "type": "Answer",
              "samples": [
                "Die Antwort heißt  {GetAnswerIntent_answer}",
                "Die Antwort lautet  {GetAnswerIntent_answer}",
                "{GetAnswerIntent_answer}"
              ] }
          ],
          "samples": [ "Dummy Keyphrase Eins" ] },
        { "name": "SetNameIntent",
          "slots": [
            { "name": "SetNameIntent_name",
              "type": "AMAZON.FirstName",
              "samples": [
                "Ich heiße {SetNameIntent_name}",
                "Mein Name ist {SetNameIntent_name}",
                "{SetNameIntent_name}"
              ] }
          ],
          "samples": [
            "Ich heiße {SetNameIntent_name}",
            "Mein Name ist {SetNameIntent_name}",
            "{SetNameIntent_name}"
          ] }
      ],


```

```
48
49      "types": [
50        { "name": "Decision",
51          "values": [
52            { "id": "Ja",
53              "name": {
54                "value": "Ja",
55                "synonyms": [ "Okay", "In Ordnung", "Na gut" ] } },
56            { "id": "Nein",
57              "name": {
58                "value": "Nein",
59                "synonyms": [ "Ne", "Nö", "Nope" ] } }
60          ] },
61        { "name": "Answer",
62          "values": [
63            { "id": "Dummy_Type_Value",
64              "name": { "value": "Dummy Type Value" } }
65          ] }
66      ] },
67
68    "dialog": {
69      "intents": [
70        { "name": "CreateHighscoreRequest",
71          "confirmationRequired": false,
72          "slots": [
73            { "name": "CreateHighscoreRequest_decision",
74              "type": "Decision",
75              "confirmationRequired": false,
76              "elicitationRequired": true,
77              "prompts": {
78                "elicitation":
79          "CreateHighscoreRequest_decision_ElicitationPrompt" } }
80          ] },
81        { "name": "GetAnswerIntent",
82          "confirmationRequired": false,
83          "slots": [
84            { "name": "GetAnswerIntent_answer",
85              "type": "Answer",
86              "confirmationRequired": false,
87              "elicitationRequired": true,
88              "prompts": {
89                "elicitation": "GetAnswerIntent_answer_answerPrompt" } }
90          ] },
91        { "name": "SetNameIntent",
92          "confirmationRequired": false,
93          "slots": [
94            { "name": "SetNameIntent_name",
95              "type": "AMAZON.FirstName",
96              "confirmationRequired": false,
97              "elicitationRequired": true,
98              "prompts": {
99                "elicitation": "SetNameIntent_name_ElicitationPrompt" } }
100         ] }
101     ] },
102
```

```
103     "prompts": [
104       { "id": "CreateHighscoreRequest_decision_ElicitationPrompt",
105         "variations": [
106           {"type": "PlainText",
107             "value": "Soll ein Highscoreeintrag angelegt werden?" }
108         ] },
109       { "id": "GetAnswerIntent_answer_answerPrompt",
110         "variations": [
111           { "type": "PlainText",
112             "value": "Was ist die richtige Antwort?" },
113           { "type": "PlainText",
114             "value": "Wie lautet die Antwort?" },
115           { "type": "PlainText",
116             "value": "Weißt dus?" }
117         ] },
118       { "id": "SetNameIntent_name_ElicitationPrompt",
119         "variations": [
120           { "type": "PlainText",
121             "value": "Wie lautet der Spielername?" },
122           { "type": "PlainText",
123             "value": "Mit welchem Spielernamen wird gespielt?" }
124         ] }
125     ] } }
```

Listing 4.1: Generiertes JSON für das Quizbeispiel (Vergleich zur Intents-Datei aus Listing 3.1)

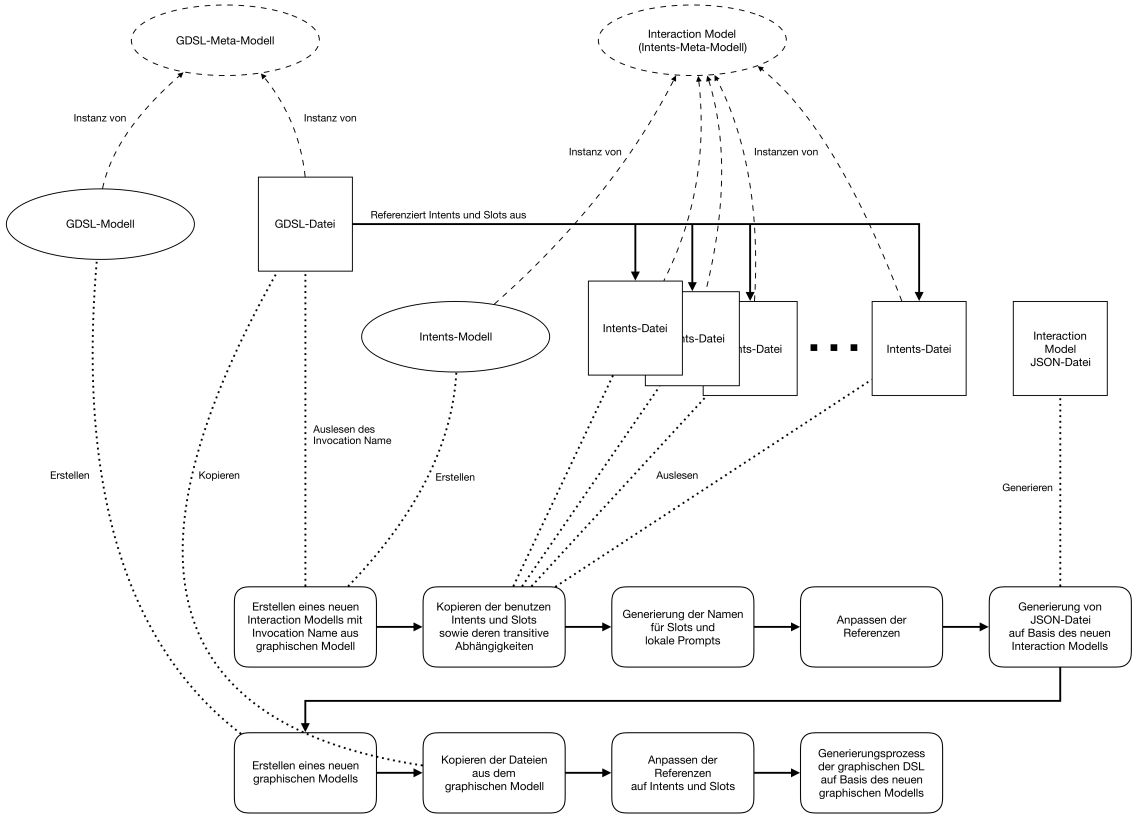


Abbildung 4.2: Ablauf des Modelltransformierungs- und Generierungsprozesses

4.2 Generierung der Projektstruktur

Nach der Modelltransformation werden die weiteren Bestandteile des ASW-Projekts generiert. Je nach gewähltem Ansatz, ob mit oder ohne DIME-App, ergeben sich feine Unterschiede im automatisierten Prozess der Generierung. Beim DIME-Ansatz wird der vom ASDT generierte Endpoint im Backend der DIME-App integriert. Dazu muss das DIME-Projekt, wie in Unterabschnitt 4.2.1 beschrieben, entsprechend angepasst werden. Bei der Generierung ohne DIME-Ansatz wird hingegen ein eigenständiges MAVEN-Projekt generiert, das den Endpoint implementiert. Die Besonderheiten des MAVEN-Projekts werden in Unterabschnitt 4.2.2 näher erläutert. Der interne Aufbau ist in beiden Ansätzen jedoch gleich und wird anschließend in den Unterabschnitten 4.2.3 bis 4.2.8 aufgeführt.

4.2.1 Integration des Alexa-Skills in ein DIME-Projekt

Befinden sich die ASW-Dateien in einem DIME-Projekt, so wird beim Starten des Generierungsprozesses zunächst der DIME-Generator angestoßen. Das DIME-Generat befindet sich anschließend im `target`-Ordner des Projekts. Es enthält ein MAVEN-Projekt (`dywa-app`), welches aus mehreren Modulen (darunter `app-presentation`) besteht. Das Generat des ASDT wird ein eigenes Modul (`app-alexa`) in diesem MAVEN-Projekt sein.

Dazu wird der Ordner `app-alexa` in `dywa-app` erzeugt und eine `pom.xml`[1] generiert. Sie enthält die benötigten Abhängigkeiten (*Dependencies*) des `app-alexa`-Moduls und Instruktionen für den MAVEN-Compiler. So werden beispielsweise die benötigten JAR-Dateien des ASK-SDK zur Kompilierzeit geladen und stehen für den generierten JAVA-Code zur Verfügung. Des Weiteren müssen noch Änderungen an den von DIME generierten Modulen vorgenommen werden. Da alle davon betroffenen Dateien XML-Dateien sind, geschieht dies mittels XML-Transformatoren. Es wird `app-alexa` als Modul in der `pom.xml`[2] der `dywa-app` und als Dependency zur `pom.xml`[3] von `app-presentation` hinzugefügt.

Letzteres hat den Hintergrund, dass `app-presentation` für das Frontend der DIME-App zuständig ist. Der Endpoint für den Alexa-Skill wird durch das Frontend von DIME zur Verfügung gestellt. Dazu muss das Skill-Servlet, das als zentrale Anlaufstelle für Alexa-Anfragen dient (siehe Unterabschnitt 4.2.5), zu der `web.xml`[4] von `app-presentation` hinzugefügt werden. Im letzten Schritt vor der Generierung des restlichen Quellcodes wird der `urlrewrite.xml`[5] von `app-presentation` eine neue Regel hinzugefügt. Sie verhindert, dass

[1] `<Projekt>/target/dywa-app/app-alexa/pom.xml`
[2] `<Projekt>/target/dywa-app/pom.xml`
[3] `<Projekt>/target/dywa-app/app-presentation/pom.xml`
[4] `<Projekt>/target/dywa-app/app-presentation/src/main/webapp/WEB-INF/web.xml`
[5] `<Projekt>/target/dywa-app/app-presentation/src/main/webapp/WEB-INF/urlrewrite.xml`

die URL, unter die der Alexa-Skill verfügbar sein wird, nicht durch das DIME-Frontend umgeleitet wird. Der in den Unterabschnitten 4.2.3 bis 4.2.8 beschriebenen Quellcode kann schließlich in den `app-alexa`-Ordner generiert werden.

4.2.2 Besonderheiten bei Generierung ohne DIME-Ansatz

Als Alternative zu dem DIME-Ansatz steht dem Benutzer die Möglichkeit zur Verfügung, ein eigenständiges MAVEN-Projekt generieren zu lassen, welches dann später im Rahmen einer WAR-Datei auf einen Webserver geladen werden kann. Der erste Schritt dieser Alternative ist das Erzeugen eines MAVEN-Projekts. Die WAR-Datei dient später als Endpoint für die Kommunikation mit Amazon. Bei Wahl dieser Möglichkeit wird das Anlegen des hierzu benötigten MAVEN-Projekts durch das von der PG entwickelte ASDT automatisiert.

Mit dem MAVEN-Projekt werden automatisch eine `pom.xml` und eine `web.xml` generiert. Beide Dateien sind wichtig für das MAVEN-Projekt und die spätere WAR-Datei. Sie beinhalten vorkonfigurierte Werte, die teilweise auf den Angaben des Skill-Entwicklers in der Skill-DSL beruhen. So wird beispielsweise später die generierte WAR-Datei nach dem Skill-Projekt benannt, was individuell für jedes Projekt ist.

Eine weitere erwähnenswerte Eigenschaft der Generierung des MAVEN-Projekts ist das automatische Laden der Abhängigkeiten aus der `pom.xml`. Der Skill-Entwickler muss nicht das Laden der Dependencies über die `pom.xml` manuell anstoßen. Stattdessen geschieht dies bei vorhandener Internetverbindung automatisch im Generierungsschritt des MAVEN-Projekts. Die benötigten Bibliotheken des ASK-SDK stehen nach dem Generieren somit sofort zur Verfügung. Die generierten Einträge der `pom.xml` sollten bestenfalls nur additiv angepasst werden. Bei der `web.xml` können die initialen Grundkonfigurationen beliebig angepasst werden, mit der Ausnahme der Angabe des Servlets.

4.2.3 Ordnerstruktur der generierten JAVA-Klassen

Folgend wird weiter auf die generierte Struktur eingegangen. Die beiden wichtigsten generierten Ordner sind `src` und `src-gen`. In ihnen werden initial JAVA-Pakete und -Klassen generiert.

Abweichend davon heißen in einer DIME-Infrastruktur die Ordner `alexa-src` und `alexa-src-gen`, wobei sie die gleiche Funktionalität aufweisen. Folgend werden diese zur Verbesserung des Leseflusses ebenfalls mit `src` und `src-gen` bezeichnet.

Beide Ordner, `src` und `src-gen`, unterscheiden sich in ihrer Bedeutung für den Skill-Entwickler. Alle anzupassenden Klassen befinden sich im `src`-Ordner mit beispielhaft vorgegebener Standardimplementierung. Im `src-gen`-Ordner hingegen befinden sich Klassen, die der Skill-Entwickler einerseits nicht editiert und die andererseits bei einer erneuten Generierung des Projekts überschrieben werden.

Im `src`-Ordner befinden sich die für den Skill-Entwickler wichtigen Implementierungsklassen. Diese können entweder über einen Doppelklick auf die Server-Objekte in der Skill-DSL geöffnet werden oder alternativ über Navigation in die Projektstruktur. Diese Implementierungsklassen haben darüber hinaus die Eigenschaft, dass sie nach erstmaliger Generierung nicht mehr durch den Generator überschrieben werden, so dass Änderungen an diesen, auch bei erneuter Generierung, erhalten bleiben. Dies bedeutet aber auch, dass die Klassen nicht einfach gelöscht werden sollten, da hier individuelle Benutzeranpassungen hinein geschrieben werden. Der Aufbau und weitere technischen Details hinter den Implementierungsklassen werden in Abschnitt 3.5 erläutert.

Sollten auf der Skill-DSL auch DIME-Prozesse verwendet werden, so werden zudem sogenannte Prozess-Klassen erzeugt, die in ihrer Funktion als Schnittstelle zwischen den Handlern und den eigentlichen DIME-Prozessen gelten. Hierauf wird auch in einem eigenen Kapitel näher eingegangen, siehe Unterabschnitt 4.2.9.

4.2.4 Generierte JAVA-Pakete

Im Folgenden wird kurz auf die generierten JAVA-Klassen innerhalb des `src-gen`-Ordners eingegangen, ohne weiter auf die technischen Details der Umsetzung einzugehen. Der `src-gen`-Ordner beinhaltet folgende generierte Paketstruktur:

- `skill`
- `skill.handlers`
- `skill.handlers.userinterfaces`
- `skill.handlers.userinterfaces.impl.response`
- `skill.handlers.userinterfaces.impl.session`
- `skill.handlers.userinterfaces.impl.slots`
- `skill.processes`

Das Paket skill.processes wird erzeugt, unabhängig davon, ob das ASW-Projekt in einer DIME-App integriert wird oder nicht. In Fällen wie diesen, dass kein DIME-Prozess benutzt werden kann, da das ASW-Projekt nicht in eine DIME-App integriert wurde, bleibt das generierte Paket leer. Generell dienen dort hinein generierte Klassen als Schnittstelle zwischen dem ASW-Projekt und dem Aufruf eines DIME-Prozesses.

4.2.5 Servlet- und Handler-Klassen

Da auf dem JAVA ASK SDK aufgebaut wird, wird als erste Anlaufstelle für Anfragen eines Skills an den Server eine sogenannte Servlet-Klasse benötigt. Diese nimmt Anfragen entgegen und verteilt sie an die entsprechenden RequestHandler, folgend vereinfacht als Handler bezeichnet. Die Servlet-Klasse befindet sich im `skill`-Paket. Die als nächstes angesprochenen Handler, welche prüfen, ob sie die empfangene Skill-Anfrage ihrerseits bedienen können, befinden sich hingegen im `skill.handlers`-Paket. Der allgemeine Ablauf einer solchen Anfrage ist in Abbildung 4.3 dargestellt.

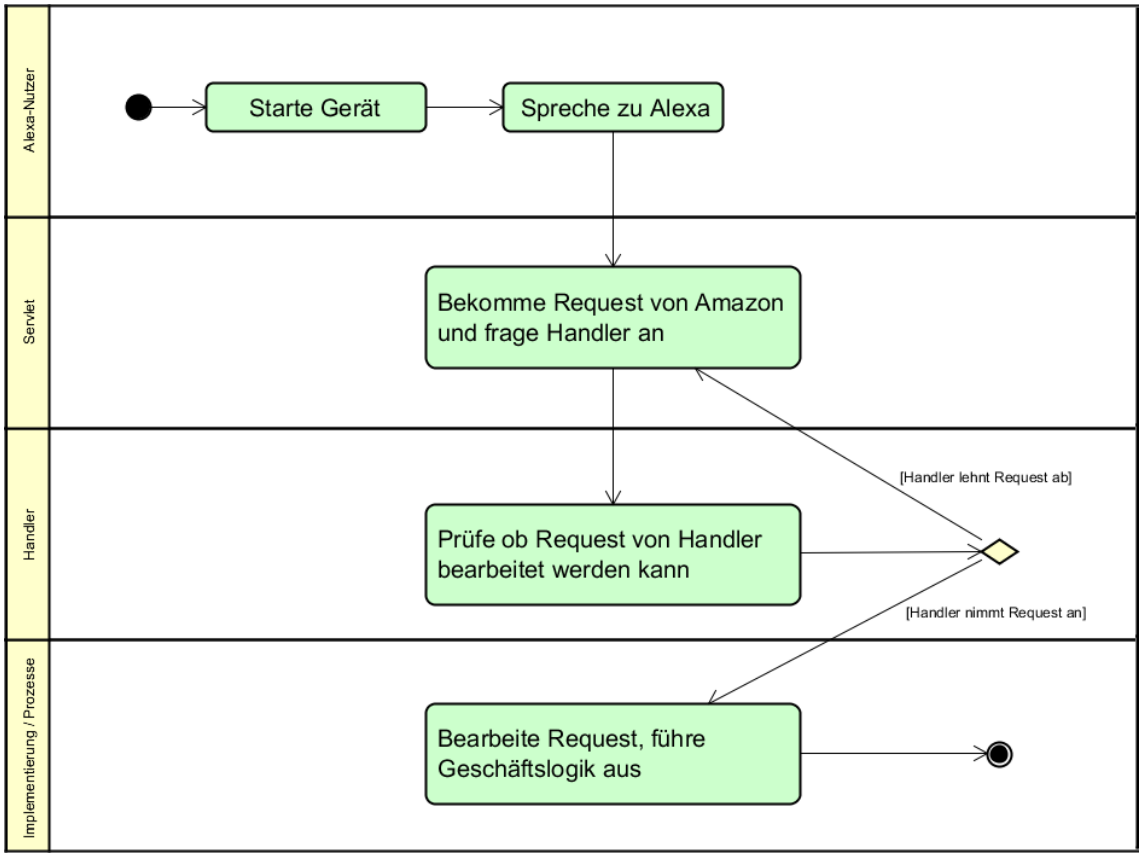


Abbildung 4.3: Aktivitätsdiagramm zur Verarbeitung eines Requests

Es existiert pro Server-Knoten und pro Prozess-Knoten der Skill-DSL je eine Handler-Klasse. Darüber hinaus gibt es generierte Standard-Handler, wie beispielsweise den `ErrorHandler`. Dieser greift als letztes, wenn keiner der anderen Handler die Skill-Anfrage an den Server beantworten kann. Der semantische Zusammenhang zwischen Handler und Intent ist, dass die Handler die JAVA-Klassen sind, welche die Logik des Intents ausführen. Hier wird auch der Zusammenhang deutlich, dass der Skill-Entwickler bei Benutzung von Server-Knoten auf der Skill-DSL die Logik des Intents manuell in den Implementierungsklassen programmieren muss. In den Handlern findet auch die wichtige Eigenschaft der Pfadprüfung statt. Es wird somit nicht nur geprüft, ob ein Alexa-Nutzer einen Intent angesprochen hat, sondern auch, ob der Alexa-Nutzer den Intent hat zu diesem Zeitpunkt überhaupt ansprechen dürfen. Spricht der Alexa-Nutzer einen Intent an und die Anfrage kommt vom Servlet an den Handler, so erkennt der Handler des Intents zwar, dass der Intent angesprochen wurde, wenn dieser zu dem gegebenen Zeitpunkt jedoch noch nicht für den Alexa-Nutzer ansprechbar ist, so reagiert der Handler nicht. Sollte kein Handler reagieren, aus welchen Gründen auch immer, reagiert spätestens der `ErrorHandler` und der Alexa-Nutzer bekommt die Rückmeldung, dass er die getätigte Aktion zu diesem Zeitpunkt nicht ausführen durfte.

4.2.6 State-Variable

Wie bereits im vorangegangenen Abschnitt beschrieben, werden einem Servlet mehrere RequestHandler hinzugefügt, die auf angesprochene Intents reagieren können. Hierzu wird überprüft, ob ein angesprochener Intent denselben Namen hat, wie ein in dem Modell referenzierter vorhergehender Intent-Knoten. Allerdings kann derselbe Intent in einem Modell mehrfach vorkommen, wie es beispielhaft in Abbildung 4.4 dargestellt wird. Deshalb musste eine Methode entwickelt werden, sodass ein Handler nur auf ein Intent reagiert, wenn zuvor der entsprechende Pfad im Modell gegangen wurde. Hierzu wurde eine Pfadprüfung beziehungsweise Zustandsüberprüfung mittels einer im Hintergrund gespeicherten State-Variable implementiert.

Die *State-Variable* (SV) wird abhängig von einer gewählten Response im Handler auf einen einzigartigen zu einem Verzweigungsart-spezifischem Knoten korrespondierenden Wert gesetzt. In diesem Fall werden zwei Arten von Verzweigungen gesondert betrachtet, die in Abbildung 4.5 abgebildet sind. Falls nach einer Response ein User-Knoten folgt, wird zum Beispiel die SV des User-Knotens gesetzt und im Falle des Intent-Chainings (siehe Abschnitt 3.2.1) wird die SV direkt über den gewählten Response-Knoten gesetzt. Wenn zum Beispiel im *CheckReadyHandler* die Response *Not Ready* gewählt wird, wird die SV auf den mit dem folgenden User-Knoten korrespondierendem Wert gesetzt. Der nächste an-

gesprochene Intent wird dann bereits deshalb nicht von dem *AnswerValidationHandler* verarbeitet, da nicht die benötigte SV gesetzt wurde.

Ein Sonderfall bildet das Starten eines Skills. Hier muss ein Handler entweder auf einen Invocation- oder einen zusätzlichen Intent-Request reagieren, wenn eventuell noch keine SV gesetzt wurde. Während der Handler dann ausgeführt wird, setzt er die SV nach dem gerade beschriebenen Verfahren.

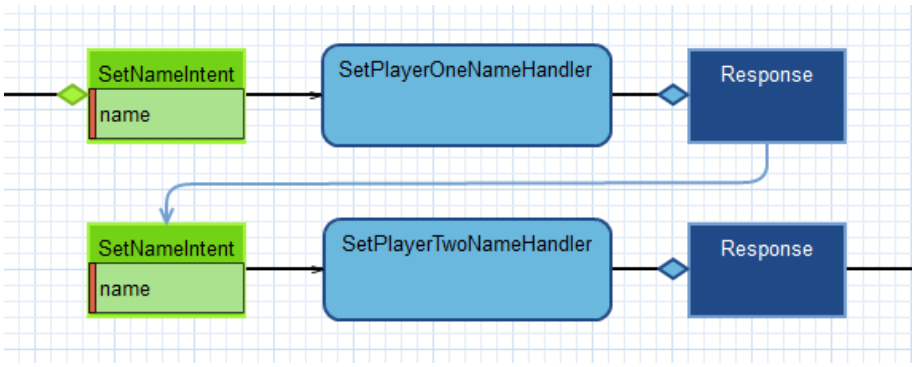


Abbildung 4.4: Beispiel zur Mehrfachverwendung desselben Intents

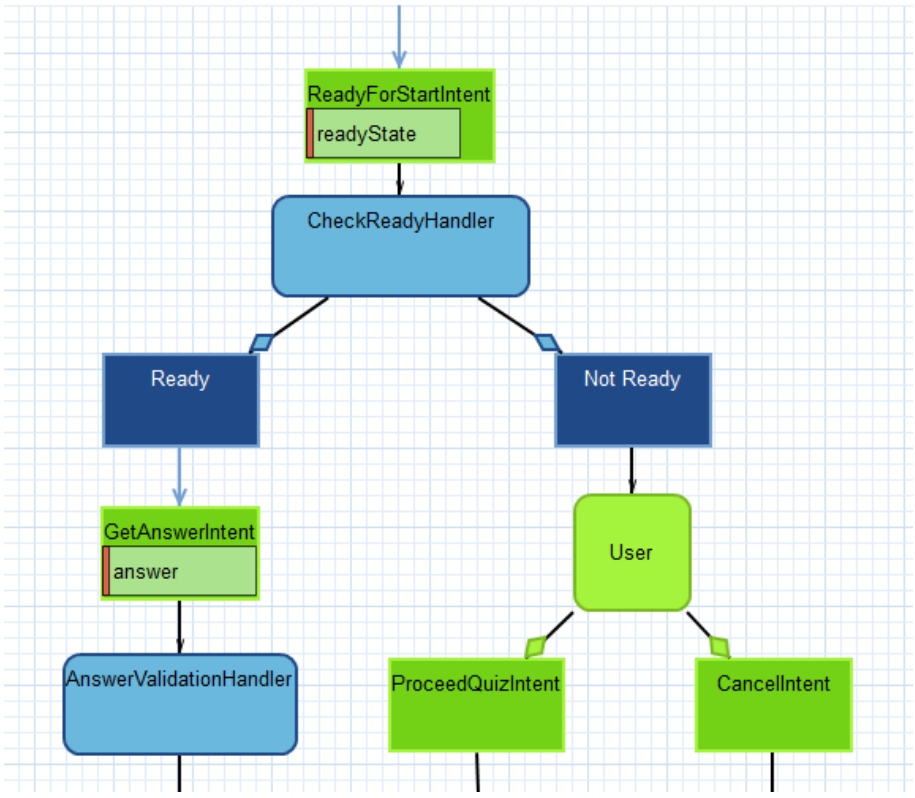


Abbildung 4.5: Beispiel zur Verzweigung über Intent-Chaining (links) und User-Knoten (rechts)

4.2.7 Slot- und Session-Klassen

Die weiteren Pakete, nach dem `skill`-Paket mit der Servlet-Klasse und dem `skill.handlers`-Paket mit den Handlern der Intents, bieten für die Implementierungsklassen des Skill-Entwickler unterstützende Funktionen. Folgend wird näher auf die Slots und die Session eingegangen.

Der Skill-Entwickler wird später in der Implementierung die Möglichkeit haben, auf die Slots zuzugreifen, welche der Alexa-Nutzer mit seinen Angaben belegt hat. Dafür muss der Skill-Entwickler keinen eigenen Code schreiben. Die dafür benötigte Architektur befindet sich im `slots`-Paket. Dort stehen für jeden Handler und damit Intent des Graphmodells die Slots für die Implementierung zur Verfügung.

Wird der Skill-Entwickler später die Inhalte der Slots auslesen, wie beispielsweise den abgefragten Namen des Alexa-Nutzers, so wird technisch die Anfrage an die entsprechende `Slots`-Klasse des Intents innerhalb des `slots`-Pakets geleitet. Die innerhalb der `Slots`-Klassen generierten `get`-Funktionen bieten einerseits dem Skill-Entwickler den Mehrwert, die vom Alexa-Nutzer bereitgestellten Informationen simpel ohne großen Aufwand abzufragen und andererseits werden auch nur die Funktionen generiert, die in der Skill-DSL visualisiert wurden.

Darüber hinaus wird für jeden Server- beziehungsweise Prozess-Knoten eine korrespondierende Session-Klasse generiert. Dazu bietet die Session-Klasse entsprechende Getter- und Setter-Methoden für die jeweiligen Variablen im SessionCache, zu welchen der Server- oder Prozess-Knoten eine Daten-Kante besitzt. Entsprechend wird für jede Input-Kante eine Getter-Methode, für jede Output-Kante eine Setter-Methode und für jede Update-Kante sowohl Getter- als auch Setter-Methode generiert.

Für die Variablen verwalten die Sessions die Daten in einer globalen Map. Bei Variablen mit primitiven Datentypen werden dabei die Werte verwaltet, für komplexe DIME-Daten, werden die DYWA-IDs der Objekte in der Map verwaltet und mit der Setter-Methode gesetzt, beziehungsweise können sie mit der Getter-Methode abgerufen werden. Der eigentliche Zugriff auf das Objekt aus der Datenbank, wird für DIME-Daten, dann in dem Server-/Prozess-Knoten verwaltet.

Die Informationen, welche in der Session gespeichert werden, werden bei der Kommunikation mit Amazon im JSON Format hin und her geschickt. Die Implementierung geht über das, was der Skill-Entwickler sieht, noch einen Schritt hinaus und schreibt automatisch beispielsweise auch die erlaubten nächsten Intents und die State-Variable mit in die Session. Diese beiden Angaben werden für die Pfadprüfung der Handler benötigt.

4.2.8 Response-Klassen

Als weiterer Teil der Paketstruktur innerhalb des `src-gen`-Ordners sei kurz auf das `response`-Paket eingegangen. In dem `response`-Paket befinden sich die Klassen, welche der Skill-Entwickler benötigt, um eine individuelle Antwort über Amazon und damit Alexa an den Skill-Benutzer weiterzugeben. Über diese Klassen wird einerseits eine Sprachausgabe realisiert, als auch gegebenenfalls SSML implementiert. SSML, die *Speech Synthesis Markup Language*, bietet die Möglichkeit, neben der Standard-Alexa-Stimme auch weitere andere Stimmen zu nutzen oder beispielsweise die Betonung von Sätzen individuell zu gestalten.

Eine weitere wichtige Funktionalität bietet die Möglichkeit, auf der Skill-DSL nach einem Server-Knoten mehrere mögliche Responses zu erstellen. Pro möglicher Response nach einem Server-Knoten auf dem Graphmodell wird eine neue `Response`-Klasse für diese generiert und zusätzlich eine `ResponseHandler`-Klasse, die verwaltet, welche `Response`-Klassen zu welchem Server-Objekt gehören. Diese Eigenschaft, auch mehrere Responses nach einem Server-Objekt auf der Skill-DSL zu haben, bietet dem Skill-Entwickler die Möglichkeit, je nach Antwort eines Alexa-Nutzers auf einen Intent verschiedene unterschiedliche Pfade im Graphmodell und damit auch im Skill zu gehen. Je nachdem für welche Response sich der generierte Skill entscheidet, wird auch ein anderer erlaubter Pfad in den Handler-Klassen geprüft.

4.2.9 Prozess-Klassen

Erzeugt der Entwickler einen DIME-Prozess im Graphmodell, so wird für diesen eine Prozessklasse erzeugt. Eine Prozessklasse wird nur innerhalb des zum entsprechenden DIME-Prozess gehörenden Handler erzeugt und verwendet. Sie legt die nötigen Eingabedaten für den DIME-Prozess bereit und stößt diesen an. Daraufhin wird dessen Ergebnis ausgewertet, indem der Pfad ausgelesen und die zurückgegebenen Daten in der Session gespeichert werden.

Jeder Zugriff des Skills auf DIME-Klassen und dessen Datenbank erfolgt in den Prozessklassen. Hierfür werden mittels *Contexts and Dependency Injection* die benötigten DIME-Klassen injected. Dadurch kann der Zugriff auf die DIME-Datenbank erfolgen. Wird also innerhalb des Skills ein Datenobjekt verändert, so verändert sich dieses auch in der Datenbank. Um solch ein Datenobjekt für den Skill zugänglich zu machen, verwenden auch die Prozessklassen die Session.

4.3 Skill-Manifest

Beim Anlegen eines Skills wird für das Projekt automatisch eine Skill-Manifest-Datei angelegt (siehe Listing 4.2). Hier sollen die für das Manifest gewünschten Parameter gemäß Abschnitt 3.4 angepasst werden. Standardmäßig sind in dieser Datei Schreib- und Leserechte (siehe `permissions`) erteilt und für den `endpoint` das Zertifikat *Trusted* generiert. Aus der Skill-Manifest-Datei wird die `skill.json`-Datei (siehe Listing 4.3) generiert, welche für das Amazon Deployment von Bedeutung sein wird.

Einzeleinheiten zu weiteren Funktionalitäten in der Manifest-Datei sind unter [11] nachzulesen. Diese können in der generierten `skill.json` ergänzt werden. Hierbei ist zu beachten, dass bei Änderungen in der Skill-Manifest-Datei jene durch die Generierung überschrieben werden.

```
manifestversion : 1.0;
storeInformation {
  regions : global
  testingInstructions : "Testing Instructions Placeholder"
  language de-DE {
    summary : "Summay Placeholder"
    examples : ["Example 1", "Example 2", "Example 3"]
    description : ""
    keywords : ["Keyword Placeholder"]
    smallIconUri : ""
    largeIconUri : ""
    name : namePlaceholder
    privacyPolicyUrl : ""
    termsOfUseUrl : ""
  }
  advertisement : false
  childDirected : false
  personalInformationUsage : false
  purchases : false
  exportCompliant : false
}
technicalInformation {
  api "API Placeholder" {
    endpoint : {
      uri : "URI Placeholder"
      sslCertificateType : "Trusted"
    }
  }
}
permissions {
  alexa::alerts:reminders:skill:readwrite
}
```

Listing 4.2: Skill-Manifest-Vorlage

```
{ "manifest":{
  "manifestVersion": "1.0",
  "publishingInformation": {
    "locales": {
      "de-DE": {
        "summary": "Summary Placeholder",
        "name": "namePlaceholder",
        "description": "",
        "examplePhrases": [ "Example 1", "Example 2", "Example 3" ],
        "keywords":[ "Keyword Placeholder" ],
        "smallIconUri": "",
        "largeIconUri": "" } } },
  "permissions": [
    { "name": "alexa::alerts:reminders:skill:readwrite" }
  ],
  "privacyAndCompliance": {
    "containsAds": false,
    "isChildDirected": false,
    "usesPersonalInfo": false,
    "isExportCompliant": false,
    "allowsPurchases": false },
  "apis": {
    "custom": {
      "endpoint": {
        "uri": "URI Placeholder",
        "sslCertificateType": "Trusted" } } } } }
```

Listing 4.3: Generierte `skill.json`-Datei

4.4 WAR-Generierung

Hat sich der Skill-Entwickler gegen eine Integration in eine DIME-App entschieden und bereits ein MAVEN-Projekt generiert, so müsste der Skill-Entwickler eigentlich aus der `pom.xml` des MAVEN-Projekts die Generierung der WAR-Datei für den Web-Server anstoßen. Dieser Umstand wird mit einer komfortablen Lösung abgenommen. Ziel ist hierbei, dem Skill-Entwickler den Schritt in das generierte MAVEN-Projekt hinein zu ersparen und die Generierung aus dem Graphmodell anzustoßen.

Starten der WAR-Generierung

Neben dem Generierungs-Icon existiert ein Icon, um die WAR-Generierung zu beginnen (siehe Abbildung 4.6). Der Skill-Entwickler braucht sich dafür nicht in die Struktur des MAVEN-Projekts zu begeben, sondern kann die Generierung der benötigten WAR-Datei direkt aus dem Graphmodell heraus über die ECLIPSE-Toolbar anstoßen.

Wichtig ist hierbei zu beachten, dass das Graphmodell, zu dessen Projekt die WAR-Datei generiert werden soll, im Editor geöffnet ist. Sollte kein Graphmodell, also beispielsweise eine andere Datei, geöffnet sein beim Betätigen der WAR-Generierung, so wird keine Generierung angestoßen. Damit soll die Benutzerfreundlichkeit des Tools weiter gesteigert werden. Die WAR-Datei wird so benannt wie das Graphmodell des Skill-Projekts und sollte daher den Namen des Projekts individuell zum Zeitpunkt der Generierung automatisch setzen. Damit wird anhand des Namens der WAR ersichtlich, zu welchem Skill-Projekt sie gehört. Der Skill-Entwickler soll sich auch nach Generierung der WAR-Datei nicht in das MAVEN-Projekt begeben müssen. Daher wird die WAR-Datei nicht, wie standardmäßig üblich, irgendwo in einen Zielordner in das MAVEN-Projekt generiert, sondern in den `src-gen`-Ordner des Skill-Projekts. Somit kann der Skill-Entwickler sowohl die MAVEN-Projekt-Generierung, die Implementierungsklassen, als auch die WAR-Generierung direkt aus dem Graphmodell heraus anstoßen und bekommt die generierte WAR-Datei ebenfalls in die Struktur des Graphmodell-Projekts gelegt.

Im Falle des erfolgreichen Generierens der WAR-Datei bekommt der Skill-Entwickler einen entsprechenden Info-Dialog eingeblendet (siehe Abbildung 4.7). In der generierten WAR-Datei befinden sich alle benötigten Dateien und Informationen für den Web-Server, so dass diese direkt auf einen Web-Server, wie beispielsweise WILDFLY, hoch geladen werden kann. Dieser dient in der Kommunikation mit Amazon als sogenannter Endpoint und die Skill-Anfragen werden automatisch durch den in der WAR-Datei befindlichen kompilierten JAVA-Code bearbeitet.

Abbildung 4.6: Icon für WAR-Generierung in der Mitte

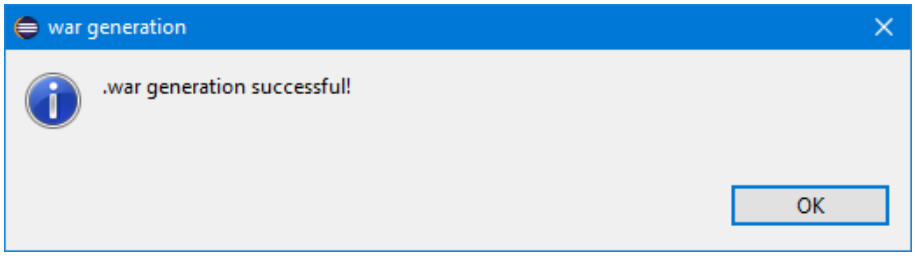


Abbildung 4.7: Erfolgsmeldung, WAR-Datei wurde generiert

Kapitel 5

Deployment

Damit die entwickelten Skills auch mithilfe von Alexa angesprochen werden können, wird ein Server benötigt, auf dem der Skill abgelegt wird. Amazon stellt mit seinen *Amazon Web Services* (AWS) dafür eigene Server zur Verfügung. Es besteht jedoch auch die Möglichkeit einen eigenen Web-Server zu verwenden. Damit hat der Entwickler zum einen mehr Kontrolle über die Infrastruktur und zum anderen ist es so möglich, eine hybride Java-Applikation zu verwenden. Diese hybride Anwendung kann somit Amazon-konform mittels des ASK SDK das Skill-Handling betreiben und zeitgleich auf Basis der Dime-Strukturen eine Web-Anwendung auf denselben Daten anbieten.

5.1 WildFly

Für eine Web-Anwendung wird ein sogenannter *Web Application Server* benötigt, in dem diese Applikationen verwaltet werden. Dime verwendet dafür standardmäßig den WildFly Application Server, der bei dem Erstellen einer Dime-Anwendung automatisch integriert ist. Damit der Technologie-Stack überschaubar bleibt, wird auch ein Wildfly verwendet, wenn letztendlich kein Dime-Produkt mit dem ASDT erstellt wird. Um die entwickelten Skills verwenden zu können, wird die generierte WAR-Datei eines Skills auf dem Server abgelegt. Wird nun ein Skill mithilfe von Alexa angesprochen, so wird auf dem Server durch den WildFly auf die zugehörige WAR-Datei umgeleitet und ausgeführt. Im Falle eines Dime-Produktes wird der unter `/target` generierte Ordner auf den Server geladen und über Docker gemanagt.

Über den an Amazon übergebenen Endpoint wird das *SkillServlet* des Skills, das als Einstiegspunkt für Alexa dient, angesprochen. Bei der reinen Skill-Anwendung genügt dabei ein Aufruf des Root-Pfades der Anwendung. Bei der DIME-Anwendung wird stattdessen auf das `/app/skill` Verzeichnis geleitet, um den SkillServlet anzusprechen.

Wie bereits unter Abschnitt 2.7 erwähnt, erfordert Amazon eine aktive TLS-Verschlüsselung und erlaubt ausschließlich eine Kommunikation über HTTPS. Für die Einrichtung der Verschlüsselung bei der Verwendung eines WILDFLYS ohne zusätzliches DOCKER-Setup reicht die übliche Konfiguration eines APACHES, NGINX oder ähnlichem. Bei der Verwendung von DOCKER gestaltet sich die Arbeit etwas aufwendiger und wird im nächsten Abschnitt mit aufgegriffen.

5.2 DOCKER

Hat sich der Entwickler für eine DIME-Anwendung entschieden, wird diese durch eine DOCKER-Instanz verwaltet. Dafür wird sowohl die DOCKER *Engine* als auch DOCKER *Compose* benötigt. Für die initiale Einrichtung muss zunächst eine TLS-Verschlüsselung eingerichtet werden. Im Anschluss erfolgen Anpassungen an dem DIME-Projekt, um einen störungsfreien Betrieb zu gewährleisten.

5.2.1 TLS mit DOCKER

Das Aufsetzen einer TLS-Verschlüsselung über DOCKER gestaltet sich etwas schwierig, da zunächst ein Dummy-Zertifikat benötigt wird, bevor ein dauerhaft gültiges erstellt werden kann. Es gibt jedoch fertige Skripte, die dem Entwickler in dieser Situation mit wenigen Schritten unterstützen. Im Anschluss kann mit der DOCKER-Instanz über HTTPS kommuniziert werden.

Damit die DIME-Anwendung auf die korrekten Zertifikate und Konfigurationen zugreift, wird bei der Generierung die von DIME bestehende `docker-compose.production.yml` überschrieben. Für den NGINX-Server, der die eingehenden Anfragen an das Backend der Anwendung steuert, werden die aktuell gültigen Zertifikate hinterlegt. Diese wurden mit Hilfe des Zertifikat-Verwaltungstools CERTBOT erstellt. Darüber hinaus wird die NGINX-Konfiguration für die Verwaltung von TLS-Anfragen hinterlegt (siehe Listing 5.1).

```
1 services:
2   nginx:
3     volumes:
4       - ./data/nginx/nginx-ssl.conf:/etc/nginx/conf.d/nginx-ssl.conf
5       - ./data/certbot/conf:/etc/letsencrypt
6       - ./data/certbot/www:/var/www/certbot
```

Listing 5.1: `docker-compose-production.yml` (Volumes)

```
services:
  nginx:
    ports:
      - "8080:80"
      - "443:443"
  angularnginx:
    ports:
      - "9090:80"
```

Listing 5.2: `docker-compose-production.yml` (Ports)

```
services:
  wildfly:
    environment:
      - "CLIENT_ORIGIN=http://ls5vs009.cs.tu-dortmund.de:9090"
```

Listing 5.3: `docker-compose-production.yml` (Origin)

Wenn es mehrere Services gibt, die auf dieselben Ports hören, kann dies zu Konflikten führen und es fehlt die Abgrenzung welcher Service aktiviert werden soll. Aus diesem Grund wurden die Ports, auf die von außen reagiert werden soll, umkonfiguriert (siehe Listing 5.2). Der Port vor dem Doppelpunkt leitet dabei jeweils auf den internen Port hinter dem Doppelpunkt um.

Zusätzlich muss der *Origin*-Wert des WildFlys auf die Domain gesetzt werden, von der der Server aus kommuniziert. Andernfalls kann es zu einem CORS-Fehler führen, da in der HTTPS-Response die Quelle nicht mit der URL aus der Request übereinstimmt. Diese Anpassung muss der Entwickler jedoch auch nicht selber vornehmen, sondern der Wert wird aus dem Skill-Manifest (Vergleich Abschnitt 3.3) entnommen und eingetragen (siehe Listing 5.3).

5.2.2 NGINX-Konfigurationen

Wie bereits erwähnt, wird für die TLS-Konfiguration eine zusätzliche Konfigurationsdatei angelegt. Diese bestimmt das Verhalten bei HTTPS-Requests und hat den Aufbau wie in Listing 5.4 beschrieben. Auch diese Datei wird mit den Werten des Skill-Manifests automatisch belegt. In der im selben Verzeichnis liegenden Datei `app.conf` sind die allgemeinen Konfigurationen hinterlegt (siehe Listing 5.5). Diese Datei wird in der Docker-Compose-Datei nicht explizit aufgeführt, da sie bereits durch die Standardkonfiguration berücksichtigt wird. Auch ihr Werte werden bereits automatisch belegt.

```
server {
  listen 443 ssl;
  server_name ls5vs009.cs.tu-dortmund.de;
  ssl on;
  ssl_certificate /etc/letsencrypt/live/ls5vs009.cs.tu-dortmund.de/fullchain.pem;
  ssl_certificate_key /etc/letsencrypt/live/ls5vs009.cs.tu-dortmund.de/privkey.pem;
  include /etc/letsencrypt/options-ssl-nginx.conf;
  ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;
  location / {
    proxy_pass          http://wildfly:8080;
    proxy_set_header    Host                    $http_host;
    proxy_set_header    X-Real-IP               $remote_addr;
    proxy_set_header    X-Forwarded-For         $proxy_add_x_forwarded_for;
    proxy_set_header    X-Forwarded-Proto       https;
  }
}
```

Listing 5.4: nginx-ssl.conf

```
http {
  server {
    if ($host = ls5vs009.cs.tu-dortmund.de) {
      return 301 https://$host$request_uri;
    }
    listen 80;
    listen [::]:80;
    server_name ls5vs009.cs.tu-dortmund.de;
    return 404;
  }
  server {
    listen 443 ssl;
    listen [::]:443 ssl;
    server_name ls5vs009.cs.tu-dortmund.de;
    ssl on;
    ssl_certificate /etc/letsencrypt/live/ls5vs009.cs.tu-dortmund.de/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/ls5vs009.cs.tu-dortmund.de/privkey.
        pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;
    location / {
      proxy_pass          http://127.0.0.1:8080;
      proxy_set_header    Host                    $http_host;
      proxy_set_header    X-Real-IP               $remote_addr;
      proxy_set_header    X-Forwarded-For         $proxy_add_x_forwarded_for;
      proxy_set_header    X-Forwarded-Proto       https;
    }
  }
}
```

Listing 5.5: app.conf

```
authc.successUrl    = http://ls5vs009.cs.tu-dortmund.de:8080/app/index.html
logout.redirectUrl = http://ls5vs009.cs.tu-dortmund.de:9090
```

Listing 5.6: shiro.ini

5.2.3 Sonstige Anpassungen

Darüber hinaus wird die Konfiguration der `shiro.ini` ebenfalls auf Basis des Skill-Manifests überschrieben. Dieser Vorgang erfolgt, nachdem die DIME-Anwendung und das Skill-Manifest generiert wurden. Hier müssen die Redirect-URLs überschrieben werden (siehe Listing 5.6).

5.3 Skill Management

Um den entwickelten Skill nutzen zu können, muss dieser über einen *Amazon-Service* verwaltet werden. Wie bereits in Abschnitt 2.8 beschrieben, ist hierfür ASK CLI vorgesehen. ASK CLI selbst verfügt über keine Methode, um einen Skill zu veröffentlichen. Dies kann jedoch über ASK CLIs Einbindung von SMAPI erfolgen.

5.3.1 Verwendung von ASK CLI

Um ASK CLI durch CINCO zu nutzen, wird JAVA-Code benötigt, der die ASK-CLI-Befehle ausführt und die Projektordner an den zu erzeugenden Skill anpasst. Dieser Code ist in Listing 5.7 dargestellt. Da ASK CLI eine Interaktion mit dem Nutzer verlangt, muss dies über die BASH beziehungsweise die *Windows-Eingabeaufforderung* (CMD) ausgeführt werden. Da der Befehl sich bei jedem Aufruf gleich verhält, ist es leicht vorherzusagen, was getan werden muss.

Je nach Betriebssystem unterscheidet sich das aufzurufende Programm. Durch die *System Properties* kann zwischen den Betriebssystemen unterschieden werden. Um BASH oder CMD zu starten, wird ein `ProcessBuilder` verwendet. Für diesen `ProcessBuilder` wird ein Thread gestartet, der ausgibt, was ASK CLI zurückgibt. Der Hauptprozess übergibt währenddessen die Eingabe. Diese beinhaltet den ASK-CLI-Befehl sowie zusätzliche Eingaben, die ASK CLI fordert.

5.3.2 Automatisches Deployment und Update

Genügen dem Benutzer die Deployment- und Update-Funktionen, so kann er diese über den *SkillDeployment*-Button nutzen. Der Button befindet sich neben dem WAR-Button und hat als Symbol eine kleine Rakete, wie in Abbildung 4.6 zu sehen ist. Beim ersten Betätigen des Buttons werden die nötigen Projektverzeichnisse, wie in Unterabschnitt 2.8.2 beschrieben, automatisch anhand eines Templates erstellt. Im zweiten Schritt wird deployt, falls alle Verzeichnisse korrekt vervollständigt worden sind (siehe Listing 5.7). Bei einem erneuten Betätigen des Buttons können Änderungen, die im *Skill Manifest* vorgenommen wurden, durch ein Update (siehe Unterabschnitt 2.8.3) bei Amazon aktualisiert werden. Die Skill-ID wird dabei aus der Datei `/.ask/config` im `CLI` Projektordner ausgelesen.

```
public static void deploySkill(String pathToSkill, String name) throws IOException
{
  // Betriebssystem abfragen
  String os = System.getProperty("os.name");

  // Programm starten
  ProcessBuilder pb = new ProcessBuilder("/bin/bash");
  if (os.contains("windows")) {
     pb.command("CMD");
  }
  Process proc = pb.start();

  // Thread, der vom Program. liest
  final Scanner in = new Scanner(proc.getInputStream());
  new Thread() {
    public void run() {
      while (in.hasNextLine()) {
        System.out.println(in.nextLine());
      }
    }
  }.start();

  // Kommandos an das Programm schreiben
  PrintWriter out = new PrintWriter(proc.getOutputStream());
  out.println("cd " + pathToSkill + "/" + name);
  out.flush();
  out.println("ask deploy");
  out.flush();
  out.close();
}
```

Listing 5.7: Implementierung einer ASK-CLI-`Deploy`-Funktion

Kapitel 6

Zusammenfassung

Die Projektgruppe beschäftigte sich mit dem Entwurf einer domänenspezifischen graphischen Modellierungssprache zur Entwicklung von Alexa-Skills. Dabei sollte DIME mit in die Modellierungssprache integriert werden. Zu Beginn wurden die wichtigsten Grundlagen, wie beispielsweise *modellgetriebene Softwareentwicklung* und CINCO, erklärt. Im Hauptteil der Ausarbeitung wurden die drei großen Themengebiete *Modellierung*, *Codegenerierung* und *Deployment* beschrieben. Für die Modellierung wurde die graphische Skill-DSL erläutert, mit welcher der grundlegende Ablauf eines Skills festgelegt wird. Hierfür wurde ein Quiz-Beispiel verwendet, um diese Sprache zu veranschaulichen. Anschließend wurde eine textuelle Intents-DSL eingeführt und erklärt, die dafür notwendig ist, dass Intents und deren Slots für die Skill-DSL zur Verfügung gestellt werden und dass eine von Amazon geforderte JSON-Datei für das Interaction Model generiert wird. Außerdem wurde eine textuelle Manifest-DSL entwickelt, mit der eine Skill-Manifest erzeugt wird, die alle nötigen Informationen eines Alexa-Skills enthält, welche für die Verwendung des Skills notwendig sind. Nachdem die Modellierung abgeschlossen wurde, wurde die Codegenerierung betrachtet. Hierbei wurde zunächst die Modelltransformation sowie JSON-Generierung erläutert. Anschließend wurde die Generierung der Projektstruktur sowie die Integration eines Alexa-Skills in ein DIME-Projekt beschrieben. Danach wurden dann die generierten Pakete und Klassen erläutert. Abschließend wurde bei der Codegenerierung die WAR-Generierung beschrieben.

6.1 Fazit

Eines der Ziele der Projektgruppe war das Entwickeln domänenspezifischer Modellierungssprachen, um Alexa-Skills zu erstellen. Daraus resultiert ist das Alexa Skill Development Tool, welches auf dem Metatool Cinco basiert. Zudem sollte das Tool so erweitert werden, dass eine parallel entwickelte Dime-App mittels Alexa angesteuert werden kann. Es wurden die graphischen sowie textuellen DSLs für den Skill-Ablauf und außerdem Alexa-Intents und ein Serversystem zum Deployen von entworfenen Skills erzeugt. Des weiteren wurden die graphische Skill-DSL um Dime-Elemente erweitert und die Codegeneratoren der Skill-DSL zum Aufrufen von Dime-Prozessen und Synchronisieren von Dime-bezogenen Daten angepasst. Zusätzlich wurden die Codegeneratoren um eine State-Variable erweitert und die textuelle DSL für die Manifest-Datei entwickelt. All das wurde anhand des mit Dime entwickelten Quiz-Beispiels mithilfe von Amazon-Geräten erfolgreich getestet. Die Ziele dieser Projektgruppe wurden somit erfüllt.

Kapitel 7

Ausblick

Abschließend erfolgt ein Ausblick auf mögliche Erweiterungen für das ASDT, sodass die Entwicklung eines Alexa-Skills für den Skill-Entwickler vereinfacht und die Möglichkeiten bei der Entwicklung vergrößert werden sollen. In diesem Kontext wurde das Konzept der *Verschachtelung* entwickelt, aber nicht vollständig in das ASDT integriert. Dieses Konzept sowie weitere Erweiterungsvorschläge werden im Folgenden im Detail beschrieben.

7.1 Verschachtelung

Bei der Entwicklung eines Skills kann es vorkommen, dass der Skill-Entwickler bereits modellierte eigenständige Skills oder Teilstücke eines Skills, sogenannte Sub-Skills, wiederverwenden möchte. Mit dem bisherigen ASDT müsste der Skill-Entwickler diese Sub-Skills mehrfach vollständig modellieren, sodass keine Wiederverwendbarkeit gewährleistet wäre. Um dies zu ermöglichen und somit das Tool modularer zu gestalten, wurde die Verschachtelung von Skills konzeptioniert. Bisher wurden bereits die Modellierung und Validierung umgesetzt, sodass die Verschachtelung auf diesen Ebenen ermöglicht wird. Lediglich die Generierung wird an dieser Stelle nur konzeptionell vorgestellt. Für die Vorstellung dieses Konzeptes wird weiterhin als Beispiel der Quiz-Skill verwendet, sodass die bisherige Modellierung in eine Modellierung mit Verschachtelung transformiert wurde.

7.1.1 Projekt-Wizard

Zum Anlegen eines neuen Projekts wird der Projekt-Wizard aus Abschnitt 3.1 dahingehend erweitert, dass zusätzlich zu einem ASW-Modell nun ein zweites Modell angelegt wird. Diese Modelle sind die ASW- sowie die ASkill-Datei. Wie auch beim bisherigen Wizard,

kann dieser mit einem Rechtsklick im Projekt-Browser unter dem Eintrag „New Alexa Skill Workflow Project“ aufgerufen und anschließend ein Projekt angelegt werden (siehe Abbildung 7.1).

7.1.2 Modellierung

Um einen Skill zu modellieren, stehen zwei Modelle zur Verfügung. Zum einen das Root-Modell, auch ASkill genannt, sowie das ASW-Modell, im Folgenden ASWorkflow genannt. Beide Modelle werden im Weiteren anhand des bisherigen Quiz-Beispiels erläutert und Unterschiede sowie Vorteile zum bisherigen Modell verdeutlicht.

ASkill

Das ASkill-Modell ist der Einstiegspunkt eines Skills. In dem Modell in Abbildung 7.2 stehen genau ein Start-Knoten, ein Flow-Knoten, welcher den Skill selbst als ASW-Modell repräsentiert (siehe (1.)), und höchstens ein Flow-Intent (siehe (2.)), welcher einen Intent in einem ASW-Modell referenziert, zur Verfügung. Ein SessionCache wie in Abschnitt 3.2 steht nicht mehr zur Verfügung, da beim Start des Skills kein Datenfluss ermöglicht wird. Um einen Skill zu starten, kann auch hier, wie in der bisherigen Modellierung, direkt ein Intent oder LaunchRequest angesprochen werden. Der Nutzer hat mit einem Rechtsklick in das Canvas eines ASkills die Möglichkeit die Funktion „Add Initial Layout“ zu verwenden, die ein initiales Layout erstellt. Dieses Layout besteht aus einem Start-Knoten mit einem Invocation Name, Skill-ID und Projekt-Namen. Der Nutzer hat zudem die Möglichkeit durch einen Doppelklick auf einen Flow das dazugehörige Modell zu öffnen. Mit einem Doppelklick auf einen Intent wird das Modell geöffnet, in dem der Intent vorhanden ist.

ASWorkflow

Mit dem ASWorkflow werden Sub-Skills modelliert, welche beliebig in anderen ASWorkflows wiederverwendet werden können, um sowohl Verschachtelung als auch Rekursion zu ermöglichen. Dabei stehen dem Skill-Entwickler dieselben Modellelemente zur Verfügung, wie sie bereits in Abschnitt 3.2 beschrieben wurden. Lediglich der Start- und End-Knoten wurden dahingehend angepasst, dass diese nun Daten beinhalten können. Dazu werden Kanten vom Start-Knoten zu Datenelementen aus dem SessionCache gezogen, um so Output-Ports anzulegen. Analog funktioniert dies für das Anlegen von Input-Ports im End-Knoten. Außerdem wurde ein *Flow-Knoten* ergänzt, welcher einen ASWorkflow repräsentiert und die eigentliche Verschachtelung realisiert.

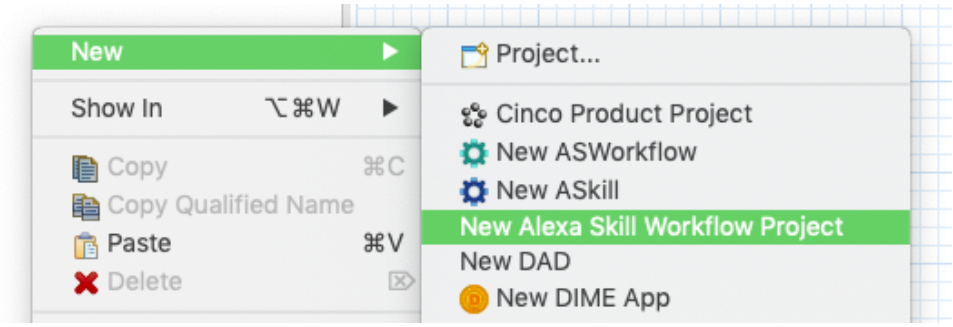


Abbildung 7.1: Anlegen eines Projekts mithilfe des neuen Wizards

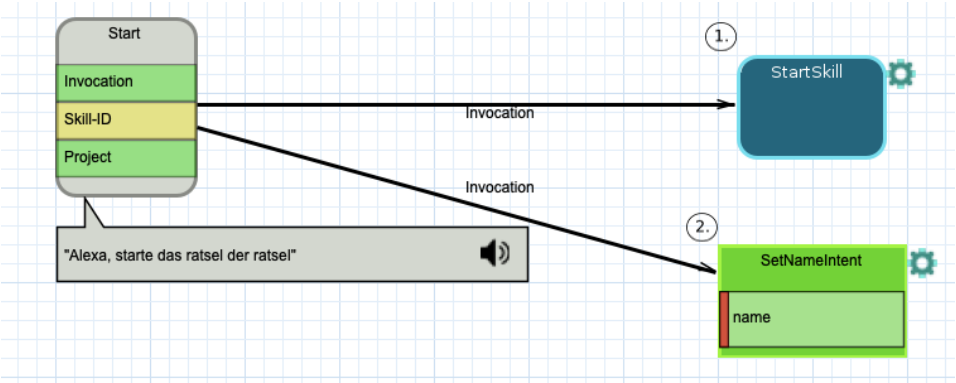


Abbildung 7.2: Beispiel eines ASkill-Modells

Nach einem Start-Knoten muss immer zuerst ein Server- oder ein Prozess-Knoten folgen (siehe Abbildung 7.3), aber ansonsten sind dieselben Abläufe möglich, wie sie bereits in Abschnitt 3.2 beschrieben wurden. Dadurch, dass ein ASWorkflow mit einem Server beziehungsweise einem Prozess beginnt, kann das Referenzieren eines ASWorkflows in Form eines Flow-Knotens nur nach einem Intent erfolgen.

Der Nutzer hat mit einem Rechtsklick in das Canvas eines ASWorkflows die Möglichkeit die Funktion „Add Initial Layout“ zu verwenden, die ein initiales Layout erstellt. Dieses Layout besteht aus einem Start-Knoten, einem Server-Knoten und einem Response-Knoten. Eine weitere Funktion ist „Synchronize with Sub-Flows“. Diese kann auch mit einem Rechtsklick in das Canvas eines Flows aufgerufen werden. Sollten sich in einem Sub-Flow innerhalb eines Flows die Input-Ports, Output-Ports oder die End-Knoten verändern, so werden mithilfe von „Synchronize with Sub-Flows“ diese Änderungen im Flow übernommen.

Zudem hat Nutzer die Möglichkeit durch einen Doppelklick auf einen Flow das dazugehörige Modell zu öffnen. Mit einem Doppelklick auf einen Intent wird das Modell geöffnet, in dem der Intent vorhanden ist.

Das Erstellen eines Flow-Knotens ist möglich, in dem aus dem Projekt Browser eine ASW-Datei auf den Canvas gezogen wird. Dabei werden, falls im referenzierten Modell vorhanden, die Output-Ports des Start-Knotens übernommen sowie für jeden End-Knoten in dem referenzierten Modell ein Branch angelegt, welcher gegebenenfalls Input-Ports beinhalten kann. Außerdem werden diese Modellelemente durch ein Zahnrad-Icon markiert (siehe (1.) und (2.) in Abbildung 7.3).

In dem Beispiel wird bei (1.) in Abbildung 7.3 das *StartQuiz* referenziert, welches in Abbildung 7.4 zu sehen ist. Das StartQuiz selbst besitzt im Start- sowie End-Knoten keine Ports und hat auch nur einen End-Knoten namens *EndQuiz*. Daher wird bei der Referenzierung dieses Modells in Abbildung 7.3 lediglich der Flow-Knoten mit dem dazugehörigen Branch *EndQuiz* ohne Ports angelegt.

Bei dem *StartSkill*-Modell handelt es sich um das erste referenzierte Modell, sodass dieser einerseits das Starten des Skills beschreibt und, anderseits keine Daten also keine Ports in Start- und End-Knoten besitzen darf. Da das ASkill-Modell keinen Datenfluss besitzt, können auch keine Daten an den ersten referenzierten ASWorkflow übermittelt werden.

Alle weiteren ASWorkflows können jedoch Datenfluss besitzen, wie es (1.) in Abbildung 7.4 zu sehen ist. Dort wird das *Create-Highscore*-Modell referenziert, welches einen Text als Input benötigt. Daher können ähnlich zu den Prozess-Knoten nun Kanten von Datenelementen aus dem SessionCache auf den Port des Flow-Knotens gezogen werden. Analog dazu funktioniert der Datenfluss bei den Branches, welche einen Output besitzen können, sodass diese im SessionCache abgespeichert werden können. Hierfür muss der Skill-Entwickler eine ausgehende Kante vom Port zu einem Datenelement ziehen. Sollte sich bei Datenelementen im SessionCache der Typ oder der Name ändern, so wird diese Änderung an allen Ports übernommen, die mit den entsprechenden Datenelementen verbunden sind.

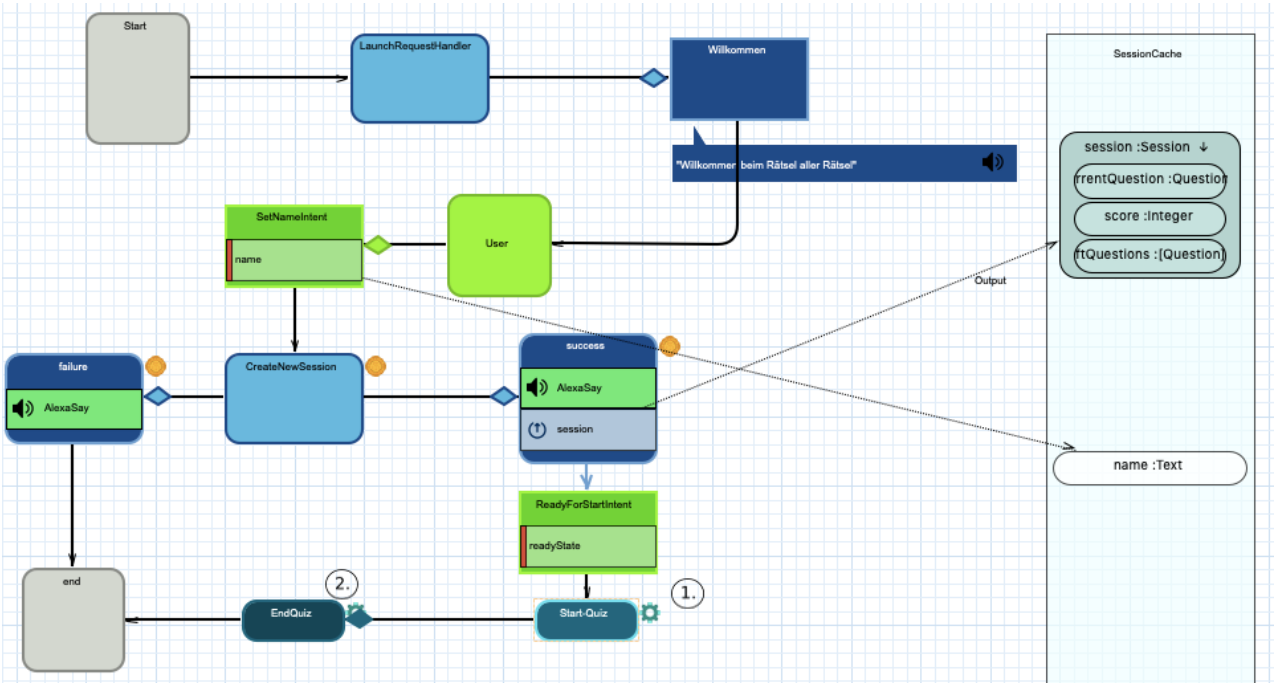


Abbildung 7.3: Beispiel eines ASWorkflow *StartSkill*

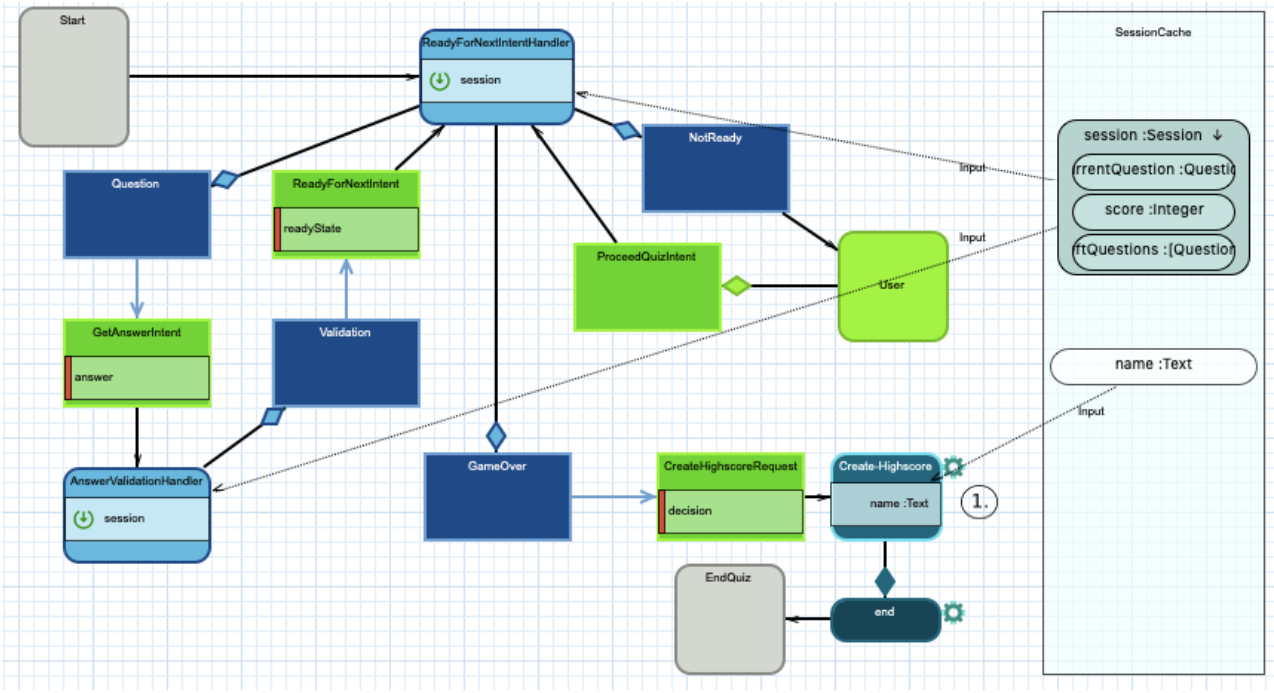


Abbildung 7.4: Beispiel eines ASWorkflow *StartQuiz*

7.1.3 Validierung

Damit der Skill-Entwickler bei der Modellierung unterstützt wird, erfolgt eine Validierung des modellierten Skills, sodass valide Modelle gewährleistet werden. Die bisher vorgestellten Validierungsfälle (siehe Unterabschnitt 3.2.2) werden nun um weitere Fälle ergänzt.

1. Ein ASWorkflow kann mehrere End-Knoten haben, wie es bereits in Unterabschnitt 7.1.2 beschrieben wurde. Damit deutlich wird an welchem End-Knoten der Sub-Skill beendet wird, müssen die Namen der End-Knoten eindeutig gewählt sein. Durch die Referenzierung eines Sub-Skills in einem anderen ASWorkflow werden die End-Knoten des Sub-Skills als Branches angelegt. Demnach müssen auch die Branches unterhalb des Sub-Skills eindeutig benannt sein. Eine uneindeutige Benennung der Branches kann nur dann auftreten, wenn die End-Knoten des Sub-Skills bereits uneindeutig sind. In beiden Fällen wird eine Fehlermeldung ausgegeben, wenn die Namen nicht eindeutig sind (siehe (1.) in Abbildung 7.7 und (2.) in Abbildung 7.7).

2. Beim Starten eines Skills in dem ASkill-Modell können keine Daten an den ersten referenzierten ASWorkflow mitgegeben werden. Demnach darf dieser Sub-Skill keine Daten im Start-Knoten in Form von Input-Ports besitzen und muss validiert werden. Bei Missachten wird eine entsprechende Warnung ausgegeben (siehe (1.) in Abbildung 7.5).

3. Analog zu der Typübereinstimmung von Prozessports und den referenzierten Daten, müssen auch die Ports der Start- und End-Knoten in den ASWorkflow-Modellen überprüft werden. Dazu wird einerseits betrachtet, ob die Typen an sich übereinstimmen und ob beides Listen beziehungsweise Maps sind (siehe (1.) und (3.) in Abbildung 7.6). Bei der Typüberprüfung muss der Typ exakt übereinstimmen, da bislang nicht auf Vererbungen im Sinne von `instanceof` geachtet wird.

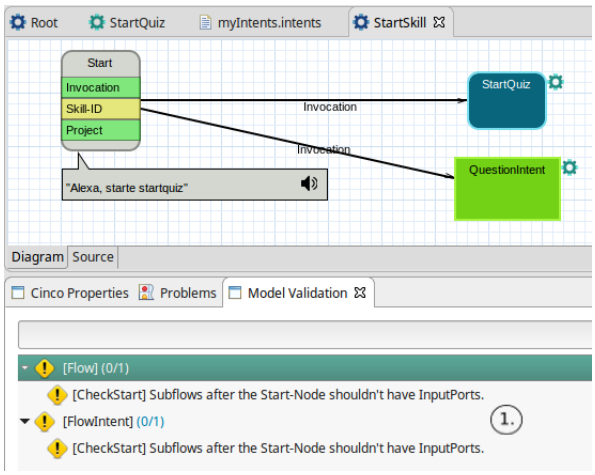


Abbildung 7.5: Beispiel eines invaliden ASkill-Modells mit Warnmeldungen

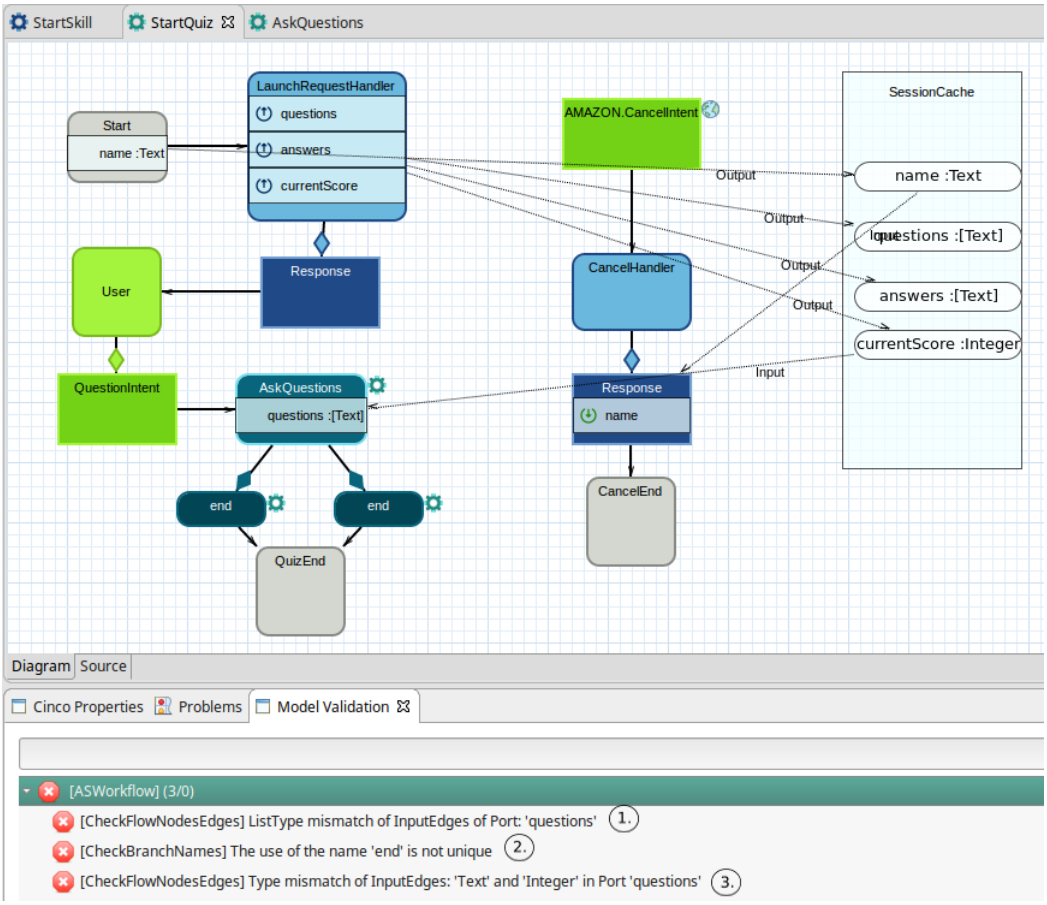


Abbildung 7.6: Beispiel eines invaliden ASWorkflow-Modells mit referenziertem ASWorkflow-Modell

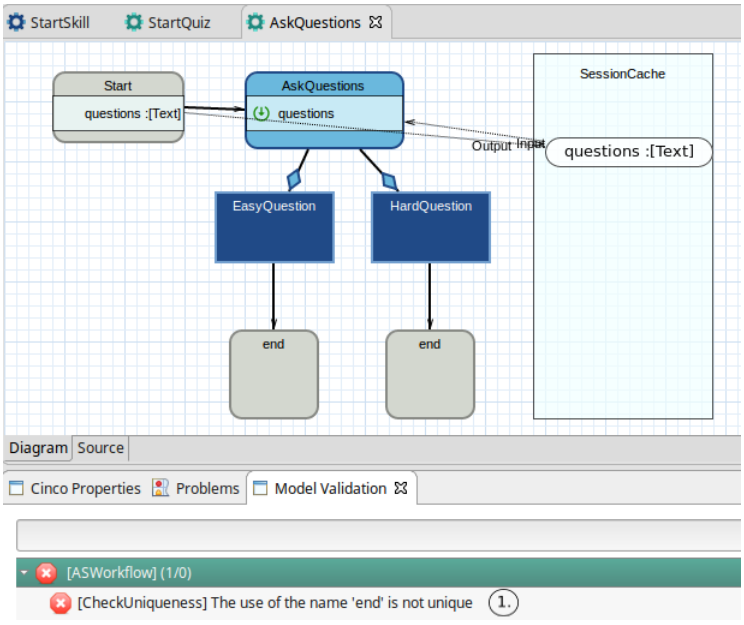


Abbildung 7.7: Beispiel eines invaliden ASWorkflow-Modells

Beispiel der Validierung

Anhand von beispielhaften invaliden Modellen sollen die erwähnten Validierungsfälle veranschaulicht werden. Dazu wird das ASkill-Modell *StartSkill* und die ASWorkflow-Modelle *StartQuiz* und *AskQuestions* betrachtet.

In Abbildung 7.5 wird im ASkill-Modell das ASWorkflow-Modell *StartQuiz* sowie der Intent *QuestionIntent* im selbigen Modell referenziert. Das Modell selbst enthält im Start-Knoten Input-Ports, wie es in Abbildung 7.6 zu sehen ist. Da der erste referenzierte ASWorkflow keine Daten enthalten darf, wird eine Warnung für den Skill-Entwickler ausgegeben.

In dem Modell aus Abbildung 7.6 wird der ASWorkflow *AskQuestions* referenziert, weshalb die Ports des referenzierten Modells ausgelesen und erstellt werden. Das Modell *AskQuestions* benötigt eine Liste von Texten als Input, weshalb der Port mit Daten aus dem SessionCache verbunden wird. Allerdings wird als Datenobjekt *currentScore* verwendet, welches ein Integer ist. Dadurch liegt einerseits ein *Type Mismatch* vor, da anstatt eines Integers ein Text erwartet wird (siehe ③.). Andererseits wird eine Liste von Texten benötigt, weshalb auch ein *list mismatch* vorliegt (siehe ①.). Neben den Ports werden auch die End-Knoten des referenzierten Modells ausgelesen und in Form von Branches erstellt. Da die End-Knoten in *AskQuestions* nicht eindeutig sind (siehe ①. in Abbildung 7.7), sind auch die Namen der Branchen uneindeutig. Daher wird auch hier eine Fehlermeldung ausgegeben (siehe ②.).

7.1.4 Generierung

Die bestehenden Generatoren müssten für die Codegenerierung aus einem Verschachtelungsprojekt angepasst werden. Mit dem entwickeltem State-Variable-Verfahren kann auch bei einem verschachtelten Modell die aktuelle Position innerhalb eines Graphens bestimmt werden. Da aber jeder Sub-Flow einen von anderen Sub-Flows unabhängigen Bereich zur Datenverwaltung im SessionCache haben soll, muss das SV-Konzept so erweitert werden, dass eine gegebenenfalls vorhandene Rekursionstiefe erkannt wird, um nur auf Daten des Rekursiontiefen-spezifischen Datenbereiches zu arbeiten. Hierzu kann zusätzlich zur SV eine Layer-Variable (kurz LV) im SessionCache gesetzt werden, mit der die Rekursions- beziehungsweise Verschachtelungstiefe gemerkt wird. Wenn ein neuer Sub-Flow gestartet wird, müsste die LV dann um eine Sub-Flow-spezifische Komponente erweitert werden, und wenn ein Sub-Flow beendet wird, müsste diese Komponente wieder aus der LV entfernt werden, sodass ein vorheriger Sub-Flow mit seiner alten LV weiterarbeiten kann.

Außerdem sollen nach dem entwickeltem Konzept Daten über Input-Ports über- und über Output-Ports ausgegeben werden können. Bisher war es gedacht, Datenkopien zu übergeben, allerdings könnten außerdem Referenzen auf Daten in anderen Bereichen des Session-

Cache übergeben werden. Damit die Daten der Input-Ports verarbeitet werden können, müsste der persönliche Datenbereich im SessionCache angelegt und über diesen Bereich verbundene Zuweisungen von Input-Daten verarbeiten werden, bevor die Implementierung des ersten Handlers ausgeführt wird, die einerseits auf Input-Daten oder Daten des separaten Datenbereichs zugreifen kann. Die zu importierenden Input-Daten werden dann über die SV und LV identifiziert. Mit der LV würde der Datenbereich des vorherigen Sub-Flows bezogen und über die mit der SV bestimmte Position im vorherigen Sub-Flow könnten somit die Daten in dem Datenbereich des Vorgängers identifiziert werden.

Insgesamt würde der erste Handler eines Sub-Flows also die folgenden Schritte in größtenteils chronologischer Reihenfolge ablaufen:

1. Mit der SV prüfen, ob er ausgeführt werden soll
2. LV beziehen, erweitern und setzen
3. Eigenen Datenbereich im SessionCache über neue LV anlegen
4. Datenbereich des Vorgängers über alte LV beziehen
5. Input-Daten im Datenbereich des Vorgängers über SV identifizieren
6. Input-Zuweisungen in eigenem Datenbereich verarbeiten
7. Handler-Implementierung ausführen und Handler-Outputs verarbeiten
8. Neue SV setzen

Innerhalb eines ASWorkflow-Modelles müssten Handler ihre Inputs und Outputs über den mit der gesetzten LV identifizierbaren Datenbereich verarbeiten.

Wenn ein Sub-Flow endet, können drei Fälle im vom Sub-Flow verwendeten Flow auftreten:

1. Der nächste Handler kann innerhalb des Vorgängers liegen
2. Der Vorgänger ruft direkt einen weiteren Sub-Flow auf
3. Der Flow des Vorgängers endet ebenfalls

Für diese Fälle wurde das Konzept der Codegenerierung allerdings noch nicht weiterentwickelt. Trotzdem sollten umsetzungsunabhängig die folgenden Schritte durchgeführt werden:

1. Output-Daten verarbeiten
2. Separaten Datenbereich löschen
3. Neue SV setzen
4. LV zurücksetzen

7.2 Sonstiges

Neben dem Verschachtelungskonzept könnten für das Alexa Skill Development Tool noch andere Erweiterungen entwickelt werden, die leider nicht im Rahmen der PG umsetzbar waren. Die Servlet-Logik könnte zum Beispiel so erweitert werden, dass Handler-Klassen nur nach Bedarf erstellt werden. Hierzu müsste bei einem Intent-Aufruf die SV bezogen werden und nur der zu der gesetzten SV korrespondierende Handler per Konstruktoraufruf erstellt und aufgerufen werden. Außerdem könnte ein Verfahren entwickelt werden, sodass die Wiederverwendung von Handler-Logik bereits innerhalb eines Modelles visualisiert und von Generatoren umgesetzt wird. Bisher kann implementierte Handler-Logik über öffentliche statische Java-Methoden wiederverwendet werden. Ebenfalls könnte ein Handler-Chaining entwickelt werden, sodass zwischen Intent und Response beliebig viele Handler aufgerufen werden können. Auch könnten Handler-Templates, die vordefinierte Methoden aufrufen, einem Entwickler die Arbeit abnehmen, selbst Implementierungen zu schreiben. Wie an den gerade vorgestellten Konzepten ersichtlich ist, können noch mehrere Erweiterungen an dem entwickelten ASDT umgesetzt werden, die Performance verbessern, die Entwicklung von Skills komfortabler gestalten oder neue Möglichkeiten bei der Modellierung schaffen sollen. Diese sind allerdings nicht für das Modellieren von Alexa-Skills notwendig, sondern würden lediglich neue Features hinzufügen oder alte Features verbessern.

Abbildungsverzeichnis

Listingverzeichnis

Literaturverzeichnis

[1] Amazon: *Account Linking Schemas.* https://developer.amazon.com/de/docs/smapi/account-linking-schemas.html – Letzter Zugriff am 07.09.2019.

[2] Amazon: *Alexa Skills Kit SDK for Java.* https://developer.amazon.com/de/docs/alexa-skills-kit-sdk-for-java/overview.html – Letzter Zugriff am 07.09.2019.

[3] Amazon: *Alexa Skills Kit SDKs.* https://developer.amazon.com/de/docs/sdk/alexa-skills-kit-sdks.html – Letzter Zugriff am 07.09.2019.

[4] Amazon: *Amazon Developer Console.* https://developer.amazon.com/de/alexa – Letzter Zugriff am 07.09.2019.

[5] Amazon: *Build Skills with the Alexa Skills Kit.* https://developer.amazon.com/de/docs/ask-overviews/build-skills-with-the-alexa-skills-kit.html – Letzter Zugriff am 07.09.2019.

[6] Amazon: *Get Started with AVS.* https://developer.amazon.com/de/docs/alexa-voice-service/get-started-with-alexa-voice-service.html – Letzter Zugriff am 07.09.2019.

[7] Amazon: *Get Started with SMAPI.* https://developer.amazon.com/de/docs/smapi/smapi-overview.html – Letzter Zugriff am 07.09.2019.

[8] Amazon: *Host a Custom Skill as a Web Service.* https://developer.amazon.com/de/docs/custom-skills/host-a-custom-skill-as-a-web-service.html – Letzter Zugriff am 07.09.2019.

[9] Amazon: *Host a Custom Skill as an AWS Lambda Function.* https://developer.amazon.com/de/docs/custom-skills/host-a-custom-skill-as-an-aws-lambda-function.html – Letzter Zugriff am 07.09.2019.

[10] Amazon: *Request and Response JSON Reference.* https://developer.amazon.com/de/docs/custom-skills/request-and-response-json-reference.html – Letzter Zugriff am 07.09.2019.

[11] Amazon: *Skill Manifest Schema.* https://developer.amazon.com/de/docs/smapi/skill-manifest.html – Letzter Zugriff am 07.09.2019.

[12] Amazon: *Speech Synthesis Markup Language (SSML) Reference.* https://developer.amazon.com/de/docs/custom-skills/speech-synthesis-markup-language-ssml-reference.html – Letzter Zugriff am 07.09.2019.

[13] Amazon: *Steps to Build a Custom Skill.* https://developer.amazon.com/de/docs/custom-skills/steps-to-build-a-custom-skill.html – Letzter Zugriff am 07.09.2019.

[14] Apache Software Foundation: *Apache Maven – Software project management and comprehension tool.* https://maven.apache.org/ – Letzter Zugriff am 07.09.2019.

[15] Behrens, Clay et al.: *Xtext User Guide.* The Eclipse Foundation, 2010.

[16] Bosselmann, Frohme, Kopetzki, Lybecait, Naujokat, Neubauer, Wirkner, Zweihoff und Steffen: *DIME: A Programming-Less Modeling Environment for Web Applications.* In: *7th International Symposium, ISoLA 2016 Imperial, Corfu, Greece, October 10–14, 2016 Proceedings, Part II*, Seiten 809–830, 2016.

[17] Brambilla, Cabot und Wimmer: *Model-driven software engineering in practice.* Synthesis Lectures on Software Engineering, 1(1):1–182, 2012.

[18] Chamberlin und Boyce: *SEQUEL: A structured English query language.* In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, Seiten 249–264. ACM, 1974.

[19] Czarnecki und Helsen: *Classification of model transformation approaches.* In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Band 45, Seiten 1–17, 2003.

[20] Dudenverlag: *Duden | Modell | Rechtschreibung, Bedeutung, Definition, Synonyme, Herkunft.* https://www.duden.de/rechtschreibung/Modell – Letzter Zugriff am 07.09.2019.

[21] Eclipse Foundation, The: *Chapter 5. Programer Guide.* https://www.eclipse.org/amp/documentation/contents/Programer_Guide.html – Letzter Zugriff am 07.09.2019.

[22] Eclipse Foundation, The: *Eclipse Modeling Project.* `https://www.eclipse.org/modeling/emf/` – Letzter Zugriff am 07.09.2019.

[23] Eclipse Foundation, The: *Ecore – Eclipsepedia.* `https://wiki.eclipse.org/Ecore` – Letzter Zugriff am 07.09.2019.

[24] Eclipse Foundation, The: *Enabling Open Innovation & Collaboration.* `http://www.eclipse.org/` – Letzter Zugriff am 07.09.2019.

[25] Eclipse Foundation, The: *Rich Client Platform FAQ – Eclipsepedia.* `https://wiki.eclipse.org/Rich_Client_Platform/FAQ` – Letzter Zugriff am 07.09.2019.

[26] Eclipse Foundation, The: *What is Eclipse? – Eclipsepedia.* `https://wiki.eclipse.org/FAQ_What_is_Eclipse%3F` – Letzter Zugriff am 07.09.2019.

[27] Eclipse Foundation, The: *Xtend Documentation.* `https://www.eclipse.org/xtend/documentation/` – Letzter Zugriff am 07.09.2019.

[28] Eclipse Foundation, The: *Xtext Documentation.* `http://www.eclipse.org/Xtext/documentation/102_domainmodelwalkthrough.html` – Letzter Zugriff am 07.09.2019.

[29] Fuhge, Annika: *Graphische Modellierung von Cinco Produktspezifikationen.* 2018.

[30] Kellokoski, Pasi: *Round-trip Engineering.* 2000.

[31] Kurpjuweit, Stephan: *Stakeholder-orientierte Modellierung und Analyse der Unternehmensarchitektur unter besonderer Berücksichtigung der Geschäfts- und IT-Architektur.* Nummer 3634. Logos Verlag Berlin GmbH, 2009.

[32] Mazanec und Macek: *On General-purpose Textual Modeling Languages.* In: *Dateso*, Band 12, Seiten 1–12. Citeseer, 2012.

[33] Naujokat, Lybecait, Kopetzki und Steffen: *CINCO: A simplicity-driven approach to full generation of domain-specific graphical modeling tools.* International Journal on Software Tools for Technology Transfer, Juni 2018.

[34] Naujokat, Stefan: *Model-Driven Domain-Specific Generation of Generative Domain-Specific Modeling Tools.* Doktorarbeit, Technische Universität Dortmund, 2017.

[35] OAuth: *OAuth 2.0 – Protocol for athorization.* `https://oauth.net/2/` – Letzter Zugriff am 07.09.2019.

[36] Object Management Group: *About the Unified Modeling Language Specification Version 2.5.1.* `https://www.omg.org/spec/UML/About-UML/` – Letzter Zugriff am 07.09.2019.

[37] Oracle Corporation: *javax.json – Java (TM) EE 7 Specification APIs.* `https://docs.oracle.com/javaee/7/api/javax/json/package-summary.html` – Letzter Zugriff am 07.09.2019.

[38] Parr, Terence: *ANTLR.* `http://www.antlr.org/` – Letzter Zugriff am 07.09.2019.

[39] Stachowiak, Herbert: *Allgemeine Modelltheorie.* Springer–Verlag, Wien, 1973.

[40] Technische Universität Dortmund, Informatik Lehrstuhl 5: *CINCO SCCE Meta Tooling Framework.* `https://cinco.scce.info/` – Letzter Zugriff am 07.09.2019.

[41] Technische Universität Dortmund, Informatik Lehrstuhl 5: *Metaplugins.* `https://projekte.itmc.tu-dortmund.de/projects/cinco/wiki/Metaplugins` – Letzter Zugriff am 07.09.2019.

[42] Vlissides und Vlissides: *Pattern hatching: Design patterns applied.* Addison-Wesley Reading, 1998.

[43] Wirkner, D.: *Merge-Strategien für Graphmodelle am Beispiel von jABC und Git.* Februar 2015.