Universität Trier
Computerlinguistik & Digital Humanities

Dokumentation

des Projekts

Chatbot Augusta

9. Juli 2019

Ali Kemal Yalama
E-Mail: s2alyala@uni-trier.de

Julia Roegner
E-Mail: s2juroeg@uni-trier.de

Inhaltsverzeichnis

1 Einleitung

„Augusta“ ist ein Chatbot, dessen Aufgabe es ist, dem Kunden Produkte aus dem Uni-Shop zu empfehlen. Dafür muss der Nutzer dem Chatbot via Texteingabe angeben, was gesucht wird. Diese Nutzereingaben werden mit Hilfe von Pattern-Matching analysiert und via PostgreSQL-Schnittstelle werden Antworten basierend auf Queries generiert, die dem geäußerten Wunsch des Nutzers entsprechen.
Das Ergebnis des Projekts kann auf GitHub unter `https://github.com/AlKYala/Chatbot` eingesehen werden.

1.1 Ziele des Projekts

Vorrangiges Ziel des Projekts war der Versuch der Entwicklung eines computerlinguistischen Programms. Das Umsetzen von im Studium angeeigneter Fähigkeiten theoretischer und praktischer Natur, wie beispielsweise der Automatentheorie, Datenbankentechnologie und Softwaretechnik, hat eine essentielle Rolle in der Konzeption und Realisierung des Programms gespielt. Weiteres Ziel war das Einarbeiten in ein fremdes Framework mit eigenem Interpreter, hier ChatScript (Wilcox 2019) von Bruce Wilcox, um ein entsprechendes Programm zu entwickeln.

1.2 Der Chatbot Augusta

Im Projektseminar der Computerlinguistik haben wir in kleinen Gruppen jeweils einen Chatbot für eine spezifische Anwendung entwickeln wollen. Unsere Gruppe, bestehend aus Ali Yalama und Julia Roegner, hat sich dazu entschieden, einen Chatbot als Kaufberatung für den Uni-Shop der Universität Trier zu bauen. Der Bot sollte für das Deutsche gelten und eine ausschließlich beratende Funktion haben, das bedeutet, dass der Bot Bestellungen oder Fragen zum Uni-Shop nicht beantworten können muss.
Als Basis dienten die Beispieldialoge (s. Anhang), die auch am Ende dieser Dokumentation abgedruckt sind. Hier wird davon ausgegangen, dass der Nutzer noch nicht weiß, welches Produkt er haben möchte, sondern ihm nur bekannt ist, wofür er es möchte, also entweder eine Geschenkidee hat oder etwas für eine bestimmte Anwendung sucht. Dann bietet der Chatbot dem Nutzer die Kategorien an, die in der Datenbank für Anwendungsgebiet oder Geschenkidee hinterlegt sind. Im Weiteren kann der Kunde dem Chatbot noch weitere Informationen über den gesuchten Artikel wie eine Farbe oder den Preis übergeben, am besten in Form eines kompakten Satzes, den der Chatbot analysiert. Dann nutzt der Bot die extrahierten Stichwörter, um in einer Datenbank, in der die Informationen über die Produkte des Uni-Shops gespeichert sind, nach etwas Passendem zu suchen.
Der gefundene Gegenstand wird dem Kunden angeboten und der Kunde kann entscheiden, ob er ihn ablehnt, dann sucht der Bot erneut, oder ihn annimmt, woraufhin der Chatbot fragt, ob eine neue Suche mit anderen Kriterien durchführen soll. Wenn nicht, beendet sich der Bot.

Der Name des Bots, Augusta, leitet sich vom Namen der antiken Stadt *Augusta Treveorum* an der Mosel her, aus der das heutige Trier hervorgegangen ist.

1.3 Struktur der Dokumentation

In Kapitel 2 werden die einzelnen Versionen Augustas erläutert sowie die Installation beschrieben. In Kapitel 3 folgen dann einige grundlegende Regeln zum Schreiben von Chatbots in ChatScript, woran sich ein Kapitel zu der Programmlogik hinter unserem Chatbot anschließt, das Augusta als deterministischen Automaten vorstellt. Informationen zu der Datenbank *Uni-Shop*, in der die Produktinformationen der Artikel gespeichert sind, und ihrer Einbindung sowie der Suchanfragen finden sich in Kapitel 5. Auf die Benutzerschnittstelle wird in Kapitel 6 eingegangen. In Kapitel 7 werden schließlich einige Probleme erläutert, auf die wir gestoßen sind und im Rahmen des Projekts leider nicht mehr lösen konnten.
Am Ende der Dokumentation sind die Dialoge abgedruckt, die als Basis dienten, und die Zusammenfassung des Ergebnisses aus einem kleinen Usability Test.

2 Installation von Augusta

2.1 Installation

Zum Ausführen des Chatbots wird Folgendes benötigt:

1. Windows als Betriebssystem
2. Eine vorhandene Installation von PostgreSQL
3. Optional: Eine Möglichkeit Webseiten zu hosten, z.B. auf Localhost

Obwohl ChatScript auch auf Mac und Linux laufen kann, existiert ein Problem bei ChatScript mit PostgreSQL-Support. Sofern die GUI getestet werden soll, muss dem Tester eine Möglichkeit bereit stehen, Webseiten, gegebenenfalls auf Localhost, zu hosten.
Um den Bot zu testen, wird ChatScript benötigt. Das Framework mit der Dokumentation in der aktuellen Version ist unter `https://github.com/ChatScript/ChatScript` zu finden.
Die Adresse der Git-Repository des Projekts lautet `https://github.com/AlKYala/Chatbot`.
Während des Projekts wurde mit Version 9.31 (30.04.2019) von ChatScript gearbeitet.
Die Ordner *ONESENTENCEGOODBYE*, *ONESENTENCEGUI* und *PRODUKTEIGENSCHAFTEN* sind alle in den Ordner *RAWDATA* im ChatScript-Verzeichnis zu kopieren. Ein Ordner steht dabei für eine Version des Chatbots. Ebenso müssen die Dateien *filesonesentencegoodbye.txt*, *filesonesentencegui.txt* und *filesprodukteigenschaften.txt*, welche zum Kompilieren der jeweiligen Version benötigt werden.
Mithilfe der Datei *UniShop_Datenbank* muss in PostgreSQL die Datenbank *Uni-Shop* erstellt und das Passwort (hier: `1234`) für PostgreSQL an der jeweiligen Stelle in *kaufabsicht.top* eingetragen werden (pro Datei fünf Mal). Die Stellen sind durch den Kommentar `# DATENBANK` gekennzeichnet. Hier ein Beispiel:

```
1 # DATENBANK
2 if (^dbinit(dbname = Uni-Shop port = 5432 user = postgres password = 1234)
     )
3         {[Lass uns anfangen] [Super, auf geht's]!  ^reuse( FIRSTQ )
4 } else {dbinit failed - $$db_error ^reuse( FIRSTQ ) }
```

Da sich der Inhalt von *kaufabsicht.top* von Version zu Version unterscheidet, ist ein Kopieren der Datei in die anderen Versionen nicht zu empfehlen.
Um ChatScript zu starten, empfiehlt es sich, die Datei *ChatScriptpg.exe* im Ordner *BINARIES* auszuführen. Es ist dabei darauf zu achten, dass die .exe-Datei „pg“ im Namen hat, da nur hier PostgreSQL unterstützt wird. Zum Kompilieren muss in der Konsole der Befehl `:build CHATBOTNAME` eingegeben werden.
Sollte der Wunsch bestehen, die GUI zu testen, so muss die .bat-Datei *LocalPgServer.bat* aus dem Ordner *SERVER BATCH FILES* ausgeführt werden, um den Server zum Laufen zu bringen, und die Dateien *index.php*, *ui.php* und *UniTrier_Logo.png* müssen in das jeweilige Verzeichnis für den Server kopiert werden. Zuvor müssen allerdings sowohl die .bat-Datei als auch *ui.php* in Bezug auf das

PostgreSQL-Passwort, den Localhost und Anderes angepasst werden.
Für Anmerkungen zu den Schwierigkeiten mit dem PostgreSQL-Support, der Benutzerschnittstelle und anderen Betriebsystemen als Windows siehe Kapitel 7.

2.2 Versionen

In diesem Projekt finden sich verschiedene Ansätze für Chatbots wieder. Die Ansätze unterscheiden sich je nach Zweck. Im Folgenden ist eine kleine Übersicht gegeben. Details zur Programmlogik finden sich in Kaptiel 4.
Die Version „Firsttry“ stellt unseren ersten Versuch da, aus der sich die anderen Versionen entwickelt haben. Deshalb ist Firsttry in der jetzigen Git-Repository nicht mehr vorhanden, kann aber über die vorherigen Speicherzustände während der Entwicklung des Projekts eingesehen werden.
Die Kategorien der Anwendungsgebiete und Geschenkideen für FIRSTQ aus dem Topic Kaufabsicht sind in diesen Versionen direkt in den Code geschrieben. Sollte eine automatische Extraktion der Kategorien aus der Datenbank gewünscht sein, so lässt sich dies leicht umstellen. Nähere Information dazu finden sich in Abschnitt 5.3.4.

2.2.1 OnesentenceGoodbye

Dieser Ansatz ist für die Verwendung auf der Konsole geschrieben. Er erlaubt es dem Kunden, sich in jedem Zustand durch ein entsprechendes Stichwort vom Bot zu verabschieden. Dies hat den Vorteil, dass ein `drop`-Befehl für die Tabelle, die dem Kunden zugeordnet wird, erfolgt und somit ein spätere Komplikationen auf Datenbankebene vermieden werden.
Unter der erweiterten Anwendung eines Pattern-Matching-Verfahrens, das von ChatScript aus bereitgestellt wird, wird der Kundenwunsch anhand eines einzelnen Satzes extrahiert und analysiert. Dies soll einen intuitiveren Ansatz darstellen, da in natürlicher Sprache oft sogenannte Kundenwünsche in einem Satz beschrieben werden, ansatt als Antworten auf konkrete Fragen.
Dieser Ansatz lässt sich mit `:build ONESENTENCEGOODBYE` testen.

2.2.2 OnesentenceGui

OnesentenceGui ist OnesentenceGoodbye recht ähnlich. Dieser Ansatz ist für die GUI gedacht, die erfordert, dass der Nutzer die erste Eingabe, wie z.B. ’Hallo’ tätigt. Außerdem werden die Informationen zu dem geführten Dialog, die ChatScript speichert, um die neue Konversation dort fortführen zu können, wo die alte endete, wenn der Benutzer zurückkehrt, beim Programmstart gelöscht, da das Speichern für unsere Anwendung nicht sinnvoll ist. Damit das Löschen allerdings funktioniert, ist es notwendig, dass zuvor der Bot „ordentlich“ beendet wurde, also beispielsweise über ein `:quit` während des Dialoges, nicht aber durch das plötzliche Schließen der .bat-Datei, des Chatbot-Fensters im Browser oder Ähnliches.
Ursprünglich war eine Version vorgesehen, in der der Bot wie in OnesentenceGoodbye jederzeit mit einem Stichwort beendet werden kann und die Tabelle in der Datenbank gelöscht wird. Leider waren der

Befehl zum Löschen der Konversation (`:reset`) und die Funktion, den Bot jederzeit beenden können, aus einem für uns nicht ersichtlichen Grund nicht kompatibel. Diese Version trug den Namen „OnesentenceGuiEnde“ und wird wie Firsttry nicht ausgeliefert.
Ansonsten unterscheidet sich die Programmlogik nicht von OnesentenceGoodbye. Dieser Ansatz kann mit `:build ONESENTENCEGUI` kompiliert werden, ist aber weniger für das Testen in der Konsole und am besten für die Benutzung in der GUI geeignet.
Beim Arbeiten mit der Benutzerschnittstelle sind einige Schwierigkeiten aufgetaucht, weshalb OnesentenceGui mit der GUI nicht richtig funktioniert (allerdings ist die Version auf der Konsole funktionsfähig). Nähere Informationen zu den Problemen finden sich in 7.2.

2.2.3 Produkteigenschaften

Die Version Produkteigenschaften unterscheidet sich von OnesentenceGoodbye und OnesentenceGui in der Extraktion und Analyse des Kundenwunsches. Statt die Informationen zu Name, Art, Ausführung und Preis frei in einem Satz eingeben zu können, wird der Kunde explizit nach diesen Eigenschaften gefragt. Das einzelne Fragen diente ursprünglich nur als Übergangslösung bis das Pattern-Matching für die Eingabe mittels eines Satzes fertig wäre, aber da sich während eines Benutzertests (s. Anhang) gezeigt hat, dass die einzelenn Fragen als einfacher empfunden werden, haben wir diese Version belassen.
Produkteigenschaften ist für die Benutzung auf der Konsole geschrieben und verfügt **nicht** über die Möglichkeit, wie OnesentenceGoodbye jederzeit beendet zu werden.
Die Version kann über den Befehl `:build PRODUKTEIGENSCHAFTEN` aufgerufen werden.

3 ChatScript: Grundlagen

ChatScript ist ein Framework, das es Entwicklern erlaubt, regelbasierte Chatbots zu entwickeln. Es handelt sich hierbei um ein „Natural Language Tool“ mit eigenem Interpreter, welcher Code in C++ übersezt.

3.1 Regeln

Regeln sind ein elementarer Bestandteil in ChatScript. In diesen Regeln werden unter Anderem Nutzereingaben analysiert, Antworten des Bots festgelegt, Datenbankabfragen ausgeführt und der Gesprächsablauf gesteuert.
In Augusta finden sich Regeln in verschiedenen Formen wieder. Diese sind:

- `t:` für Ausgaben des Chatbots, ohne folgende Antwort
- `u:` für Ausgaben in Form von Fragen, das beduetet, dass auf Regeln mit u: eine Antwort des Nutzers folgen muss
- `a:`, `b:` bis `q:` für Antworten des Nutzers. Auf diese Regeln können Pattern-Matching-Ansätze angewandt werden, um Teile der Eingabe ggf. zu extrahieren. Sie finden sich in den Regeln `t:` und `u:`.

Regeln haben, wenn auch optional, meist einen Bezeichner, eine Bedingung zur Ausführung und es lassen sich dabei Variablen zuweisen. Ein abstraktes Beispiel:

```
t: BEZEICHNER (KONDITION) $meineVariable=Wert           Ausgabe des Bots
```

Im Folgenden einige Beispiele für Regeln:

```
t:  ( %input<%userfirstline )
    ^keep()
    [Hallo] [Hi] [Guten Tag], ich bin Augusta und kann dich beraten, wenn
       du etwas aus dem Online-Shop suchst. ^reuse( GREET )
```

In dieser Regel wird der Nutzer begrüßt und dann zur Regel GREET weitergeleitet. Der Dialog kann genauer mit den Antwortmöglichkeiten über `a:`, `b:` und so weiter wiedergegeben werden. Ein Beispiel dafür:

```
u: INTRO ($enter211) [Das ist gut] [Das ist toll] [Das freut mich], [
    dabei kann ich dir helfen] [ich kann dir dabei helfen, etwas zu finden
    ] [ich kann dich beraten]. Also dann, wollen wir loslegen?

        a: ( ~positiv ) $introyes = 1a
        # DATENBANK
            if (^dbinit(dbname = Uni-Shop port = 5432 user = postgres
               password = 1234))
```

```
6                   {[Lass uns anfangen] [Super, auf geht's]!  ^reuse( FIRSTQ
                        )
7               } else {dbinit failed - $$db_error  ^reuse( FIRSTQ ) }
8
9           a: ( ~negativ ) Tut mir Leid, ich kann eigentlich nur beraten.
                Bist du dir sicher, dass du nichts aus dem Shop möchtest?
10          # DATENBANK
11                  b: ( ~negativ )  $introyes = 1a if (^dbinit(dbname = Uni-
                        Shop port = 5432 user = postgres password = 1234 )) {
                        Schön zu hören. ^reuse( FIRSTQ )}
12                                      else {dbinit failed - $$db_error ^
                                            reuse(FIRSTQ)}
13            b: ( ~positiv ) $enterEnd2 = 1b Das ist schade. ^reuse( ~ende.
                ASKIFHAPPY )
```

In diesem Abschnitt fragt der Bot mit zufällig ausgewählter Frage, ob der Kunde etwas kaufen möchte. Falls die Eingabe des Kunden ein Wort aus dem Konzept positiv enthält, wird entsprechend die Datenbank geladen und die Regel FIRSTQ im selben Topic aktiviert. Ist die Eingabe allerdings einem Wort aus dem Konzept negativ zuzuordnen, so wird gefragt, ob der Kunde sich sicher ist. Je nach Antwort darauf wird eine entsprechende Regel, das heißt, eine der beiden `b:`-Regeln ausgeführt.

3.2 Topics

Die Quelldateien in einem ChatScript-Programm werden auch als „Topics“ bezeichnet und haben die Dateiendung .top. Topics können als Zusammenfassung mehrerer Regeln betrachtet werden. Daher bietet es sich an, Topics als Module eines Chatbots zu handhaben. So wird zum Beispiel in diesem Projekt der Datenbankabfrage ein eignenes Topic zugewiesen.
Topics beginnen immer mit einer Tilde („ “) als Präfix gefolgt vom Topicnamen, meist als erste Zeile in einer .top-Datei oder vor dem Auflisten der Regeln:

```
1 topic: ~dbsearch  [] ($gosearch)
```

Gegebenenfalls findet sich vor dem Topicnamen noch die Definition eines Outputmacros, das in dem Topic verwendet wird. Für weitere Informationen über Outputmacros siehe Unterkapitel 5.2.2.

3.3 Konzepte

Syntaktisch werden Konzepte wie Topics gehandhabt, das heißt, dass Konzepte mit `~konzeptname` referenziert werden. Ein Konzept kann abhängig Einträgen als eine Menge von Synonymen, Hyponymen oder beidem verstanden werden, ähnlich zu den Einträgen in einem Thesaurus. Obwohl es sich hier um Mengen handelt, stehen diese aus syntaktischer Sicht in eckigen Klammern („[]“). Ein Beispiel anhand des Konzepts ~ciao, das Synonyme beschreibt:

```
1 concept: ~ciao [ciao tschüss "good bye" bye "daje" "eddi un merci" "äddi a
      merci"]
```

Anzumerken ist, dass Ausdrücke aus mehreren Wörtern, sogenannte „multiword expressions“, in Anführungszeichen stehen müssen, da die einzelnen Bestandteile dieser sonst als eigenständige Elemente betrachtet werden.

In „Augusta“ sind Konzepte besonders wichtig, da es sich anbietet, das Pattern-Matching von Konzepten abhängig zu machen. So wird mit Hilfe von Konzepten das Arbeiten mit der PostgreSQL-Schnittstelle gewährleistet: Für jeden möglichen Eintrag pro Spalte in der PostgreSQL-Datenbank existiert ein Konzept. Als Beispiel alle Einträge für die Spalte „Anwendungszweck“ (zur Datenbank s. 5.1), hier handelt es sich also eher um Hyponyme:

```
concept: ~anwendungszweck [Unterhaltung Schreibwaren EssenTrinken
    Essentrinken Alltag Accessoire]
```

Der Nutzer beschreibt beispielsweise in einem Satz, welchen Anwendungszweck das gesuchte Produkt erfüllen soll. Wird ein Wort aus dem Konzept gefunden, so wird dieses nach den Vorgaben des Skriptes als solches erkannt. Damit wird sichergestellt, dass in den Queries nur Eigenschaften vorkommen, die in Datenbankeinträgen existieren. Sollte der Anwender nach Eigenschaften suchen, die nicht existieren, so werden diese ignoriert (zum Pattern-Matching s. 3.7 und zur Query s. 5.3).

Des Weiteren erlaubt es ChatScript, mehrere verschiedene Konzepte in einem Konzept zu vereinen. Beispielsweise ist das Konzept ~positivkaufen als Oberkategorie, bestehend aus drei anderen Konzepten, die bedeutungsähnlich sind, zu verstehen:

```
concept: ~positivkaufen [ ~yes ~zustimmung ~kaufen]
```

~positivkaufen enthält somit alle Wörter, die in ~yes, ~zustimmung und ~kaufen vorkommen.

3.4 ChatScript: Variablen

Variablen in Augusta kommen größtenteils in der Form `$variablenname` vor. Das „`$`“ vor dem Identifier bedeutet, dass diese Variable gloabl ist und über die Regel hinaus existiert. Variablen werden für gewöhnlich im Kopf einer Regel instanziiert. Bei den zugewiesenen Werten kann es sich auch etwas handeln, das mit Hilfe von Pattern-Matching gefunden und aus der Antwort des Benutzers extrahiert wurde. Im folgenden Beispiel wird das erste Wort der Nutzereingabe, welches sich im Konzept ~geschenkidee befindet, in der Variable `$geschenkidee` gespeichert.

```
a: ( _~geschenkidee ) $anwendungszweck = ^"'nichts'" $geschenkidee = ^"''
    _0'"}
```

Bereits instanziierte Variablen lassen sich mit dem Befehl `^walkvariables(^outputmacro)` einsehen, sofern ein entsprechendes Outputmacro vorhanden ist.
Da das Zurücksetzen von Variablen nicht trivial ist, lassen sich diese mit dem Befehl `^reset( VARIABLES )` löschen. Allerdings ist dieser Befehl sehr mächtig, da er fast alle Variablen (außer $_-Variablen) zurücksetzt. In der Dokumentation (S. 54 in ChatScript - Sytem-Functions-Manual in Wilcox (2019)) wird zwar daraufhingewiesen, dass $$-Variablen und Bot-Variablen bestehen bleiben, in unserem Fall hat sich das allerdings nicht bewahrheitet.

Wenn man verhindern möchte, dass eine Variable zurückgesetzt wird, empfiehlt es sich diese als temporäre Variable wie folgt zwischenzuspeichern:

```
1         $_cs_bottmp = $cs_bot
2     ^reset( VARIABLES )
3     #....
4         $cs_bot = $_cs_bottmp
```

Dieser Schritt findet in dem Topic Ende statt. Dies ist nötig, um zu gewährleisten, dass sowohl Informationen wie der Name des Kunden als auch die Instanz des Bots nicht verloren gehen, wenn eine weitere Empfehlung im selben Gespräch erfolgen soll.

3.5 Variablen als Bedingungen zur Steuerung des Programmablaufs

Nativ wählt ChatScript Regeln in einem Topic zufällig aus, um durch themenbezogene zufällige Antworten natürlich zu wirken. Da Augusta, ein Chatbot zur Kaufberatung, jedoch eine lineare Konversation gewährleisten muss, wird in fast allen Regeln mit Bedingungen gearbeitet. Der Regelkopf aus FIRSTQ aus dem Topic Kaufabsicht als Beispiel:

```
1 u: FIRSTQ ($introyes) Wir können nach etwas zum ...
```

Die Regel FIRSTQ darf nur aktiviert werden, sofern die Variable $introyes existiert.

Ähnlich lässt es sich auch verhindern, dass der Bot in eine Regel übergeht. Es wird eine Variable, die in keiner Regel instanziiert wird, als Bedingung gestellt, wie in STARTKAUF aus Kaufabsicht:

```
1 t: STARTKAUF ($enter999) ^reuse( INTRO)
```

Die Regel STARTKAUF ist gewöhnlich nicht betretbar für den Bot, da `$enter999` als Variable nie instanziiert wird. Selbiges gilt für alle Regeln, deren Bedingung die Existenz einer Variable mit `$enter...` erfordert. Diese Bedingungen dienen dazu, um ein zufälliges Springen des Bots zu verhindern. Hier muss manuell von einer anderen Regel auf diese weitergeleitet werden, was mit Hilfe von Befehlen zur Steuerung des Programmablaufs erfolgt.

Alternativ, jedoch nicht so häufig, ist die Bedingung, dass ein Wort eines Konzepts zu erwähnen ist, damit eine Regel betreten wird. Als Beispiel der Regelkopf einer Regel aus Kaufabsicht:

```
1 b: (~willGeschenk) $anwendungszweck = ^"'nichts'" Du möchtest etwas
     verschenken. Ich kann dir folgende Kategorien anbieten:
```

Diese Regel, sofern erreichbar, erfordert von der Nutzereingabe, dass ein Wort aus dem Konzept ~willGeschenk erwähnt wird. Ist dies nicht der Fall, so wird diese Regel nicht betreten.

3.6 Befehle zur Steuerung von Programmablauf

In Augusta wurden zwei Methoden genutzt, um den Gesprächsverlauf zu lenken. Dabei handelt es sich zum einen um `^gambit(~topic)` und zum anderen um `^reuse(~topic.rule)`. Im Laufe des

Projekts hat sich `^reuse(~topic.rule)` als praktischer herausgestellt, da das direkte Springen in Regeln desselben Topics als auch in Regeln anderer Topics möglich ist. Des Weiteren ignoriert `^reuse` die Bedingungen der Zielregel, sodass Regeln, unabhängig davon, ob ein Wort des geforderten Konzepts oder die geforderte Variable existiert, betreten werden können. Sofern sich eine Regel im selben Topic befindet, muss in `^reuse` nur die Zielregel angegeben werden:

```
t: STARTKAUF ($enter999) ^reuse( INTRO)
```

Wird die Regel STARTKAUF betreten, so wird in die Regel INTRO desselben Topics übergegangen.

```
a: (~positiv) $gosearch = 1c Alles klar! Ich suche mal im Shop. ^reuse( ~
   dbsearch.WELCOME )
```

Sofern ein Wort aus dem Konzept ~positiv aus der Nutzereingabe erkannt wird, so wird diese Regel ausgeführt und anschließend die Regel WELCOME in Dbsearch, einem anderen Topic, betreten.

Im Verlauf der Entwicklung wurde `^gambit` mit `^reuse` ersetzt, da `^reuse` eine präzisere Steuerung des Programmablaufs erlaubt. Während `^gambit` lediglich in eine anderes Topic springt und dieses von der ersten betretbaren Regel an ausführt, lässt sich durch `^reuse` festlegen, welche Regel zu betreten ist, ohne auf die Bedingung der Zielregel achten zu müssen.

3.7 Pattern-Matching

ChatScripts Interpreter bietet einen hauseigenen Pattern-Matcher zum Vergleichen von Nutzereingaben. Diese Ansätze eignen sich an erster Stelle dazu, Bedingungen für Regeln zu schreiben. Die einfachste Form, um eine Nutzereingabe abzugleichen, ist, zu überprüfen, ob Nutzereingaben ein aus Konzepten bekanntes Token enthalten. Die folgende Regel überprüft, ob die Nutzereingabe ein Wort aus dem Konzept ~positiv enthält. Ist dies der Fall und ist die Regel erreichbar, so wird die folgende Regel ausgeführt:

```
c: ( ~positiv ) $enterEnd2 = 1b Das ist schade. ^reuse( ~ende.ASKIFHAPPY )
```

Es besteht des Weiteren die Möglichkeit, eine Nutzereingabe auf mehrere Konzepte zu überprüfen:

```
b: ( [~no ~nicht] ) $enterEnd2 = 1b In Ordnung. ^reuse( ~ende.ASKIFHAPPY )
```

Außerdem stellt ChatScript Mittel bereit, um Aussagen über die Reihenfolge von Wörtern in einer Nutzereingabe zu treffen. In der Regel KEINE_Vorstellung aus Introductions wird überprüft, ob das Wort „nicht“ in derselben Nutzeringabe wie ein Wort aus dem Konzept ~sagen vorkommt. Dabei wird nicht eingeschränkt, an welchem Index die Wörter in der Nutzereingabe vorkommen, sodass Nutzereingaben wie „Das will ich nicht verraten“ oder „Nicht das will ich verraten“ von dieser Bedingung akzeptiert werden.

```
a: KEINE_VORSTELLUNG (<<[sagen] nicht>>) [Das verstehe ich] [Das kann ich
   nachvollziehen]. ^reuse(~kaufabsicht.STARTKAUF)
```

Es ist auch möglich, Wörter aus Nutzereingaben zu herauszusuchen. Für diesen Zweck bietet ChatScript Wildcards der Form `*n`. Diese erlauben es, Token an einer bestimmten Stelle zu extrahieren und gegebenenfalls in einer Variable zu speichern.

```
a: VORSTELLUNG ([heiße bin ist lautet] _*1 >)
            if ($cs_token == $stdtoken)
           {
           $cs_token = #DO_INTERJECTION_SPLITTING |
                       #DO_SUBSTITUTE_SYSTEM | #DO_NUMBER_MERGE |
                       #DO_PARSE
           retry(SENTENCE)
           }
        $kunde =  pos(noun '_0 proper)
```

Im Grunde wird hier das erste Wort nach „heiße“, „bin“, „lautet“ oder „ist“ genommen und in der Variable `$kunde` gespeichert.

Wildcards spielen in Augusta eine zentrale Rolle, wenn es darum geht, Kundenwünsche zu verarbeiten. In KeyExOnesentence finden sich verschiedene Muster zum Verständnis von Kundenwünschen. Dabei wird darauf geachtet, dass Wörter erwähnt werden, die in den Konzepten vorkommen, sodass eine Query anhand der gegebenen Wörtern möglich ist:

```
    a: ( !!~positiveinteger * _~thingsandart {in} _~ausfuehrung * {_~
       positiveinteger} *  )
      # Test, ob _0 Name oder Art ist
      if (pattern _0~things) {$things = ^"'' _0'" zusammenfassung = ^join(
         $zusammenfassung ^" Das Produkt heißt $things.")
      } else {$art = ^"'' _0'"  $zusammenfassung = ^join($zusammenfassung
         ^" Dein Produkt fällt unter den Obergriff $art.")}
      $ausfuehrung = ^"'' _1'"
      # Optionaler Preis
      if ( _2 AND ^isnumber(_2) ) {$preis = ^"'' _2'" $zusammenfassung = ^
         join($zusammenfassung ^" Es ist in $ausfuehrung und maximal soll
         es $preis Euro kosten.")
      } else {$zusammenfassung = ^join($zusammenfassung ^" Es soll in
         $ausfuehrung sein und eine preisliche Obergrenze hast du nicht
         angegeben.")}
      ^reuse( SUMMARY )
```

In diesem Pattern, dem 1. Muster aus der Regel, wird anfangs vorgegeben, dass eine positive Zahl nicht am Anfang stehen darf. Es können beliebig viele Token kommen, bis das erste Token aus dem Konzept ~thingsandart erwähnt wird, optionaler Weise gefolgt von einem 'in' und einem Begriff aus dem Konzept ~ausfuehrung. Darauf dürfen dann beliebig viele weitere Token folgen. Es ist freigestellt, ob danach noch eine positive Zahl für den Preis angegeben wird.
Dabei stehen `_0`, `_1`, `_2` und so weiter für das erste, zweite oder dritte erkannte Wort aus den Konzepten. Im Folgenden wird in einer if-Abfrage ermittelt, ob es sich bei dem ersten Wort um ein Token aus dem Konzept ~things oder ~art handelt und entsprechend eine Rückmeldung gegeben.
Insgesamt 9 verschiedene Muster dieser Art kommen in keyonesentence vor, aufgrund der Tatsache, dass der Nutzer frei angeben kann, ob und wie er Angaben zu Name des Gegenstands, Art, Ausführung oder dem Preis macht.

3.8 Zufällige Ausgabe

ChatScript erlaubt es, dass an einer Stelle mehrere ungesteuerte Ausgaben durch den Bot möglich sind. Mögliche Ausgaben werden mit Hilfe von [] unterschieden:

```
1 u: INTRO ($enter211) [Das ist gut] [Das ist toll] [Das freut mich], [dabei
      kann ich dir helfen] [ich kann dir dabei helfen, etwas zu finden] [ich
      kann dich beraten]. Also dann, wollen wir loslegen?
```

Hier wählt der Bot zufällig aus, ob „Das ist gut“, „Das ist toll“ oder Anderes ausgegeben wird, gefolgt von „Also dann, wollen wir loslegen?“.

4 Augusta: Programmlogik

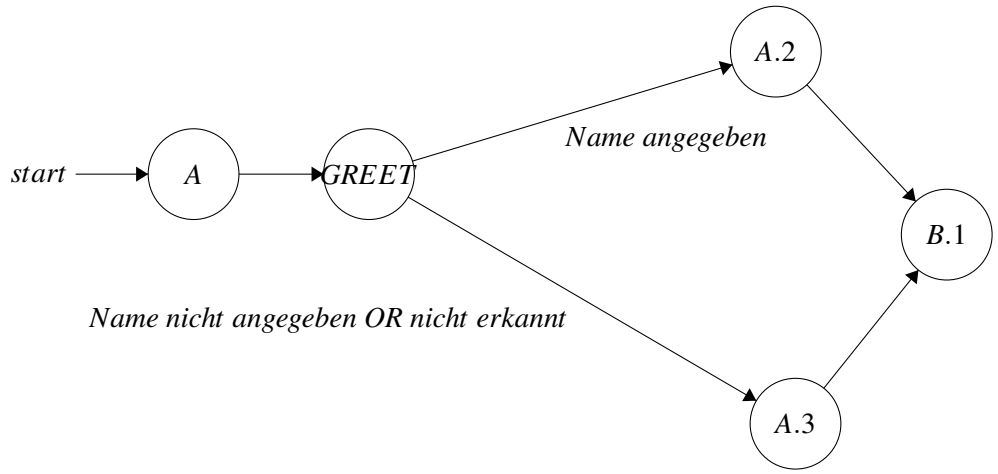
Da Augusta im Grunde ein deterministischer Automat ist, bietet es sich an, den Programmablauf als Automaten zu beschreiben. Im Folgenden wird der Programmablauf als Automat auf einzelnen Topics aufgeteilt dargestellt. Dabei beschreiben Zustände die jeweiligen Regeln in Augusta.

4.1 Introductions

Folgendes gilt:

1. A ist die erste Regel, die in Introductions aufgerufen wird
2. A.2 ist ~introductions.VORSTELLUNG
3. A.3 ist ~introductions.KEINE_VORSTELLUNG
4. B.1 ist ~kaufabsicht.STARTKAUF

In Introductions grüßt der Bot zu Beginn den Nutzer, geht in Regel GREET über und fragt, ob dieser seinen Namen verraten möchte. Je nachdem, ob bei der Nutzereingabe ein Name erkannt wird (siehe Abschnitt: 3.7) wird in die Regel VORSTELLUNG beziehungsweise KEINE_VORSTELLUNG übergegangen. Daraufhin wird die Regel STARTKAUF des Topics Kaufabsicht betreten.

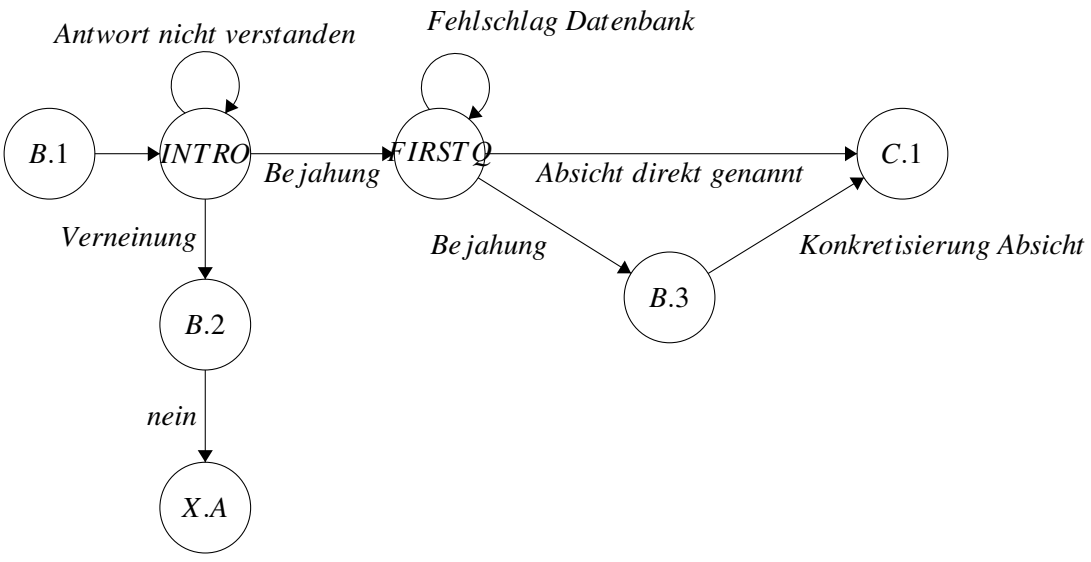


4.2 Kaufabsicht

Folgendes gilt:

1. B.1 ist STARTKAUF
2. B.2 ist die Nachfrage, ob ein Kaufwunsch besteht
3. B.3 ist die Regel nach Angabe, dass man ein Geschenk bzw. etwas zu einer Anwendung möchte
4. X.A ist ~ende.ASKIFHAPPY
5. C.1 ist keyexprodukteigenschaften.STARTPRODUKTEIGENSCHAFTEN bzw. keyexonesentence.STARTKAUF
6. Die restlichen Zustände sind nach ihren Regeln benannt

Von der Regel STARTKAUF geht der Bot in INTRO über und fragt, ob ein Kaufwunsch besteht. Dies wird so lange erfragt, bis der Bot die Antwort des Nutzers verstanden hat.
Besteht kein Kaufwunsch, erfolgt erneute Nachfrage. Bestätigt der Nutzer, dass er nichts kaufen möchte, so wird dieser zu Regel ASKIFHAPPY im Topic Ende weitergeleitet. Besteht doch ein Kaufwunsch, so wird er in die Regel FIRSTQ weitergeleitet. Sollten Probleme mit der Initialisierung der Datenbank auftreten, so wird dieser Zustand so lange wiederholt, bis die Datenbank aktiviert werden kann.
In FIRSTQ wird der Nutzer gefragt, ob dieser etwas zum Verschenken oder anhand eines Anwendungszweck sucht. Nennt der Nutzer nur, dass er ein Geschenk beziehungsweise etwas für einen gegebenen Zweck sucht, so wird ihm vorgeschlagen, welche Geschenkideen beziehungsweise Anwendungszwecke zur Verfügung stehen. Der Nutzer muss eine der aufgelisteten Geschenkideen oder Anwendungszwecke bennen, welche als Variable abgespeichert werden. Nennt der Nutzer stattdessen direkt, was für eine Geschenkidee oder Anwendungszweck er möchte und ist dies auch bekannt, dann muss die Absicht nicht weiter konkretisiert werden und wird in einer Variable abgespeichert.
Danach betritt der Bot die Regel keyexprodukteigenschaften.STARTPRODUKTEIGENSCHAFTEN bzw. keyexonesentence.STARTKAUF, abhängig vom vorliegenden Ansatz (beide Ansätze im Projekt enthalten, s. 2.2).

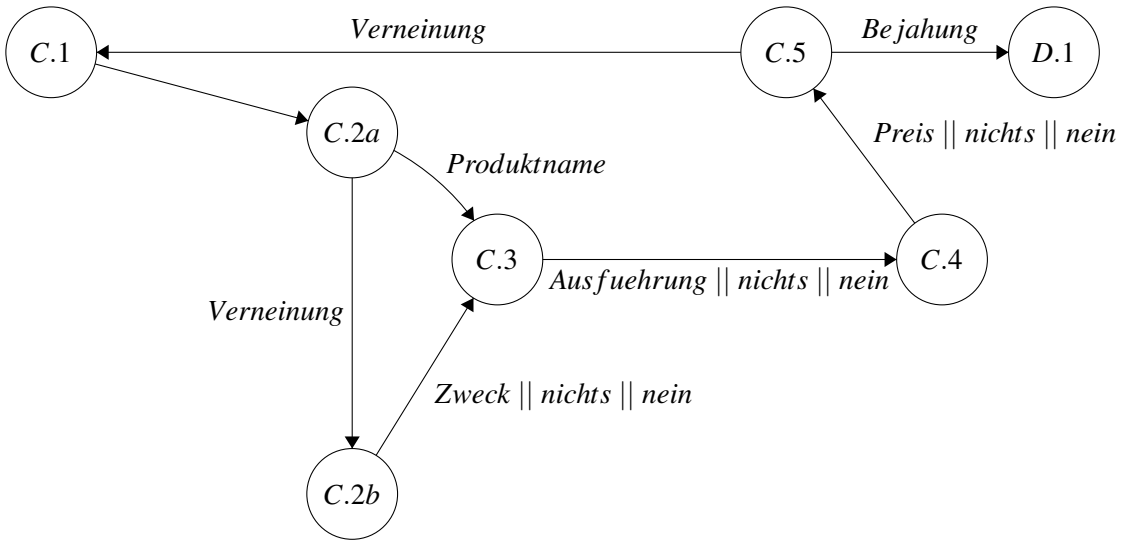


4.3 KeyExProdukteigenschaften

Folgendes gilt:

1. C.1 ist STARTPRODUKTEIGENSCHAFTEN
2. C.2a ist ASKNAME
3. C.2b ist KIND
4. C.3 ist DESIGN
5. C.4 ist PRICE
6. C.5 ist SUMMARY
7. D.1 ist ~dbsearch.WELCOME

Wird die Version Produkteigeschaften gewählt, so erfolgt ein einzelnes Erfragen der jeweiligen Eigenschaften.
In der Regel gilt: Werden in den Eingaben Stichworte erkannt, die als Eigenschaften in der Datenbank bekannt sind, so werden diese in Variablen gespeichert (s. 3.7).
Zunächst wird gefragt, ob der Nutzer sein gewünschtes Produkt beim Produktnamen kennt. Ist dies nicht der Fall oder wird die Eingabe nicht als Name identifiziert, wird gefragt, ob der Nutzer weiß, von welcher Kategorie, zum Beispiel Kleidung oder Buch seine Idee ist. Erfolgt eine Eingabe, die im Konzept ~art bekannt ist, so wird in der entsprechenden Variable `$art` dieser Wert gespeichert.
Daraufhin wird gefragt, ob der Kunde etwas zur Ausführung, zum Beispiel Farbe oder Sprache, sagen kann. Wird eine bekannte Ausführung erkannt, so wird diese gespeichert, ansonsten wird die Eingabe ignoriert.
Im Anschluss wird erfragt, welchen Preis man maximal zahlen möchte. Hierbei muss als Preislimit eine positive Integerzahl angegeben werden.
In SUMMARY wird zusammengefasst, was der Bot erkennen konnte und fragt den Nutzer, ob dies seinem Wunsch entspricht. Ist dies der Fall, so betritt der Bot die Regel WELCOME in Dbsearch, ansonsten wird bei STARTPRODUKTEIGENSCHAFTEN von Vorne angefangen. Für die Zusammenfassung wurden ab FIRSTQ ein entsprechender Satz in der Variable `$zusammenfassung` gespeichert, sodass in SUMMARY ein zusammenhängender Text ausgegeben werden kann.

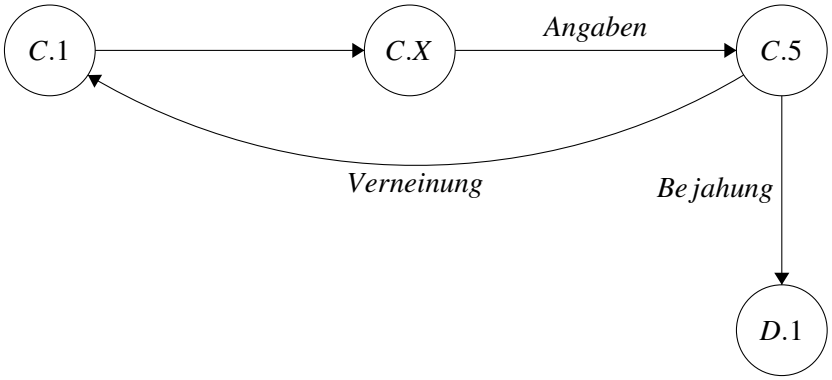


4.4 KeyExOnesentence

Folgendes gilt:

1. C.1 ist STARTONESENTENCE
2. C.X ist QUALITY
3. C.5 ist SUMMARY
4. D.1 ist ~dbsearch.WELCOME

In KeyExOnesentence kommt der in Abschnitt 3.7 zum Pattern-Matching besprochene Ansatz zur Extrahierung mehrerer Informationen aus einem einzelnen Satz zum Tragen.
In QUALITY anhand der Nutzereingabe ermittelt, um welches der insgesamt 11 Muster es sich dabei der Nutzereingabe handelt und die Informationen entsprechend extrahiert. Es gibt 9 Muster zur Eingabe der Produkteingenschaften und 2 Muster für den Fall, dass der Kunde keine Angaben machen kann.
In SUMMARY wird zusammengefasst, was der Bot erkennen konnte und fragt den Nutzer, ob dies seinem entspricht. Ist dies der Fall, so betritt der Bot die Regel WELCOME in Dbsearch, ansonsten wird bei STARTPONESENTENCE von Vorne angefangen. Für die Zusammenfassung wurden ab FIRSTQ ein entsprechender Satz in der Variable `$zusammenfassung` gespeichert, sodass in SUMMARY ein zusammenhängender Text ausgegeben werden kann.



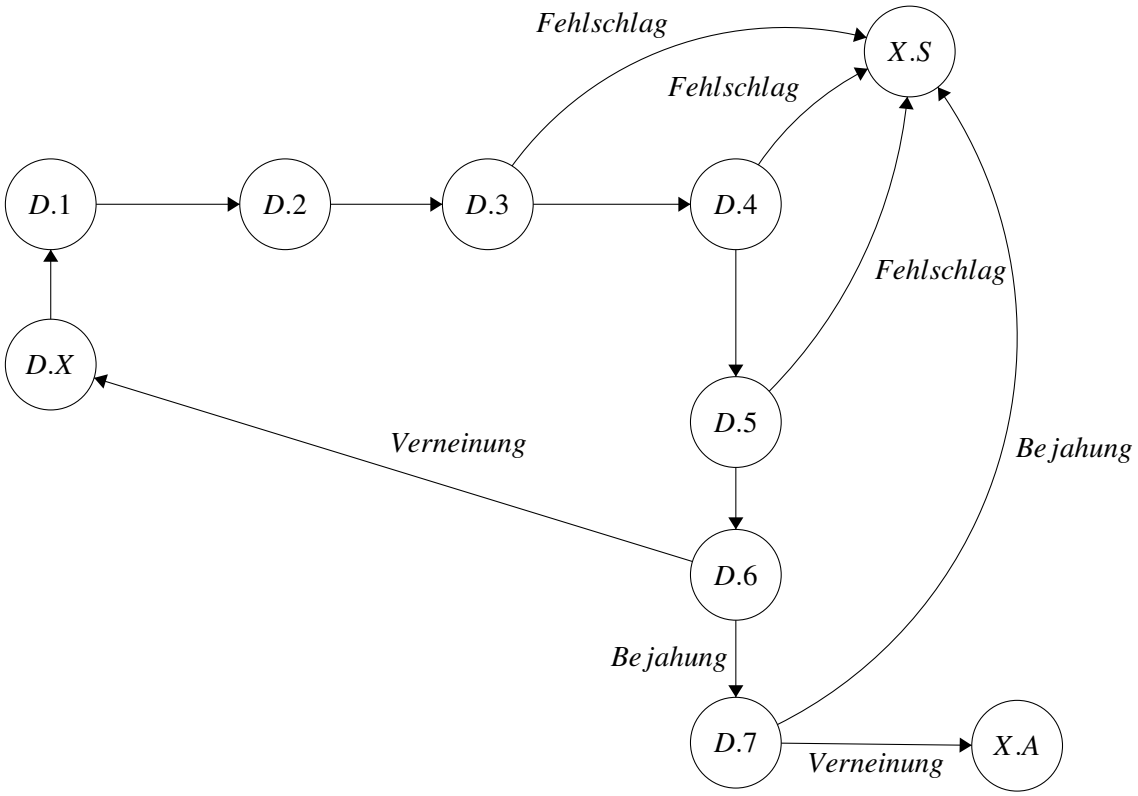
4.5 Dbsearch

Folgendes gilt:

1. D.1 ist WELCOME
2. D.2 ist SEARCHCREATE
3. D.3 ist SEARCHPRODUCT
4. D.4 ist PRODUCTNAME
5. D.5 ist PRODUCTDESCR
6. D.6 ist PRODUCTPRICE
7. D.7 ist RESPONSE

8. D.X ist SEARCHAGAIN
9. X.S ist ~ende.SETNULLREP
10. X.A ist ~ende.ASKIFHAPPY

Basierend auf den ermittelten Nutzerinformationen wird nun eine Query erstellt. Dabei wird für jeden Nutzer in der Regel SEARCHCREATE eine eigene Tabelle erstellt, die eine Kopie der originalen Datenbank ist.
Daraufhin wird der Schritt SEARCHPRODUCT betreten, in der nach Artikeln, die der Beschreibung des Nutzers entsprechen, gesucht wird. In den den folgenden drei Regeln PRODUCTNAME, PRODUCTDESCR und PRODUCTPRICE wird der Name, die Beschreibung beziehungsweise der Preis eines der gefundenen Produkte aus der Datenbank vorgestellt.
In ReESPONSE wird gefragt, ob das Produkt dem Nutzerwunsch entspricht. Ist dies nicht der Fall, so wird in SEARCHAGAIN übergegangen und erneut gesucht, wobei das bereits gefundene Produkt aufgrund Eigenschaft queried in der Datenbank (s. 5.3.1) nicht erneut gefunden werden kann. Ist der Nutzer hingegen zufrieden, so wird dieser gefragt, ob er noch ein Produkt suchen, bereits ausgewählte Artikel einsehen möchte oder ob er fertig ist.
Ist Ersteres der Fall, so wird in die Regel SETNULLANDREP im Topic Ende weitergeleitet, sodass eine erneute Suche möglich ist. Ist Zweiteres der Fall, so wird in der Datenbank erneut anhand der queried-Eigenschaft eine Anfrage für alle Produkte aufgerufen, die akzeptiert worden sind. Will der Nutzer kein weiteres Produkt mehr suchen, so wird die Regel ASKIFHAPPY in Ende betreten.

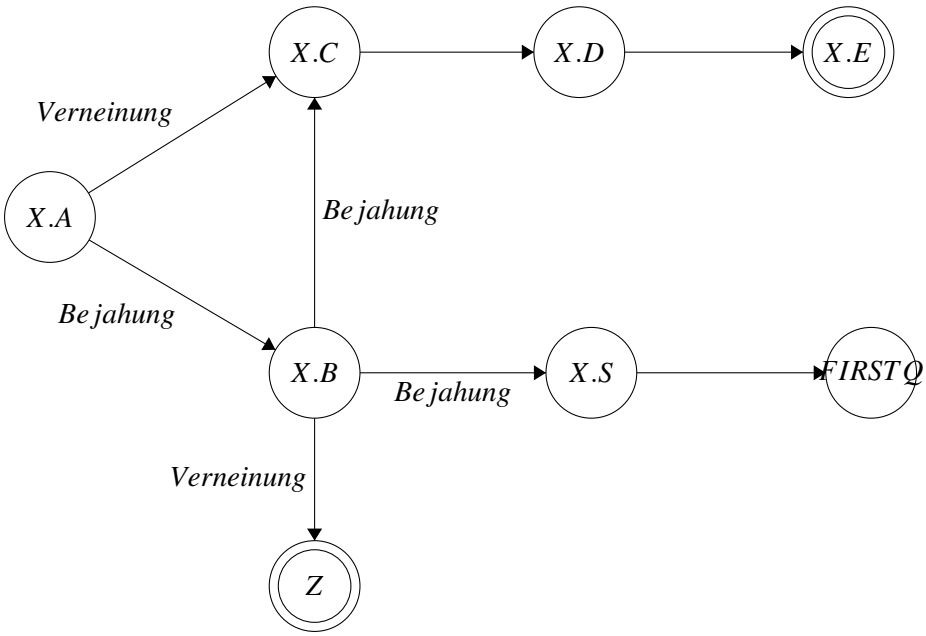


4.6 Ende

Folgendes gilt:

1. X.A ist ASKIFHAPPY
2. X.B ist STILLHELP
3. X.C ist das Ablehnen einer weiteren Suche
4. X.D ist fgt
5. X.E ist GOODBYE
6. X.S ist SETNULLREP
7. FIRSTQ ist FIRSTQ aus Kaufabsicht
8. Z ist ein anderer Endzustand

Wird die Regel ASKIFHAPPY in Ende betreten, so wird dem Nutzer die Frage gestellt, ob der Bot noch was für ihn tun könne oder nicht.
Ist dies nicht der Fall, so wird im Folgenden die eigene Tabelle des Nutzers gelöscht und der Bot beendet (X.E). Ist dies jedoch der Fall, so wird in STILLHELP gefragt, ob man eine neue Suche starten möchte. Wenn nicht, so wird der Bot beendet. Andererseits wird in SETNULLUNDREP übergegangen, wo die für die Kaufberatung relevanten Variablen zurückgesetzt werden und anschließend FIRSTQ im Topic Kaufabsicht zur erneuten Suche eines Produkts betreten.



5 Datenbank und ihre Einbindung in den Chatbot

Herz des Dialogsystems ist eine Datenbank, in der die Produkte des Uni-Shops sowie Informationen zu ihnen wie Beschreibung oder Preis gespeichert sind.
Der folgende Abschnitt behandelt den Aufbau der Datenbank des Uni-Shops sowie ihre Einbindung in den Chatbot.

5.1 Aufbau der Datenbank

Für die Kundenberatung war eine Datenbank erforderlich, auf die der Bot zurückgreifen kann, um einzelne Produkte zu empfehlen. Die Datenbank *Uni-Shop*, die mittels des SQL-Skripts in *UniShop_Datenbank.sql* in PostgreSQL gebaut wird, besteht aus den drei Relationen *Artikel*, *Anwendung* und *Geschenk* sowie der Relation *PseudotabellePro*, die sich wiederum aus den drei anderen zusammensetzt. Während des Dialoges erstellt der Chatbot in Uni-Shop noch für jeden Nutzer eine weitere Relation, *mystringXXXX*, die vor der Beendigung des Bots automatisch gelöscht wird.
Die Daten der Artikel betreffend Name, Preis, Beschreibung und Ausführung sind größtenteils dem Uni-Shop der Universität Trier (`https://www.uni-trier.de/index.php?id=50041`, Stand: Mai 2019) entnommen. Alle Produkte, die dort angeboten werden, finden sich auch in der Datenbank. Um eine größere Auswahl an Produkten zur Verfügung zu haben, was unter Anderem dem Testen des Bots dienlich ist, wurden noch eigene, fiktive Einträge hinzugefügt. So gibt es den Einkaufschip, die Bücher und die Spiele nicht im originalen Universitätsshop. Zusätzlich wurde für einige Produkte, wie dem Schlüsselband oder der Kaffeetasse, noch eine weitere Farbe eingefügt.
Die Einträge für das Anwendungsgebiet und die Geschenkideen (s. 5.1.2 und 5.1.3) beruhen auf unserer Vorstellung. Selbstverständlich sind hier andere Interpretationen möglich. Zudem kann die Datenbank jederzeit um weitere Einträge in den Relationen *Artikel*, *Anwendung* und *Geschenk* erweitert werden, wonach jedoch die Relation *PseudotabelePro* neu erstellt werden muss.
Da es bei den Relationen *Anwendung* und *Geschenk* mehrwertige Abhängigkeiten gibt, befindet sich die Datenbank in der vierten Normalform.

5.1.1 Die Relation *Artikel*

In der Relation *Artikel* sind die grundlegenden Informationen zu den einzelnen Produkten des Uni-Shops gespeichert. Jedes Produkt wird über eine eindeutige Artikelnummer, einen Namen, eine Kategorie (Art), einen Preis, eine kurze Beschreibung, sowie eine optionale Art der Ausführung näher bezeichnet.
Die Artikelnummer stellt, da sie für jeden Artikel einzigartig ist, den Schlüssel für ein Produkt dar. Die Nummer ist sechsstellig und setzt sich aus drei fortlaufenden Ziffern, einem Buchstaben für die Kategorie und zwei Ziffern oder Buchstaben für die Ausführung zusammen.

ArtNr	Name	Art	Preis	Beschreibung	Ausführung
001D01	Doktorhut	Deko	6.00	Schwarzer Doktorhut mit Siegel-Aufdruck der Universität Trier.	Schwarz
002A04	Schlüsselband	Alltag	2.00	Blau mit Logo und Internetadresse.	Blau
003A05	Schlüsselband	Alltag	2.00	Grün mit Logo und Internetadresse.	Grün
004D00	Wein-Set	Deko	9.50	Wein-Set mit Schachspiel „Checkmate“ bestehend aus Kellnermesser mit Gravur „Universität Trier“, Tropfring, Weinthermometer, Flaschenverschluss und Schachfiguren aus Holz, in schwarzer Holzbox mit Spielbrett auf dem Deckel.	

Tabelle 5.1: Auszug aus der Relation *Artikel*

Der Name der Produkte ist nicht eindeutig, es kommt also vor, dass Artikel in, zum Beispiel, verschiedenen Farben unterschiedliche Artikelnummern haben, aber denselben Namen tragen.
Die Kategorie des Produkt wird in der Relation mit *Art* bezeichnet. Hierbei handelt es sich um eine Art Hyperonym, dem das Produkt zuzuordnen ist. So gehört die Kaffeetasse beispielsweise zur Kategorie „Alltag“ und das Notizbuch zum Oberbegriff „Uni“.
Der Preis, wird, wie im englischen Sprachraum üblich, mit einem Punkt, statt, wie in Deutschland, mit einem Komma angegeben, was dem der Struktur des PostgreSQL geschuldet ist. Zudem ist vorgegeben, dass der Preis maximal nur zwei Vorkomma-Stellen (d. h. weniger als 100,00 Euro) haben darf.
Bei der Beschreibung ist es wichtig zu beachten, dass diese aus nicht mehr als 30 Wörtern insgesamt besteht, da sie andererseits nicht vollständig über den Chatbot ausgegeben werden kann. Zu näheren Informationen hierzu siehe 5.2.
Die Ausführung behandelt bestimmte Eigenschaften eines Produkts, die für den Käufer relevant sind. Hierzu zählt die Farbe oder, bei Büchern, die Sprache. Zu beachten ist, dass nicht bei jedem Produkt ein Eintrag für die Ausführung vorliegt.
Die Größe eines Artikel, was im Fall der Kleidungsstücke für den Kunden von Bedeutung ist, wird in der Datenbank nicht berücksichtigt. Für diese bewusste Entscheidung gibt es mehrere Gründe: Da der Chatbot nur eine Kundenberatung darstellt und keine Bestellmöglichkeit beinhaltet, muss der Kunde persönlich beim Uni-Shop erscheinen, um das Produkt zu erwerben. Dort wird er über die vorhandenen Größen informiert und kann das Kleidungsstück möglicherweise auch anprobieren. Da Kleidunggrößen generell sehr unterschiedlich ausfallen können, gerade bei Unisex-Kleidung, ist der Aussagewert der Größe in diesem Fall ohnehin als gering anzusehen. Des Weiteren erteilt der Chatbot zu keinem Zeitpunkt eine Auskunft über die Größe.
Tabelle 5.1 stellt einen Auszug aus der Relation dar.

5.1.2 Die Relation *Anwendung*

In der Relation *Anwendung* sind Anwendungsvorschläge für die einzelnen Produkte notiert. Ebenso wie in der Relation *Geschenk* muss es nicht für jedes Produkt eine Anwendung geben und dasselbe Produkt kann mehreren verschiedenen Anwendungen zuzuordnen sein.
Die Einträge in der Spalte ArtNr referenzieren die ArtNr der einzelnen Produkte in der Relation *Artikel*. Die

ArtNr	Anwendungsgebiet
001D01	Accessoire
002A04	Alltag
003A05	Alltag
004D00	EssenTrinken
004D00	Accessoire

Tabelle 5.2: Auszug aus der Relation *Anwendung*

ArtNr	Geschenkidee
001D01	Erinnerungsstück
002A04	Mitbringsel
003A05	Mitbringsel
004D00	Geschenk

Tabelle 5.3: Auszug aus der Relation *Geschenk*

Einträge in Anwendungsgebiet zeigen die Anwendungsmöglichkeiten. Derzeit gibt es fünf verschiedene Anwendungen: Accessoire, Alltag, EssenTrinken, Schreibwaren und Unterhaltung.
Tabelle 5.2 zeigt einen Ausschnitt aus *Anwendung*.

5.1.3 Die Relation *Geschenk*

In der Relation *Geschenk* ist notiert, als was für ein Geschenk ein Artikel dienen könnte. Ebenso wie in der Relation *Anwendung* muss es nicht für jedes Produkt eine Geschenkidee geben und dasselbe Produkt kann mehreren verschiedenen Arten von Geschenken zuzuordnen sein.
Die Einträge in der Spalte ArtNr referenzieren die ArtNr der einzelnen Produkte in der Relation *Artikel*. Die Einträge in Geschenkidee zeigen, wofür das Produkt dienen kann. Derzeit gibt es vier verschiedene Geschenkarten: Erinnerungsstück, Gastgeschenk, Geschenk und Mitbringsel.
Im Falle des Geschenks ist zu beachten, dass dem Begriff hier eine Ambiguität zukommt: zum einen als Oberbegriff und zum anderen als Geschenkart im Sinne eines Präsents zum Geburtstag oder zu Weihnachten. Um des Weiteren das Mitbringsel vom Geschenk unterscheiden zu können, wurde mit der internen Definition gearbeitet, dass ein Mitbringsel weniger als 6,00 Euro kostet.
Tabelle 5.3 zeigt einen Ausschnitt aus *Geschenk*.

5.1.4 Die Relation *PseudotabellePro*

Die Relation *PseudotabellePro* bildet die Grundlage für die späteren Relationen, die während des Dialogs speziell für den Nutzer erstellt werden.
Sie resultiert aus einem Natural Full Outer Join von *Artikel* mit *Anwendung* und *Geschenk*, weshalb sie im Zuge der Aktualität nach einer Änderung in einer oder mehreren der drei Relationen neu erstellt werden muss. Außerdem wird die Eigenschaft queried mit einer Integer als Wert (Default: 0) hinzugefügt

und die Einträge, die Null sind, werden durch den String „nichts“ ersetzt. queried und das Ersetzen der Null-Werte ist für die Produktsuche von Bedeutung (s. 5.3.1).
Da *PseudotanellePro* nur 60 Einträge umfasst, ist es in diesem Fall am einfachsten eine eben solche Relationen aus den anderen drei Relationen zu erstellen, anstatt erst während der Suche Joins zum Beispiel nur *Artikel* und *Anwendung* auszuführen.

5.1.5 Die Relation *mystringXXXX*

Während des Dialoges und der Suche wird für jeden Nutzer eine eigene Relation erstellt, die *mystringXXXX* heißt. Der Name setzt sich aus dem String „mystring“ und einer zufällig gewählten, vierstelligen Zahl zusammen. Diese Zahl wird in Dokumentation als „XXXX“ wiedergegeben, daher die Betitlung *mystringXXXX*, während die Relation in der Datenbank beispielsweise *mystring4083* heißen kann.
Eine eigene Relation für jeden Benutzer ist notwendig, da während der Produktsuche in der Relation die Einträge von queried individuell für jeden Nutzer verändert werden. Bevor der Chatbot in den Endzustand übergeht, wird *mystringXXXX* gelöscht.
Bei *mystringXXXX* handelt es sich um eine Kopie der Relation *PseudotabellePro*. Da die Datenbank Uni-Shop aufgrund der mehrwertigen Abhängigkeiten von Anwendungsgebiet und Geschenkidee in der vierten Normalform ist und *PseudotabellePro* alle drei grundlegenden Relationen in sich vereint, ist ArtNr als Schlüssel für die gesamte Relation nicht mehr ausreichend; der Schlüssel setzt sich nun aus ArtNr, Anwendungsgebiet und Geschenkidee zusammen. Da ein Schlüssel aus drei Eigenschaften hier die Suche unnötig verkomplizieren würde, wird noch die Eigenschaft row_number in *mystringXXXX* eingefügt, die jedes Objekt der Relation eindeutig referenziert.

5.2 Einbindung von PostgreSQL in ChatScript

Seit Version 4.2 erlaubt ChatScript das Einbinden von PostgreSQL-Datenbanken in den Chatbot (changes.txt in Wilcox (2019)). Dafür stellt ChatScript die Funktionen `^dbinit`, `^dbclose` und `^dbexecute` zur Verfügung. Im Folgenden werden die Funktionen kurz vorgestellt. Für weitere Informationen sei auf ChatScript-PostgreSQL in Wilcox (2019) verwiesen. Dort findet sich auch eine kurze Einführung in PostgreSQL.

5.2.1 Die Funktionen `^dbinit` und `^dbclose`

Mit `^dbinit` wird eine Verbindung zu einer bereits existierenden Datenbank aufgebaut. Ohne diese Funktion schlagen alle weiteren Datenbank-Operationen fehl. Wenn bereits eine Verbindung zur Datenbank besteht und `^dbinit` erneut aufgerufen wird, scheitert dies ebenfalls. Als ein Beispiel für `^dbinit` sei hier einer der Aufrufe aus Kaufabsicht wiedergegeben:

```
1 if (^dbinit(dbname = Uni-Shop port = 5432 user = postgres password =
      1234))
2                 {[Lass uns anfangen] [Super, auf geht's]! ^reuse( FIRSTQ )
3             } else {dbinit failed - $$db_error ^reuse( FIRSTQ ) }
```

Die Datenbank, die mit `^dbinit` initialisiert wurde, bleibt die Datenbank, mit der gearbeitet wird, bis die Verbindung mit `^dbclose` getrennt wird.

5.2.2 Die Funktion `^dbexecute` und Outputmacros

Um die SQL-Anweisungen aus ChatScript in PostgreSQL ausführen zu können, wird `^dbexecute` benötigt. Ein Aufruf könnte beispielsweise so aussehen:

```
1 if (^dbexecute(^"SELECT DISTINCT anwendungsgebiet FROM anwendung;" '^
     dbFirst )) {}
2 else {dbexecute failed - $$db_error ^reuse( FIRSTQ ) }
```

Dieses Beispiel entstammt ebenfalls Kaufabsicht. Der String („SELECT DISTINCT anwendungsgebiet FROM anwendung;“) enthält dabei die Queries, während sich `^dbFirst` auf das Outputmacro bezieht. Es ist dabei darauf zu achten, dass die einzelne Query mit einem Semikolon beendet wird und der Anweisungsblock insgesamt von doppelten Anführungszeichen, wie im Beispiel vorgegeben, eingerahmt wird. Durch die Verwendung der doppelten Anführungszeichen können die Werte von Variablen aus der ChatScript-Umgebung an PostreSQL übergeben werden. Allerdings müssen die Variablen instanziiert sein, da eine Null-Variable aus ChatScript in SQL als leerer String übersetzt wird. Zur Lösung dieses Problems in Augusta siehe 5.3.1.
Das Outputmacro gibt vor, wie die Ergebnisse der SQL-Query ausgegeben werden. Im Prinzip handelt es sich dabei um eine selbstdefinierte Funktion am Kopf der .top-Dateien, bei der die einzelnen Wörter des SQL-Ergebnisses die Argumente darstellen. Die Zahl der Argumente ist hierbei auf maximal 31 beschränkt. Das folgende Beispiel für das Outputmacro ist recht einfach:

```
1 outputmacro: ^dbFirst($_arg1)
2              $_arg1.
3              ^flushoutput()
```

`^dbFirst` ist für Suchergebnisse geschrieben, die pro Objekt nur aus einem Wort bestehen. Sollte es ein zweites Wort geben, wird dieses ignoriert. Hier wird lediglich nach jedem Wort ein Punkt gesetzt. Es sind aber auch kompliziertere Outputmacros möglich, die beispielsweise if-Anweisungen oder die zufällige Ausgabe von Wörtern und Sätzen (s. 3.8) beinhalten. Bei SQL-Anweisungen, die keine Ausgabe haben, wie beim Einfügen von Daten in eine Relation, ist die Angabe eines Outputmacros nicht erforderlich.

5.3 Der Gerbauch der Datenbank Uni-Shop in Augusta

Nach dem erfolgreichen Herstellen der Verbindung zu Uni-Shop in INTRO aus dem Topic Kaufabsicht ist die Datenbank einsatzbereit.
Vor dem ersten Durchsuchen der Datenbank nach einem passenden Produkt wird in SEARCHCREATE die Relation *mystringXXXX* erstellt. Da während der Suche die Werte für queried individuell verändert werden, ist eine eigene Relation für jeden Benutzer unerlässlich. *mystringXXXX* wird bei einem sauberen Programmende in ASKIFHAPPY gelöscht.
Zentraler Punkt der Suche ist die Eigenschaft queried. Sie funktioniert als Gedächtnis des Chatbots und

queried	Bedeutung
0	Default-Wert. Das Produkt wurde weder ausgewählt noch verworfen. Nur Produkte mit queried = 0 können noch gefunden werden.
1	Das Produkt wurde während der aktuellen Suche gefunden und dem Kunden vorgeschlagen. Es wurde noch nicht angenommen oder abgelehnt.
2	Das Produkt wurde bei einer Suche gefunden, dem Kunden vorgeschlagen und abgelehnt. Bei weiteren SQL-Queries wird es ignoriert
3	Das Produkt wurde bei der aktuellen Suche gefunden und vom Kunden angenommen.
4	Das Produkt wurde in einer vergangenen Suche angenommen und wird dem Kunden beim Warenkorb angegeben.
5	Das Produkt wurde in einer vergangenen Suche angenommen, wird dem Kunden aber nicht im Warenkorb angegeben. Bei weiteren SQL-Queries wird es ignoriert.

Tabelle 5.4: Bedeutung der Werte für queried

stellt die Grundlage für die Ausgabe der Produktinformationen an den Benutzer und den Warenkorb dar. In der vorliegenden Version, wird die Datenbank erst zur Produktsuche im Topic Dbsearch verwendet, es ist allerdings auch möglich, die Anwendungsgebiete und Geschenkideen in FIRSTQ automatisch aus der Datenbank heraussuchen zu lassen. Näheres dazu siehe in 5.3.4.

5.3.1 Die Eigenschaft queried

Obwohl ChatScript über die Möglichkeit verfügt, den geführten Dialog für jeden Nutzer einzeln zu speichern, erschien es uns einfacher und sinnvoller, die Informationen, welche Produkte schon gefunden, vom Kunden akzeptiert oder abgelehnt wurden, in der Datenbank selbst zu speichern. Dies geschieht in der Eigenschaft queried der Relation *mystringXXXX* (für *mystringXXXX* s. 5.1.5).
queried ist vom Typ Integer und kann einen Wert von 0 bis einschließlich 5 betragen. Zu Beginn, wenn noch keine Suche ausgeführt wurde, beträgt queried also bei jedem Objekt 0. Das Produkt, das den Suchkriterien entspricht und dem Kunden vorgeschlagen wird, wird mit 1 markiert. Da es dasselbe Produkt aufgrund der mehrwertigen Abhängigkeit mehrere Einträge in der Relation haben kann, ist es möglich, dass während eines Suchvorgangs mehrmals die 1 vorgeben wird. Mit einer 2 gekennzeichnete Artikel wurden vom Kunden abgelehnt. In Folge dessen werden diese auch in folgenden Suchen mit möglicherweise veränderten Kriterien dem Benutzer nicht mehr angezeigt.
Der Artikel, den der Kunde bei der aktuellen Suche ausgewählt hat, wird mit 3 kenntlich gemacht. In einem weiteren Schritt werden aus den Produkten mit 3 entweder eine 4 oder 5. Der Grund dafür liegt erneut darin, dass in *mystringXXXX* ein Produkt mehrere Einträge haben kann. Dem Kunden werden nur der Name, die Beschreibung und der Preis angezeigt. Da sich die Einträge aber in Anwendungsgebiet oder Geschenkidee unterscheiden, ist ein Unterschied für den Kunden nicht erkennbar und er wundert sich, weshalb ihm dieselbe Information mehrmals präsentiert wird oder hält es für einen Fehler. Um dies zu vermeiden, wird einer der mit 3 markierten Einträge mit 4 gekennzeichnet und die anderen mit 5. Nur der Eintrag des Produkts mit einer 4 wird dem Kunden bei Bedarf, zum Beispiel beim Warenkorb gekennzeichnet.
Tabelle 5.4 bietet eine Übersicht der Bedeutung der queried-Werte.

queried bildet die Grundlage der Suche. Es wird einmal nach einem Produkt gesucht und dieses daraufhin mit 1 gekennzeichnet. Alle darauffolgenden Operationen, die auf dieser Suche basieren, zum Beispiel Ausgabe des Namens und der Beschreibung, Markierung mit den anderen Zahlen und Ausgabe des Warenkorbs, erfolgt anschließend auf der Basis des queried-Werts.

5.3.2 Die Query zur Produktsuche

Die eigentliche SQL-Query zur Suche hat noch keine Ausgabe, sondern verändert nur den Wert für queried auf 1. Die Query besteht aus zwei Teilen: Zuerst wird **ein** Artikel gesucht, dessen Eigenschaften den Suchkriterien entsprechen, und in einem zweiten Schritt werden dann alle Einträge in *mystringXXXX*, die im Namen mit dem gefundenen Artikel übereinstimmen und daher das gleiche Produkt beschreiben, mit 1 markiert. Der folgende Programmcode zeigt die SQL-Abfrage:

```
1  UPDATE $tablename
2  SET queried = 1
3  WHERE name =
4    (SELECT name
5    FROM $tablename
6    WHERE
7      QUERIED = 0
8    AND
9      name = (CASE WHEN $things LIKE '%nichts%' THEN name ELSE $things END)
10   AND
11     art  = (CASE WHEN $art LIKE '%nichts%' THEN art ELSE $art END)
12   AND
13     preis <= (CASE WHEN $preis < 0 THEN preis ELSE $preis END)
14   AND
15     ausfuehrung = (CASE WHEN $ausfuehrung LIKE '%nichts%' THEN ausfuehrung
           ELSE $ausfuehrung END)
16   AND
17     geschenkidee = (CASE WHEN $geschenkidee LIKE '%nichts%' THEN
          geschenkidee ELSE $geschenkidee END)
18   AND
19     anwendungsgebiet = (CASE WHEN $anwendungszweck LIKE '%nichts%' THEN
          anwendungsgebiet ELSE $anwendungszweck END)
20   LIMIT 1);
```

In den Zeilen 4 bis 20 findet sich der erste Teil der Suche, in dem nach dem passenden Produkt gesucht wird, wobei nur Produkte berücksichtigt werden, die dem Kunden vorher noch nicht gezeigt worden sind (queried = 0).
Das Einbinden von Variablen aus der ChatScript-Umgebung in die SQL-Abfrage ist nicht unproblematisch, da Variablen mit NULL-Werten in SQL nicht als NULL sondern als leerer String interpretiert werden, was in der Folge zu einem Fehlschlagen der Abfrage führt. Um dennoch nicht für jeden der 68 möglichen Fälle eine eigene Abfrage schreiben zu müssen, haben wir eine Lösung gefunden, die dieses Problem umgeht.
Die Variablen für die extrahierten Stichwörter (`$anwendungszweck`, `$geschenkidee`, `$things`, `$art`, `$preis` und `$ausfuehrung`) werden, wenn ihnen durch den Nutzer keine Kriterien zugewiesen werden, mit dem String „nichts“ bzw. bei `$preis` mit -1 initialisiert. In der Relation *mystringXXXX* wurden

die Null-Werte ebenfalls durch den String „nichts“ ersetzt.
Um in PostgreSQL explizit anzugeben, dass eine Eigenschaft unwichtig ist, kann dies so formuliert werden:

```
1 SELECT *
2 FROM table
3 WHERE property = property
```

Hier kommen für die Eigenschaft property also alle Werte in Frage, die es in der Relation für property gibt und damit ist `property = property` zu jedem Zeitpunkt wahr und bildet kein einschränkendes Kriterium zur Suche.
Die Abfrage in Augusta funktioniert ähnlich. Beim Betrachten von beispielsweise Z. 11 aus der SQL-Query oben fällt auf, dass das hier auch eine CASE-Anweisung enthalten ist.

```
1 art  = (CASE WHEN $art LIKE '%nichts%' THEN art ELSE $art END)
```

Es gibt also zwei Möglichkeiten, wie die Zeile die Suche beeinflussen kann: Wenn `$art` den String „nichts“ beinhaltet, was nur der Fall ist, wenn der Kunde keine Angaben zur Art des Produkts gemacht hat, lautet das Suchkriterium `art = art`. Hier würde die Produktauswahl nicht weiter eingeschränkt werden.
Ist nun aber `$art` ein anderer String als „nichts“ zugewiesen, dann lautet das Suchkriterium `art = $art`, sodass die Art des Artikels bei der Suche berücksichtigt wird. Analog werden die anderen Variablen gehandhabt.
Es wäre schön gewesen, wenn wir bei Name und Art ein LIKE hätten verwenden können, sodass bei der Eingabe von zum Beispiel „Tasse“ für Name die Kaffeetasse gefunden worden wäre. Leider ist das Verwenden von LIKE bei unserer Lösung nicht möglich, sodass die Werte in den Variablen exakt mit den Einträgen in der Tabelle übereinstimmen müssen.
Nach dem die Produkte gefunden wurden, wird das Ergebnis auf ein Produkt reduziert.
Der zweite Teil der Suche findet sich in Zeile 1 bis 3. queried wird in *mystringXXXX* dort auf 1 gesetzt, wo der Name des Eintrags mit dem des im ersten Teil gefundenen Produkts übereinstimmt. Wie bereits erwähnt, gibt es in der Relation aufgrund der mehrwertigen Abhängigkeit von Anwendungsgebiet und Geschenkidee für einige Produkte mehrere Einträge. Auch wird bei den Suchergebnissen nicht zwischen der Ausführung unterschieden, da dem Kunden ohnehin nur Name, Beschreibung und Preis angezeigt werden.

5.3.3 Die Ausgabe der Produkteigenschaften und der Warenkorb

In den Regeln PRODUCTNAME, PRODUCTDESCR und PRODUCTPRICE wird jeweils einzeln der Name, die Beschreibung bzw. der Preis des gefunden Produkts ausgegeben. Die SQL-Query für den Namen sei hier als Beispiel gegeben; die Abfragen für Beschreibung und Preis sind analog aufgebaut:

```
1 SELECT name FROM $tablename WHERE queried = 1 LIMIT 1;
```

Das Outputmacro `^dbsec` ist am Anfang von *dbsearch.top* definiert. Es sieht eine unterschiedliche Ausgabe für die Eigenschaften vor.
Zudem wird in `^dbsec` noch die Variable `$ergebnis` auf „positiv“ gesetzt. `$ergebnis` wird in PRODUCTNAME instanziiert und dient dazu, überprüfen zu können, ob ein Schritt zuvor überhaupt ein Produkt gefunden wurde. Da in der Suche nur die Einträge in der Relation verändert werden, aber

standardmäßig keine SQL-Ausgabe erfolgt, kann in ChatScript an dieser Stelle noch nicht festgestellt werden, ob ein Artikel gefunden wurde. Das passiert erst in PRODUCTNAME, wenn nach Einträgen mit queried = 1 gesucht wird. Sind solche Einträge in der Relation vorhanden, wird entsprechend einer davon ausgewählt, `^dbsec` wird aufgerufen und `$ergebnis` wird „positiv“.
Wurde nun allerdings bei der vorangeganenen Suche kein passender Artikel gefunden, so wird auch kein Eintrag mit queried = 1 gefunden. Dementsprechend wird weder das Outputmacro `^dbsec` aufgerufen noch `$ergebnis` verändert, woraufhin eine entsprechende Meldung an den Kunden weitergegeben und der Kunde zu FIRSTQ in dem Topic Kaufabsicht weitergeleitet wird.
Es gibt außerdem die Möglichkeit für den Kunden, dass er sich ansehen kann, welche Artikel er bereits akzeptiert hat. Diese Option, in Anlehnung an die reale Welt und Internetshops als „Warenkorb“ bezeichnet, findet sich in der Regel REPONSE in Dbsearch.
Akzeptiert der Benutzer in RESPONSE das ihm angebotene Produkt, wird queried erst auf 3 und dort anschließend bei einem Eintrag auf 4 und bei den anderen 5 gesetzt.
Beim Warenkorb werden zunächst die Artikel gezählt, die queried = 4 sind, um dem Kunden darüber informieren zu können, wie viele Produkte er bereits ausgewählt hat. In einem zweiten Schritt wird für diese Artikel dann Name und Preis ausgegeben. Da davon auszugehen ist, dass sich der Kunde noch erinnern kann, was er ausgewählt hat, wenn er den Namen des Produkts liest, ist die Ausgabe der Beschreibung hier nicht nötig. Die Ausgabe erfolgt über das Outputmacro `^dbThird`, das ebenfalls in *dbsearch.top* definiert ist.

5.3.4 Alternative Ausgabe von Anwendungsgebiet und Geschenkidee in FIRSTQ

In FIRSTQ werden die Werte, die für Anwendungsgebiet und Geschenkidee möglich sind, direkt in den Code geschrieben. Es gibt allerdings auch die Möglichkeit, diese automatisch aus der Datenbank extrahieren zu lassen und anschließend dem Kunden zu präsentieren.
Da es nur 4 verschiedene Werte für Geschenkidee und 5 für Anwendungsgebiet gibt, haben wir uns der Einfachheit halber dagegen entschieden.
Wenn es in der Datenbank nun aber mehr Werte dafür gibt oder sich diese häufig ändern, kann es aus diesen oder anderen Gründen gewünscht sein, eine automatische Ausgabe durch den Chatbot zu generieren. Die Grundlage dafür ist FIRSTQ durchaus vorhanden.
Für die automatische Ausgabe müssen in der Regel FIRSTQ aus Kaufabsicht die Kommentarzeichen (#) vor den Datenbankabfragen

```
1 if (^dbexecute(^"SELECT DISTINCT geschenkidee FROM geschenk;" '^dbFirst ))
      {}
2 else {dbexecute failed list from table- $$db_error ^reuse( FIRSTQ ) }
```

und

```
1 if (^dbexecute(^"SELECT DISTINCT anwendungsgebiet FROM anwendung;" '^
    dbFirst )) {}
2 else {dbexecute failed list from table- $$db_error ^reuse( FIRSTQ ) }
```

sowie die Werte für Anwendungsgebiet und Geschenkidee direkt im Code entfernt werden. Dabei gilt es zu beachten, dass es die Datenbankabfragen jeweils zweimal in FIRSTQ gibt. Anschließend müssen auch die Kommentarzeichen um das Outputmacro `^dbFirst` am Kopf der *kaufabsicht.top*-Datei gelöscht werden.
Damit die Änderungen übernommen werden, muss vor einem neuen Gebrauch der Bot neu kompiliert werden.

6 Benutzerschnittstelle des Chatbots

Da eine Konsole nicht als benutzerfreundlich einzustufen ist, brauchten wir noch eine Benutzerschnittstelle. ChatScript bietet hier auch mehrere Möglichkeiten an, unter anderem gibt es eine serverbasierte Version, die von vorneherein mitgeliefert wird (s. ChatScript - ClientServer - Manual in Wilcox (2019)). Allerdings sind wir beim Einrichten auf mehrere Probleme gestoßen, auf die in 7 näher eingegangen wird. Aufgrund dieser Probleme war es leider nicht möglich, die Benutzerschnittstelle mit dem Chatbot im Ganzen zu verwenden, weshalb im Folgenden nur kurz und der Vollständigkeit halber auf die GUI eingegangen werden soll.
Die vorgegebene serverbasierte Version war vorteilhaft, da sie zum einen die potentielle spätere Einbindung in den Uni-Shop auf der Webseite der Universtiät Trier ermöglicht und zum anderen kaum Aufwand unsererseits für die Konstruktion einer eigenen GUI nötig war. Lediglich Stylesheet-Befehle wurden eingefügt, um das Design unseren Bedürfnissen anzupassen.
Abbildung 6.1 zeigt einen Screenshot der Benutzerschnittstelle. Die Darstellung betreffend haben wir den Stil der Webseite der Universität nachgebildet, um die Zugehörigkeit zu betonen. So befindet sich in der linken oberen Ecke das Universitätslogo und rechts daneben auf dunkelblauen Grund in weißer Schrift die Überschrift des Chatbots. Die Kommentare Augustas und die des Kunden erscheinen in Sprechblasen. Augustas sind blassblau, da blau die dominantere Farbe im Farbschema der Universität ist. Das Gesagte des Kunden ist blassgrün, da auch grün mit der Institution assoziiert wird.

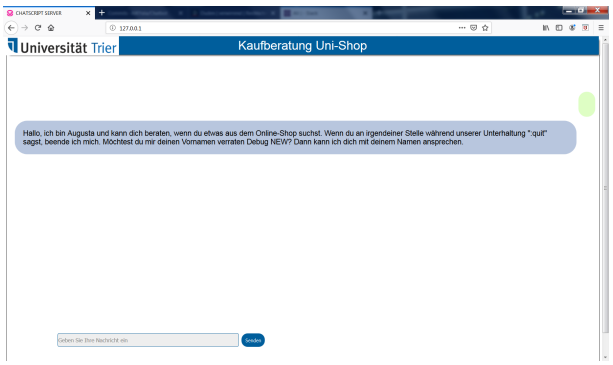


Abbildung 6.1: Screenshot der Benutzerschnittstelle

7 Probleme

Dieses Kapitel behandelt Schwierigkeiten, die bei der Arbeit mit ChatScript aufgetaucht sind und haben für uns ein ernstes und großes Hindernis dargestellt. Im Falle des zweiten Problems haben wir auch keine Lösung gefunden.

7.1 Zwei-Query-Problem

Ein Problem, für das ein Workaround gefunden werden konnte, war das 'Zwei-Query-Problem'. Dieses Problem ist in den allen Versionen vorhanden und tritt wegen einem der folgenden Fehler auf:

1. Suche auf einer vor der Nutzung existierenden Datenbank
2. Fehlschlag in Queries

Ersteres tritt auf, wenn man mit Hilfe des Befehls `:build onesentence` oder Ähnliches den Bot, hier onesentence, neu kompiliert. Dies hat den Ursprung darin, dass der Bezeichner für die Datenbanken per Client mit einer Zufallszahl konkateniert wird und jeder `:build`-Befehl dieselbe Zufallszahl generiert. Wird eine Tabelle nach ihrer Nutzung nicht gelöscht, dann tritt dieser Fehlschlag auf.
Dieses Problem erscheint in dieser Form nicht außerhalb des Testens, da es unwahrscheinlich ist, dass zwei Tabellen mit demselben Bezeichner zufallsgeneriert werden. Die Komplikation entsteht nicht, wenn ein Bot regelmäßig, das heißt durch Verabschiedung in Ende, beendet wird. Man kann dieses Problem manuell durch explizite `drop`-Befehle außerhalb von Chatscript behandeln.
Zweiteres tratt auf, wenn mit den angegebenen Eigenschaften kein Produkt gefunden werden konnte. Wenn kein Produkt gefunden werden kann, ist dem Skript zu Folge in den Zustand ~ende.SETNULLANDREP überzugehen, um eine neue Suche zu gewährleisten. Nach der Programmlogik müssten hier die einzelnen Regeln bis zu der Datenbanksuche wieder iteriert werden, um Nutzerangaben zu sammeln. Stattdessen versuchte der Bot nach dem Zurücksetzen der Variablen, erneut eine Suche mit NULL-Variablen auszuführen, indem er eigenständig in SUMMARY sprang und ohne eine Antwort abzuwarten eine Suche startete.
Dieses Problem wurde größtenteils umgangen, indem man den Nutzer mehrmals hintereinander fragt, ob man eine neue Suche ausführen will. Der Ursprung dieses Problems bleibt jedoch ungeklärt. Eine Vermutung ist, dass ChatScript intern Regeln, die bereits besucht worden sind, markiert, um nicht erneut von alleine in jene Regeln überzugehen und so Wiederholungen im Gesprächsablauf zu vermeiden (Wilcox 2019).

7.2 Probleme mit der GUI

Bei dem Testen der GUI fallen einige Probleme auf:

1. Fehlender Support von PostgreSQL-Server für Mac und Linux
2. Server mit PostgreSQL-Unterstüzung vs. Server ohne PostgreSQL-Unterstüzung

Das Testen der GUI erfolgte durch das Hosten auf Localhost (127.0.0.1) der durch ChatScript bereitgestellten Dateien im Ordner *SERVER BATCH FILES* (Wilcox 2019). Während das Aufsetzen eines Servers auf Windows recht einfach war, fiel schnell auf, dass einige unvorhergesehene Probleme auftraten.

7.2.1 Fehlender Support von PostgreSQL-Server für Mac und Linux

In der Dokumentation für ChatScript wird beschrieben, wie ein Server ohne Postgres-Unterstützung auf Windows, Mac und Linux-Distributionen aufzusetzen ist. Für Windows wird die .bat-Datei *LocalPgServer.bat* mitgeliefert, sodass das Aufsetzen eines Servers auf Windows keine Probleme bereitet. Im Gegensatz dazu wird nichts über Server mit Postgres-Support für Linux oder Mac in der Dokumentation erwähnt. Nach mehreren erfolglosen Versuchen wurde im Forum für Chatbots (`https://www.chatbots.org/ai_zone/viewthread/3681/`, Stand: 08.07.19) LINK richtig??? nachgefragt, wobei diese Frage unbeantwortet blieb.

7.2.2 Server mit PostgreSQL-Unterstüzung vs. Server ohne PostgreSQL-Unterstüzung

Ein weiteres Problem bei dem Testen der GUI war, dass der Bot verschiedene Verhaltensweisen je nach .bat-Datei aufweist. So entspricht der Programmablauf des Bots dem der Konsolenausgabe, wenn ein Server ohne Postgres-Support, zum Beispiel durch *LocalServer.bat*, eingerichtet wird. Hier verhält sich der Bot in den Zustandsübergängen identisch zu der Konsolenausgabe, mit dem Unterschied, dass der Nutzer bei der GUI / des Webinterfaces die erste Eingabe machen muss.
Das bedeutet, dass bis zum Zeitpunkt der Datenbankabfrage in Dbsearch keine Probleme auftreten. Zu Schwierigkeiten kommt es in diesem Fall verständlicherweise erst dann, wenn eine Datenbankabfrage ausgeführt werden muss, da hier keine Unterstützung für PostgreSQL besteht.
Im Gegensatz dazu gab es in vielerlei Hinsicht Probleme mit der Einrichtung eines Servers mit Postgres-Unterstüzung. Bei einigen Anläufen startete der Bot gar nicht, im besten Fall wird ~dbsearch.RESPONSE erreicht, jedoch geht der Bot von dort aus in keine weitere Regel, sondern iteriert weitergehend auf ~dbsearch.RESPONSE.
In vielen Versuchen erwiesen sich Übergänge in Regeln mit `^reuse ()` als problematisch. Bei Aufsetzen des Servers mit *LocalPgServer.bat*, ist es vorgekommen, dass einige der vorgesehenen Regelübergänge per `^reuse()` nicht aktiviert wurden.
Da Probleme mit *LocalPgServer.bat* nicht in der Dokumentation von ChatScript behandelt werden und der Macher und Entwickler von ChatScript, Bruce Wilcox, nicht auf Nachfragen mit dem Hosten eines Servers mit Postgres-Unterstützung antwortet (`https://www.chatbots.org/ai_zone/viewthread/3681/`, Stand: 08.07.19) LINK richtig??, konnten alle Probleme mit dem Server mit Postgres-Unterstützung zeitnah nicht gelöst werden.
Es besteht unsererseits die Vermutung, dass es bei *LocalPgServer.bat* Schwierigkeiten mit der User-Datenbank gibt, die ChatScript automatisch erstellt. In der Datenbank *user* werden die einzelnen User mit Name und deren Protokolle abgespeichert. Diese Protokolle sind auch über den Ordner *USER*

zugänglich und tragen den Namen *topic_username_botname.txt*. Interessanterweise wird bei *LocalPgServer.bat* zwar ein Protokoll für den Benutzer angelegt, das existiert aber nur in der Datenbank und erscheint in dem USERS-Ordner nicht. Zudem bricht es nach SUMMARY ab, als wäre der Bot noch immer in diesem Zustand, obwohl die Datenbanksuche schon ausgeführt wurde und er aufgrund des ausgegebenen Textes schon in der Regel RESPONSE sein müsste.
Wie bereits erwähnt, tritt dieses Problem bei der Version über die Konsole und *LocalSever.bat* nicht auf. Hier die Protokollführung nicht ab und die entsprechenden Protokolle sind über USERS einsehbar.
In der Datei *bugs.txt* aus dem Ordner *LOGS* wird das Problem folgendermaßen beschrieben: `Postgres filessys write failed for USERS/topic\_127001\_augusta.txt`. Eine Suche über Google ergab, dass diese Beschreibung schon bei Schwierigkeiten mit ChatScript und dem Amazon Server aufgetaucht war (`https://www.chatbots.org/ai_zone/viewthread/3664/`). Die dortige Lösung das Problems, das Unique-Constraint der Relation *userfiles* zu entfernen, haben wir auf unseren Fall übertragen und ausprobiert, allerdings hat dies nicht den gewünschten Effekt gehabt.
Aufgrund Zeitmangels, mangelnder Dokumentation und fehlender Einsicht in die Binärdateien bzw. Konfiguration ist dieses Problem bis dato nicht gelöst. Die genaue Fehlerquelle konnte nicht ausfindig gemacht werden und es ist ein Rätsel, warum auch Nicht-Postgres-Abläufe sich bei dem Server mit Postgres-Unterstützung anders verhalten als ohne Postgres-Unterstützung.

Beispiele zum Dialogablauf

Im Folgenden gibt es einige Beispiele für den Ablauf des Dialogs zwischen Chabot und Kunde, die wir vor und teilweise auch während des Programmieren geschrieben haben. So enthält beispielsweise Dialog 3 gegenüber Dialog 1 bereits Änderungen, die wir mit in das Programm übernommen haben.
Die kursiv geschriebenen Ausgaben des Chatbots (z. B. *ABC* oder *Beschreibung*) werden in der Realität durch Name, Beschreibung und Ähnliches ersetzt.

Dialog 1: Abgelehntes Produkt

Dieser Dialog war der erste, der für den Chatbot geschrieben wurde. Er stellt eine Situation dar, in der ein Kunde das ihm vorgeschlagene Produkt ablehnt, wonach der Chatbot erneut suchen muss.
In der ersten Vorstellung hatten wir noch geplant, dass der Kunde bei Interesse nach Produktinformationen fragen kann. Im Verlauf des Projekts haben wir uns gegen das Nachfragen entschieden und präsentieren dem Kunden den Namen, die Beschreibung und den Preis sofort (s. 5.3.3). Zum einen ist diese Variante einfacher und zum anderen sind Beschreibung und Name zusammen aussagekräftiger als der Name alleine. Außerdem fiel beim Schreiben des Dialogs auf, dass Fragen mit oder zu vermeiden sind, da manche Personen dazu neigen, solche Fragen mit „Ja.“ und „Nein.“ zu beantworten.

CHATBOT:	Hallo. Suchen Sie ein Geschenk oder ein Produkt für einen bestimmten Zweck?
KUNDE:	Ich suche ein Geschenk.
CHATBOT:	Nach was für einer Art (von Geschenk) suchen Sie? Ich kann Ihnen folgende Optionen anbieten: Geschenk, Gastgeschenk, Mitbringsel, Erinnerungsstück.
KUNDE:	Gastgeschenk.
CHATBOT:	Ich kann Ihnen dieses Produkt empfehlen: *ABC*. Möchten Sie weitere Informationen darüber oder soll ich weitersuchen?
KUNDE:	Weitere Informationen.
CHATBOT:	*Beschreibung*. Ist es das, was Sie gesucht haben?
KUNDE:	Nein.
CHATBOT:	Ich werde weitersuchen. Das kann ich noch empfehlen: *DEF*. Möchten Sie weitere Informationen darüber oder soll ich weitersuchen?
KUNDE:	Ich will mehr Infos.
CHATBOT:	*Beschreibung*. War es das, was Sie gesucht haben?
KUNDE:	Wie viel kostet das?
CHATBOT:	*DEF* kostet *9,50* Euro. Möchten Sie *DEF* erwerben?
KUNDE:	Ja.
CHATBOT:	Sie können das Ticket für *DEF* über den Tukan erwerben und dieses dann in der Pressestelle oder im Shop abgeben. Suchen Sie ein weiteres Produkt? Ein Geschenk oder einen Artikel für einen bestimmten Zweck?
KUNDE:	Nein. Tschüss.
CHATBOT:	Auf Wiedersehen.

Dialog 2: Zweite Suche

In diesem Dialog wird nach einem ersten Treffer noch nach einem zweiten Produkt mit verschiedenen Kriterien gesucht. Nach der Bestätigung, dass nach einem neuen Produkt gesucht werden soll, beginnt der Dialog ab der Frage nach einem Geschenk oder Anwendungszweck von vorne.
Da wir die Kategorien der Geschenkideen und Anwendungsgebiete automatisch ausgeben wollten, hat sich hier die Ausgabe ein wenig verändert. Die Oder-Frage des vergangenen Dialogs haben wir herausgenommen. Dazu wird die Information über der Uni-Shop erst dann ausgegeben, wenn der Kunde nicht mehr möchte, damit auch bei mehreren Suchen dieser Satz nur einmal erwähnt wird.

CHATBOT:	Hallo. Suchen Sie ein Geschenk oder ein Produkt für einen bestimmten Zweck?
KUNDE:	Ich suche ein Geschenk.
CHATBOT:	Nach was für einer Art Geschenk suchen Sie? Ich kann Ihnen folgende Optionen anbieten: Geschenk. Gastgeschenk. Mitbringsel. Erinnerungsstück.
KUNDE:	Ich will ein Gastgeschenk.
CHATBOT:	Ich kann Ihnen dieses Produkt empfehlen: *ABC*. Möchten Sie weitere Informationen dazu? Wenn Sie möchten, kann ich weitersuchen.
KUNDE:	Informationen.
CHATBOT:	*Beschreibung*. Ist es das, was Sie gesucht haben?
KUNDE:	Ja.
CHATBOT:	Möchten Sie ein weiteres Produkt suchen?
KUNDE:	Ja.
CHATBOT:	*Beschreibung.* Suchen Sie ein Geschenk oder ein Produkt für einen bestimmten Zweck?
KUNDE:	Bestimmter Zweck.
CHATBOT:	Wofür benötigen Sie das Produkt: Ich kann Ihnen folgende Kategorien anbieten: Accessoire. Alltag. EssenTrinken. Deko. Uni.
KUNDE:	Alltag.
CHATBOT:	Ich kann Ihnen dieses Produkt empfehlen: *DEF*. Möchten Sie weitere Informationen dazu? Wenn Sie möchten, kann ich weitersuchen.
KUNDE:	Wie teuer ist das?
CHATBOT:	*DEF* kostet *9,50* Euro. War es das, was Sie gesucht haben?
KUNDE:	Ja.
CHATBOT:	Möchten Sie ein weiteres Produkt suchen?
KUNDE:	Nein. Tschüss.
CHATBOT:	Sie können das Ticket für die Produkte über den Tukan erwerben und dieses dann in der Pressestelle oder im Shop abgeben. Auf Wiedersehen.

Dialog 3: Suche mit genaueren Kriterien und Warenkorb

Dieser Dialog ist etwas später entstanden. Hier gibt es bereits die Möglichkeit, zusätzlich zu Geschenk und Anwendung noch weitere Kriterien wie die Ausführung oder den Preis anzugeben. In den Onesentence-Chatbots ist dies in der Variante implementiert, wie im Dialog beschrieben: Der Kunde gibt die Kriterien in einem Satz an und der Chatbot weist Variablen zu. In KeyExProdukteigenschaften wird einzeln nach Name, Art, Ausführung und Preis gefragt. Dies hatten wir zu Beginn geschrieben, um zumindest einen Dialog führen und die Variablen zuzuweisen zu können und dann es später die schwierigere Variante mit einem einzigen Satz auszutauschen. Allerdings stellte sich im Anwendungstest

(s. Ergebnis des Usability Testing) heraus, dass vom Benutzer der Dialog mit den einzelnen Abfragen als einfacher empfunden wird.
Dazu wird beim Produktvorschlag gleich die Beschreibung mit ausgegeben und der Kunde erhält die Möglichkeit, sich anzusehen, was er bereits ausgewählt hat.
Zudem duzt der Chatbot den Kunden und fragt nach einem Namen, um persönlicher sein zu können.

CHATBOT:	Hallo. Möchtest du mir deinen Namen verraten? Das ist etwas persönlicher.
KUNDE:	XYZ.
CHATBOT:	Hallo, *XYZ*. Du bist hier, um etwas zu kaufen, oder?
XYZ:	Ja.
CHATBOT:	Dann fangen wir an. Suchst du ein Geschenk oder ein Produkt für einen bestimmten Zweck?
XYZ:	Geschenk.
CHATBOT:	Nach was für einer Art Geschenk suchst du? Ich kann dir folgende Optionen anbieten: Geschenk, also zum Beispiel zum Geburtstag oder zu Weihnachten. Gastgeschenk. Mitbringsel. Erinnerungsstück.
XYZ:	Ein Gastgeschenk.
CHATBOT:	Hier kannst du in einem Satz noch weitere Kriterien für das gesuchte Produkt hinzufügen. Den Namen oder die Art des Produkts, zum Beispiel. Zudem bieten wir manche Produkte in verschiedenen Farben oder Sprachen an. Außerdem kannst du eine preisliche Obergrenze festlegen.
XYZ:	Ein Buch in Englisch für 10 Euro.
CHATBOT:	Ich wiederhole: Du möchtest ein Gastgeschenk. Es handelt sich dabei um ein Buch. Es ist in Englisch und kostet maximal 10 Euro. Stimmt das so?
XYZ:	Ja.
CHATBOT:	Ich kann dir dieses Produkt empfehlen: *ABC*. *Beschreibung.* Ist es das, was du gesucht hast? Wenn du möchtest, kann ich weitersuchen.
XYZ:	Ja.
CHATBOT:	Möchtest du ein weiteres Produkt suchen? Du kannst dir auch ansehen, was du bereits ausgewählt hast.
XYZ:	Ich will mir ansehen, was ich bereits ausgewählt habe.
CHATBOT:	Du hast *1* Produkt ausgewählt. Es ist *ABC*. Die Beschreibung lautet *Beschreibung.* Es kostet *6,50* Euro. Möchtest du ein weiteres Produkt suchen?
XYZ:	Nein.
CHATBOT:	Du kannst das Ticket für die Produkte über den Tukan erwerben und dieses dann in der Pressestelle oder im Shop abgeben. Auf Wiedersehen.

Dialog 4: Kein Ergebnis / Fehler bei der Suche

Dieser Dialog stellt die Möglichkeit, dass es bei der Suche zu einem Fehler kommt. Wenn der Chatbot kein Produkt findet, wird es statt „Oh, tut mir leid, da habe ich mich verrechnet. Lass uns von vorne anfangen.“ „Ich habe leider kein Produkt gefunden, das auf deine Kriterien passt. Versuchen wir es nochmal mit anderen Kriterien“ ausgeben.
Dazu kann der Kunde hier den Bot auch mitten im Dialog beenden, was im Chatbot OnesentenceGoodbye implementiert ist.

CHATBOT:	Hallo. Du kannst jederzeit „Tschüss“ sagen, wenn du mich beenden Möchtest du mir deinen Namen verraten? Das ist etwas persönlicher.
KUNDE:	Nein.
CHATBOT:	Das kann ich verstehen. Du bist hier, um etwas zu kaufen, oder?
KUNDE:	Ja.
CHATBOT:	Dann fangen wir an. Suchst du ein Geschenk oder ein Produkt für einen bestimmten Zweck?
KUNDE:	Geschenk.
CHATBOT:	Nach was für einer Art Geschenk suchst du? Ich kann dir folgende Optionen anbieten: Geschenk, also zum Beispiel zum Geburtstag oder zu Weihnachten. Gastgeschenk. Mitbringsel. Erinnerungsstück.
KUNDE:	Gastgeschenk.
CHATBOT:	Hier kannst du in einem Satz noch weitere Kriterien für das gesuchte Produkt hinzufügen. Den Namen oder die Art des Produkts, zum Beispiel. Zudem bieten wir manche Produkte in verschiedenen Farben oder Sprachen an. Außerdem kannst du eine preisliche Obergrenze festlegen.
KUNDE:	Englisch Buch 10.
CHATBOT:	Ich wiederhole: Du möchtest ein Gastgeschenk. Es handelt sich dabei um ein Buch. Es ist in Englisch und kostet maximal 10 Euro. Stimmt das so?
KUNDE:	Ja.
CHATBOT:	Oh, tut mir leid, da habe ich mich verrechnet. Lass uns von vorne anfangen. Suchst du ein Geschenk oder ein Produkt für einen bestimmten Zweck?
KUNDE:	Tschüss.
CHATBOT:	Du kannst das Ticket für die Produkte über den Tukan erwerben und dieses dann in der Pressestelle oder im Shop abgeben. Auf Wiedersehen.

Ergebnis des Usability Testing

Anfang Juni haben wir unser Programm einer kleinen Evaluierung unterzogen. Die Testgruppe bestand aus zwei Personen, die einige Male mit Augusta gesprochen haben. Im Folgenden findet sich die Zusammenfassung dieses kleinen Usability Testings.
Ein Problem dabei war allerdings, dass das Programm zu diesem Zeitpunkt noch einige Bugs aufwies und nicht voll funktionsfähig war. Doch ein kurzes Feedback war möglich. Getestet wurde zudem nur eine Version auf der Konsole.

Zu Beginn gibt es zwei Mal in anderer Formulierung die Frage, ob der Kunde hier sei, um etwas zu kaufen (aus INTRO). Diese Fragen, vor allem in doppelter Ausführung, wurden nicht nur als unnötig sondern auch als nervig empfunden. Ebenso wird die Frage angezeigt, wenn der Kunde nach einem Fehler bei der Suche neu anfangen muss. Auch dies war für die Benutzer störend.
Verwirrend erschien des Weiteren die Angabe der Kategorien aus Geschenkidee (FIRSTQ). So sei bei „Geschenk, also zum Beispiel zum Geburtstag oder zu Weihnachten. Gastgeschenk. Mitbringsel. Erinnerungsstück.“ nicht klar, dass „Geschenk“, „Gastgeschenk“, „Mitbringsel“ und „Erinnerungsstück“ um die Auswahlmöglichkeiten handle und es wurde angenommen, dass man auch „zum Geburtstag“ oder „zu Weihnachten“ eingeben könne. Generell führte die Ambiguität des Begriffs „Geschenk“ zu Unklarheit.
Kritisch betrachtet wurde auch, dass Augusta „Gerne.“ als Ersatz für „Ja.“ nicht akzeptierte.
Außerdem zeigten sich die Benutzer von den vielen Eingabemöglichkeiten in KeyExOnesentence beim Erfragen von zusätzlichen Produktkriterien überfordert. Beispielsweise wusste eine Testperson nicht recht, „wie genau man das eingeben soll.“ So sei unklar gewesen, ob man nur die Begriffe oder einem vollständigen Satz eingeben solle. Ein zweiter Test mit KeyExProdukteigenschaften statt KeyExOnesentence ist als einfacher empfunden worden, da hier der Bot den Benutzer mit gezielten Fragen nach Produktname, Ausführung und Preis führe.

Als Reaktion auf die Tests haben wir die doppelte Frage nach der Kaufabsicht aus INTRO herausgenommen. Inzwischen geht Augusta von Anfang an davon aus, dass der Kunde etwas kaufen möchte oder zumindest nach etwas sucht.
Die Ausgabe der Kategorien haben wir vor Ort noch in „Geschenk, Gastgeschenk, Erinnerungsstück und Mitbringsel“ umformuliert. Zudem fragt der Chatbot am Anfang von FIRSTQ ob der Kunde „etwas zum Verschenken oder für einen besonderen Zweck“ suche, wodurch der Begriff „Geschenk“ desambiguiert werden soll. Die Testpersonen reagierten auf diese beiden Änderungen sehr positiv. Es sei nun verständlicher.
Ursprünglich war KeyExProdukteigenschaften nur als Übergangslösung bis zur Fertigstellung von KeyExOnesentence gedacht. Allerdings haben wir uns dazu entschieden, aufgrund der Bemerkungen der Testpersonen, einen zweiten Chatbot mit KeyExProdukteigenschaften anstatt KeyExOnesentence als Alternative zur Verfügung zu stellen.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

WILCOX, BRUCE (2019) *ChatScript*. https://github.com/ChatScript/ChatScript.