Masterarbeit

A Framework for Handheld Recommender Applications

Stefan Reiterer

Graz, 2012

Institute for Software Technology
Graz University of Technology

Supervisor/First reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Alexander Felfernig

Abstract (English)

This work describes the Framework *AdviSense*, which supports the creation of *knowledge-based* Recommender applications for touch screen devices. The innovative user interface of the framework has been kept general, so a wide range of product recommender applications for different domains can be realized. The framework relies on a conjunctive query based representation of domain knowledge. Because of *SQLite* as the basis for the queries and the basis for the configuration of the user interface, a once developed recommender runs on the *iPad*, as well as on *Android Tablets*. The framework supports essential functions for the consultation process: The input of customer requirements and essential functions of *knowledge-based* recommenders, such as the repair of inconsistent requirements. As part of this thesis, three recommenders were developed based on the *AdviSense* framework: two recommender applications from the financial services sector and one recommender to support the identification of suitable mountain bikes.

Abstract (German)

Diese Arbeit beschreibt das Framework *AdviSense*, welches die Erstellung von wissensbasierten Recommender Anwendungen für Touch Devices unterstützt. Die innovative Benutzeroberfläche des Frameworks wurde allgemein gehalten, sodass ein breites Spektrum von Recommender Anwendungen für verschiedene Produkt Domänen realisiert werden kann. Das Framework setzt auf einer *conjunctive Query* basierten Repräsentation des Domänenwissens auf. Aufgrund von *SQLite* als Basis für die *Queries*, als auch als Basis für die Konfiguration der Oberfläche, ist ein entwickelter Recommender sowohl auf dem *iPad*, als auch auf *Android Tablets* lauffähig. Das Framework unterstützt wesentliche Funktionen für den Beratungsprozess: Die Eingabe von Kundenanforderungen und essenzielle Funktionen von wissensbasierten Recommendern, wie beispielsweise die Reparatur von Inkonsistenten Anforderungen.
Im Rahmen dieser Masterarbeit wurden drei Recommender auf Basis des Frameworks umgesetzt: Zwei Recommender Anwendungen aus dem Bereich Finanzdienstleistungen und ein Recommender zur Unterstützung der Identifikation von geeigneten Mountain Bikes.

Acknowledgement

Vorweg möchte mich mich bei meinem Betreuer Univ.-Prof. Dipl.-Ing. Dr.techn. Alexander Felfernig bedanken, welcher mir jederzeit mit einer guten Idee zur Seite stand und mich mit wertvollen Anregungen unterstützte.

Besonders bedanken möchte ich mich auch bei meiner Familie für ihr Vertrauen, ihren Rückhalt und ihre nachsichtige Unterstützung während meines gesamten Studiums, sowie für die Einbringung von Ideen und die Unterstützung der Korrektur dieser Arbeit.

Stefan Reiterer
Graz, 2011

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Place, Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Ort, Datum Unterschrift

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1. Motivation

Touch Devices sind höchst erfolgreich und gewinnen kontinuierlich an Popularität. Sie eröffnen neue Potentiale, Benutzer zu begeistern und ermöglichen komplexe Anwendungen. *Knowledge-based* Recommender Systeme bieten Möglichkeiten, komplexe Produktsortimente für Kunden besser zugänglich zu machen. Integrierte Mechanismen ermöglichen Produktvorschläge, auch für Kundenanforderungen die keine Lösung liefern, was eine positive User Experience garantiert. Die nahezu unbegrenzte Einsatzvielfalt von wissensbasierten Recommendern legt die Entwicklung eines Frameworks nahe, welches die einfache Erstellung von Tablet-Recommendern für verschiedene Domänen ermöglicht. Wissensbasierte Recommender Systeme mit Touch Devices in einem Framework zu kombinieren stellt eine ebenso große Herausforderung wie Motivation dar.

1.2. Ziele

Ziel dieser Masterarbeit ist die Entwicklung eines Frameworks zur effizienten Erstellung wissensbasierter Recommender Anwendungen (Apps) für Tablet Devices. Dazu wird ein User Interface benötigt, welches die Interaktion mit dem Benutzer möglichst intuitiv gestaltet. Die Eingabe der Kundenanforderungen und die Produktpräsentation soll in einem Eingabefenster Platz finden und die Reparaturoptionen in das Gesamtkonzept der GUI integriert sein. Eine weitere Anforderung an das Framework ist die Darstellung von Produktdetails und Informationen zu den Fragen, die dem Kunden während des Empfehlungsprozesses gestellt werden. Das Framework soll auch eine Werbefläche bieten, wo auf andere Produkte verlinkt werden kann. Da der Tablet Markt von Apple *iPad* und *Android Tablets* beherrscht wird, soll ein für das Framework erstellter Recommender auf beiden Plattformen einsetzbar sein.

1.3. Übersicht

Die Masterarbeit ist wie folgt gegliedert:

- Zu Beginn werden in Kapitel 2 (Related Work) die Prinzipien von *Constraint Programing* erläutert. Constraint Programming bildet die Basistechnologie des entwickelten Recommendation Frameworks. Anschließend wird erklärt, wie ein *Constraint Satisfaction Problem* im Allgemeinen gelöst werden kann und wie dessen Darstellung als SQL Query (Conjunctive Query) aussieht. Der zweite Teil des Kapitels setzt sich mit den unterschiedlichen Arten von Recommender Systemen, mit Fokus auf *Knowledge-based Recommendation* auseinander. Auch verschiedene Benutzeroberflächen für diese Systeme werden hier untersucht.

- In Kapitel 3 wird die Idee der Umsetzung eines wissensbasierten Empfehlungsproblems als *conjunctive Query* erläutert, sowie das Management widersprüchlicher Anforderungen in wissensbasierten Empfehlungssystemen.

- Das im Rahmen der Masterarbeit entwickelte Framework *AdviSense* wird in Kapitel 4 im Detail diskutiert. Die Tablet Benutzeroberfläche wird ebenso erklärt, wie der Komponenten-basierte Aufbau des Frameworks.

- Der Entwicklungsprozess für Wissensbasierte Empfehlungssysteme wird anhand der Entwicklung eines Bike-Recommender auf Basis des *AdviSense* Frameworks erklärt. Dabei wird die Entwicklung ausgehend vom Erstellen der Datenbank bis zum ausführbaren Produkt im Detail erkäutert.

Kapitel 2

Related Work

2.1. Constraint Programming

2.1.1. Übersicht

Das Konzept der Constraints wurde erstmals von Sutherland (1963) zum Entwickeln des interaktiven Grafik Systems *Sketchpad* eingesetzt. *Sketchpad* ermöglichte schon damals mittels eines *light pen* direkt auf dem Bildschirm Grafiken zu zeichnen. Constraint Programming (CP) bezeichnet ein Programmierparadigma, welches im Gegensatz zu imperativen Programmiersprachen eine effizientere Art der Problembeschreibung bietet (siehe Barták, 1999).

Definition

> "Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." (siehe Freuder, 1997).

Ein wesentlicher Vorteil von CP liegt in der deklarativen Beschreibung des Problems. Unter anderem werden Constraints im täglichen Leben verwendet um unsere Zeit zu planen, ein typischer Constraint ist: Ich kann von 4 bis 5 Uhr zu dir kommen. Gewöhnlich folgen mehrere Constraints, die im Sinne einer Tagesplanung aufgestellt und gelöst werden können. Die Festlegung von Anforderungen erfolgt also in Form von Constraints. Wie diese Anforderungen erfüllt werden können muss nicht spezifiziert werden. Die Idee von CP ist, für alle festgelegten Anforderungen im Rahmen einer Problemstellung eine Lösung zu finden, welche diesen genügt (siehe Barták, 1999).
Im Laufe der letzten Jahre erwies sich CP für viele Anwendungsgebiete als effizient, unter anderem für Planungssysteme (siehe Castillo et al., 2005), Konfigurationssysteme (siehe Fleischanderl et al., 1998; Sinz et al., 2007) Hardware Verifikation, Echtzeitsysteme und viele mehr (siehe Freuder, 1997). So gewann CP auch im Bereich Recommender Systeme immer größere Bedeutung.

Der klassische Ansatz zur Repräsentation von diesbezüglichen Problemstellungen ist die Formulierung als sogenanntes *Constraint Satisfaction Problem* (CSP). Bezüglich seiner Komplexität wurde bewiesen, dass CSP in der Komplexitätsklasse NP-hart liegt (siehe Hentenryck and Saraswat, 1997). In den letzten Jahrzehnten wurden dennoch performante Methoden zur Lösungsfindung erforscht.

2.1.2. Definition eines *Constraint Satisfaction Problems* (CSP)

Ein CSP besteht nach Brailsford et al. (1999) aus

- einer Menge von Variablen $X = \{x_1, x_2, ..., x_n\}$
- für jede Variable x_i aus X gibt es eine endliche Menge D_i (Domäne der Variable) von möglichen Belegungen
- einer Menge von Constraints $C = \{c_1, c_2, ..., c_m\}$, die Werte, welche von den Variablen angenommen werden können, einschränken. Alle Constraints müssen gleichzeitig erfüllt sein.

Die Menge der Werte D_i müssen nicht in der Menge der Integer liegen, in der weiteren Behandlung und Lösungssuche von CSPs können Variable jedoch nur eine endliche Menge von verschiedenen Werten annehmen.
Kann jede Variable eines CSP mit einem Wert aus ihrer Domäne belegt werden, ohne einen Constraint zu verletzen, so stellt diese Belegung eine Lösung für das gegebene CSP dar. Das Problem ist erfüllbar. Kann keine Belegung gefunden werden, die alle Bedingungen erfüllt, so wird das Problem unerfüllbar genannt.
Mögliche Zielsetzungen der Lösungssuche sind die folgenden:

- eine Lösung zu finden
- alle Lösungen zu finden
- eine optimale Lösung bezüglich eines definierten Optimalitätskriterium zu finden

Constraints beschreiben die Beziehung der Variablen zueinander, zum Beispiel: $x_1 \neq x_2$, $x_2 \leq 2x_3$, $x_1 \geq x_3$. Formal kann ein Constraint $C_{ijk}...$ zwischen x_i, x_j, x_k aus einer beliebigen Untermenge und Kombination von Variablen bestehen. Die einzelnen Variablen können Werte aus deren Domänen D_i, D_j, D_k annehmen, zum Beispiel $C_{ijk} \subseteq D_i \times D_j \times D_k$.
Zur Definition eines CSP wird das Tripel $\{X, D, C\}$ benötigt, wobei X die Variablen, D deren Domänen und C die Constraints beschreibt.
Ein Beispiel für ein CSP sieht wie folgt aus: Die Variable x_1 kann Werte aus der Domäne $\{1,2,3\}$ annehmen, eine weitere Variable x_2 soll die Werte $\{0,1\}$ annehmen können, die dritte Variable x_3 kann nur den Wert $\{3\}$ annehmen. Die Constraints dazu könnten wie folgt aussehen: $C_1 : x_1 \geq x_2$, $C_2 : x_1 \leq x_3$, $C_3 : x_2 \leq x_3$. Kann jede der Variablen mit einem Wert aus ihrer Domäne belegt werden, so stellt diese Belegung eine Lösung für das gegebene CSP dar. Eine Lösung für das beschriebene Problem stellen die Belegungen: $x_1 = 1$, $x_2 = 1$ und $x_3 = 3$ dar.

Beispiel: Cryptarithmetic puzzle

Ein beliebtes Beispiel für ein Problem, das als CSP ausgedrückt werden kann, ist das sogenannte *Cryptarithmetic puzzle* (siehe Russell and Norvig, 2003). Das Puzzle besteht dabei aus Buchstaben, für die eine numerische Belegung gefunden werden soll.

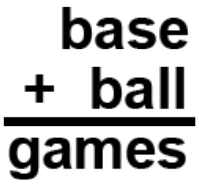

Abbildung 2.1.: Ein Beispiel für ein Cryptarithmetic Puzzle (siehe Brailsford et al., 1999, S. 3)

Die erste Constraint muss festlegen, dass die Buchstaben unterschiedliche Werte besitzen:
$C_0 : b \neq a \wedge b \neq s \wedge b \neq e...$

Um das Puzzle in Abbildung 2.1 zu lösen, werden folgende fünf Constraints aufgestellt:
$C_1 : e + l = 10 * C_1 + s,$
$C_2 : s + l + V_1 = 10 * V_2 + s,$
$C_3 : a + a + V_2 = 10 * V_3 + m,$
$C_4 : b + b + V_3 = 10 * V_4 + a,$
$C_5 : V_4 = g$

Die Variablen $V_1...V_4$ werden eingeführt und können jeweils die Werte 0 oder 1 annehmen. Eine Lösung ergeben die folgenden Variablenbelegungen: $b = 7, a = 4, s = 8, e = 3, l = 5, g = 1, m = 9$. Dieses und andere *Cryptarithmetic Puzzles* sowie ein *Cryptarithmetic Puzzle Solver* sind auf der Universität von Naoyuki Tamura / Dept of CS / Kobe University / Japan * zu finden.

2.1.3. Lösungssuche

Eine Lösung für ein CSP ist dann gefunden, wenn alle Variablen belegt und alle Constraints erfüllt sind. Unterschiedliche Anforderungen an die Lösung bedingen unterschiedliche Vorgehensweisen in der Lösungssuche. Es gibt mehrere Ansätze, welche unterschiedlich performant und für verschiedene Ziele geeignet sind. Wenn beispielsweise alle Lösungen gefunden werden sollen, eignet sich Breitensuche. Diese ist jedoch speicherintensiv. Zum Finden einer einzigen Lösung würde Tiefensuche in Betracht kommen. Algorithmen zum Lösen von CSPs sind meist Suchalgorithmen, welche systematisch mögliche Zuweisungen von Werten aus der Domäne zu Variablen suchen. In weiterer Folge entspricht d der Anzahl der Werte, welche einer Variable zugewiesen werden können und n der Anzahl

*http://bach.istc.kobe-u.ac.jp/llp/crypt.html

der Variable im System. Die Zuweisung ist gültig sofern alle gegebenen Constraints erfüllt sind. Systematische Suche und *Backtracking* sind jedoch schon für einfache Probleme ineffizient. Der Suchbaum besitzt immer eine Tiefe von n, also der Anzahl der Variablen. Der branching Faktor beträgt zu Beginn $n * d$ im nächsten Level $(n-1)d$ und so weiter. Schlussendlich ergibt ein Baum mit $n! * d^n$ Blättern bei einer maximalen Anzahl von Zuweisungen d^n was zu einer exponentiellen Laufzeit führt, welche jedoch durch Einbeziehen der Struktur des gegebenen Problems verbessert werden kann siehe (siehe Russell and Norvig, 2003)).

Backtracking

Im ersten Schritt des Algorithmus wird eine Variable ausgewählt und dieser ein Wert zugewiesen. Danach wird getestet, ob die belegte Variable mit den Constraints konsistent ist. Im Falle von Konsistenz wird die nächste Variable zugewiesen, sollte beim Testen eine Inkonsistenz auftreten, erfolgt ein *Backtracking* Schritt. Da *Backtracking* zu den vollständigen Suchalgorithmen zählt, wird jede existierende Lösung gefunden.
Die folgenden Methoden ergänzen *Backtracking*, (siehe Russell and Norvig, 2003; Brailsford et al., 1999).

Forward Checking

Der *Backtracking* Algorithmus arbeitet nur mit der aktuellen und den zuvor betrachteten Variablen. *Forward Checking* zählt zu den *lookahead* Algorithmen und ergänzt *Backtracking* um die Möglichkeit auch zukünftige Variablen betrachten zu können. Wenn eine Variable v_i zugewiesen wird, testet *Forward Checking* alle noch nicht zugewiesenen Variablen z.B. v_j, welche durch eine Constraint zu v_i verbunden ist und sortiert sämtliche Belegungen für v_j aus, welche eine Inkonsistenz mit der Belegung v_i erzeugen würden. Wenn beim Überprüfen einer zukünftigen Variablen deren Domäne leer wird und ihr kein Wert zugewiesen werden kann, so steht fest, dass ein Backtracking Schritt durchzuführen ist. Das *4-Damen Problem* beispielsweise kann durch Anwendung des *Backtracking* Algorithmus, ergänzt durch *Forward Checking* effizient gelöst werden. Das Problem muss zunächst als CSP formuliert werden. Dabei erfolgt die Positionierung von n-Damen auf einem $n*n$ Schachbrett, sodass sich die Damen nicht gegenseitig schlagen können. Um dies zu gewährleisten dürfen zwei Damen nicht in der selben Zeile, Spalte oder Diagonale positioniert werden. Die Spalten des Schachbretts korrespondieren mit den n Variable $(v_1, ..., v_n)$ des aufzustellenden CSP. Jede Variable besitzt eine Domäne $1...n$, welche die Zeile repräsentiert in der die Dame positioniert wird.
Der Suchbaum für ein $4 * 4$ Schachfeld mittels *Forward Checking* ist in Abbildung 2.2 dargestellt: Die Damen werden Spalte für Spalte auf dem Brett positioniert. Eine Dame zu setzen entspricht dem Zuweisen eines Wertes aus der Domäne zu einer Variable (bzw. Spalte). Nach setzen der ersten Dame in die erste Spalte, der ersten Zeile sind sowohl die Zeile und auch die Diagonale nicht mehr von

anderen Damen belegbar. Die Werte der Felder, welche diese Dame attackieren kann, können also aus der Domäne entfernt werden. Eine Spalte voll von Kreuzen besitzt also keine zukünftigen Belegungsmöglichkeiten mehr. Also folgt *Backtracking* zu einer Stellung, in welcher noch erlaubte Felder zur Verfügung stehen. In Abbildung 2.2 sind diese Verzweigungen mit *X* markiert. *Forward Checking* kann somit zum effizienten Ermitteln von erlaubten Positionen für alle *Damen* auf dem Spielfeld herangezogen werden (siehe Brailsford et al., 1999). Alles in allem benötigt *Forward Checking* in jedem Variablenassignment mehr Schritte als einfaches *Backtracking*, die Größe des Suchbaumes wird dadurch jedoch stark reduziert.

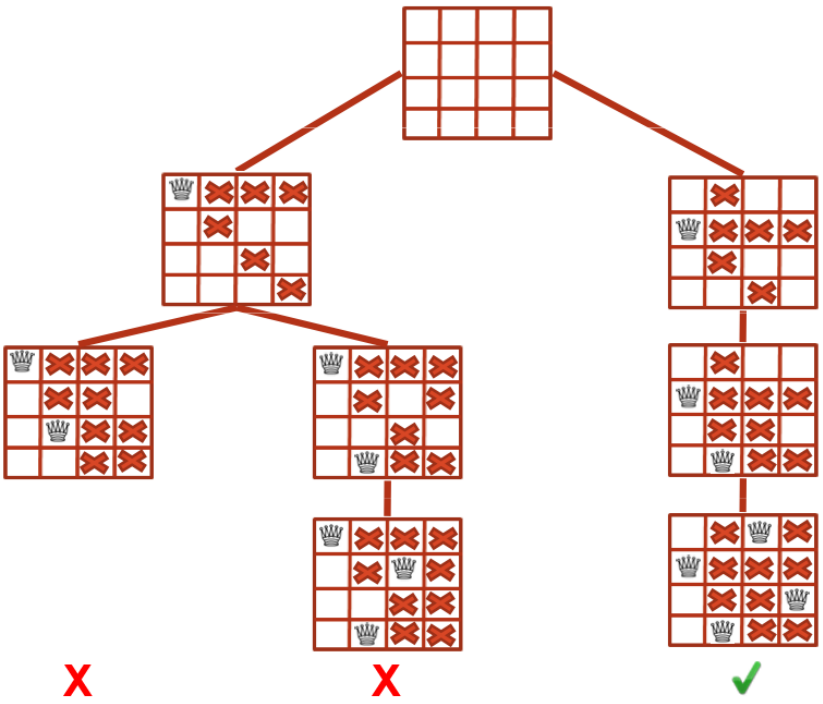

Abbildung 2.2.: Beispiel für *Forward Checking* anhand des Dame Spiels (siehe Brailsford et al., 1999).

Local Consistency

Voraussetzung für *local consistency* (LC) ist, dass alle Constraints eines CSP binär sind. Dies trifft zu, wenn zwei Variablen sich gegenseitig beeinflussen. Die Variablen und Constraints können dann in einem Graphen dargestellt werden, wobei die Variablen durch Knoten veranschaulicht werden, und die Constraints durch Kanten zwischen den Knoten. Die Vorgehensweise von LC teilt sich in 2 Teile: Zum einen in die Konsistenz der Knoten (node consistency), welche bedingt, dass jeder unäre Constraint einer Variablen von allen Werten in der Domain der Variable erfüllt wird. Zum anderen muss die *arc consistency* erfüllt sein. Bedingung für *arc consistency* ist, dass 2 Variablen eines CSP genau dann *arc* konsistent sind, wenn jeder Wert der ersten Variable v_i mit zumindest einem Wert der

zweiten Variable v_j konsistent ist. Die Konsistenz eines CSP ergibt sich aus der *arc* Konsistenz jeder einzelnen Variable mit jeder anderen Variable (siehe Brailsford et al., 1999).

Variable And Value Ordering

Bei der Reihung (im Rahmen der Suche) von Variablen und Domänenwerten kommen zwei Prinzipien zum Einsatz die helfen, die Effizienz der Lösungssuche zu verbessern.
Das *First-Fail* Prinzip basiert auf der Idee schnell auf eine Inkonsistenz zu stoßen und damit unnötige Backtracking Schritte zu vermeiden. In diesem Zusammenhang wird zuerst jene Variable instanziert, welche die wenigsten noch verbleibenden Möglichkeiten zur Instanziierung besitzt. Falls zwei oder mehr gleichgestellte Variable auftreten, so wird die Variable mit der größten Anzahl an verbundenen Constraints verwendet (*degree heuristic*).
Das Gegenstück zu *First Fail* auf der Ebene der Auswahl eines Wertes für eine bestimmte Variable ist das *Succeed First* Prinzip. Dieses wählt zunächst jenen Variablenwert aus, welcher die meisten Belegungsmöglichkeiten für die verbleibenden Variablen übrig lässt. Das Ziel dieser Methode ist, möglichst schnell eine Lösung zu identifizieren.

2.1.4. Darstellung eines CSP als Datenbank Query

Definition

Ein CSP (Definition siehe Kapitel 2.1.2), kann als *conjunctive Query* dargestellt werden, es ist daher möglich, ein CSP mit Hilfe sogenannter *conjunctive Queries* zu lösen. Dies bringt zum einen den Vorteil der Verwendung der Sprache SQL zum beschreiben des CSP, zum anderen den Vorteil jedes System auf dem eine Datenbank verfügbar ist verwenden zu können, also auch mobile Systeme welche SQLite unterstützen. Es muss kein Constraint Solver installiert werden um das CSP lösen zu können.

Die Elemente der Menge $\{X, D, C\}$ müssen in SQL beschrieben werden:

- Für jedes Element aus der Menge der Variablen $X = \{x_1, ..., x_n\}$ wird eine Datenbank Tabelle angelegt.
- Für jede Variable x_i aus X, gibt es eine endliche Menge von Belegungen, D_i, diese werden als Werte in der Datenbanktabelle gespeichert.
- Die Menge der Constraints $C = \{c_1, ..., c_n\}$ muss gleichzeitig erfüllt sein. Die Constraints werden mittels konjunktiver Klauseln in der Datenbank Query beschrieben.

Abbildung 2.3 zeigt den beispielhaften Aufbau eines CSP, wobei X die Variablen, D die Domänen und C_1, C_2, C_3 die Constraints (Abb. 2.3(a)) darstellen. Die Umsetzung des CSP als *conjunctive Query* wird in Abbildung 2.3(b) gezeigt.

```
X = {v0.value, v1.value, v2.value}
D = { v0, v1, v2 }

C1 = (v0.value < v1.value)

C2 = (v1.value <> v2.value)

C3 = (v0.value > v2.value)
```

(a)

```
SELECT v0.value, v1.value, v2.value
FROM v0, v1, v2
WHERE
(v0.value < v1.value)
AND
(v1.value <> v2.value)
AND
(v0.value > v2.value)
```

(b)

Abbildung 2.3.: Darstellung eines CSP als Conjunctive Query. (a) zeigt die Formulierung des CSP, in (b) wird die Darstellung des CSP als *conjunctive Query* gezeigt.

Beispiel

Als Beispiel wird das *Cryptarithmetic puzzle* aus Kapitel 2.1.2 herangezogen. Die Buchstaben der Addition **base + ball = games** (siehe Abb. 2.1) sollen durch Zahlen von 0..9 ersetzt werden und das Ergebnis der Addition korrekt sein. Zunächst wird für jede Variable, also für jeden Buchstaben des Puzzles $V = \{a,b,e,g,l,m,s\}$ eine Tabelle angelegt. Listing 2.1 zeigt die Erzeugung der Tabelle für Variable a. Die Erzeugung der weiteren Tabellen, $b\_tbl, e\_tbl, g\_tbl, l\_tbl, m\_tbl, s\_tbl$ sowie der Tabellen für den Übertrag der Addition $x1\_tbl, x2\_tbl, x3\_tbl, x4\_tbl$ erfolgt nach dem selben Prinzip.

Listing 2.1: Beispiel für die Erzeugung von Tabelle *a_tbl* für die Variable *a*

```
CREATE TABLE "a_tbl" ("key" INTEGER PRIMARY KEY  AUTOINCREMENT  NOT NULL , "v"
    INTEGER)
```

Um das Ziel des *Cryptarithmetic puzzle*, eine Ersetzung der Buchstaben durch Zahlen, zu erreichen, sodass das Ergebnis der Rechnung korrekt ist, wird für alle Variablen die selbe Domäne $D = \{0,1,2,3,4,5,6,7,8,9\}$ festgelegt. Der erste Constraint definiert, dass alle Variablen unterschiedlich belegt sein müssen, dies wird mittels $C_0 : b \neq a \wedge b \neq s \wedge b \neq e...$, bzw. der *conjunctive Query* aus Listing 2.2 erreicht.

Listing 2.2: SQL Statement für C_0: Alle Variablen müssen unterschiedliche Werte besitzen

```
(a_tbl.v <> b_tbl.v) AND (a_tbl.v <> e_tbl.v) AND
(a_tbl.v <> g_tbl.v) AND (a_tbl.v <> l_tbl.v) AND
(a_tbl.v <> m_tbl.v) AND (a_tbl.v <> s_tbl.v) AND
(b_tbl.v <> e_tbl.v) AND (b_tbl.v <> g_tbl.v) AND ...
```

Die weiteren Constraints für die Addition aus Kapitel 2.1.2 werden ebenfalls in SQL Syntax übersetzt, die Tabellen für die Variablen eingesetzt und die einzelnen Constraints mit *AND* verknüpft.

Listing 2.3: SQL Statements für *C_1...C_5*

```
(e_tbl.v + l_tbl.v = s_tbl.v + 10 * x1_tbl.v)
```

```
2  AND
3  (x1_tbl.v + s_tbl.v + l_tbl.v = e_tbl.v + 10 * x2_tbl.v)
4  AND
5  (x2_tbl.v + a_tbl.v + a_tbl.v = m_tbl.v + 10 * x3_tbl.v)
6  AND
7  (x3_tbl.v + b_tbl.v + b_tbl.v = a_tbl.v + 10 * x4_tbl.v)
8  AND
9  (x4_tbl.v = g_tbl.v)
```

Für das fertige *SELECT* Statement werden an C_0 (2.2), $C_1 ... C_5$ (2.3) angefügt, davor die gesuchten Spalten $a\_tbl.v$, $b\_tbl.v$, $e\_tbl.v$, $g\_tbl.v$, $l\_tbl.v$, $m\_tbl.v$, $s\_tbl.v$ eingefügt und schließlich noch die Tabellen, aus denen Daten selektiert werden, angegeben: a_tbl, b_tbl, e_tbl, g_tbl, l_tbl, m_tbl, s_tbl, $x1\_tbl$, $x2\_tbl$, $x3\_tbl$, $x4\_tbl$. Die zusammengefügte SQL Abfrage ist in Listing 2.4 dargestellt. Wird diese Abfrage auf einer Datenbank ausgeführt, ist das Resultat die Belegung $a\_tbl.v = 4$, $b\_tbl.v = 7$, $e\_tbl.v = 3$, $g\_tbl.v = 1$, $l\_tbl.v = 5$, $m\_tbl.v = 9$, $s\_tbl.v = 8$.

Listing 2.4: Das fertige SQL Statement

```
1  SELECT a_tbl.v, b_tbl.v, e_tbl.v, g_tbl.v, l_tbl.v, m_tbl.v, s_tbl.v
2  FROM a_tbl, b_tbl, e_tbl, g_tbl, l_tbl, m_tbl, s_tbl, x1_tbl, x2_tbl, x3_tbl,
       x4_tbl
3  WHERE
4  (a_tbl.v <> b_tbl.v) AND (a_tbl.v <> e_tbl.v) AND
5  (a_tbl.v <> g_tbl.v) AND (a_tbl.v <> l_tbl.v) AND
6  (a_tbl.v <> m_tbl.v) AND (a_tbl.v <> s_tbl.v)
7  AND
8  (b_tbl.v <> e_tbl.v) AND (b_tbl.v <> g_tbl.v) AND
9  (b_tbl.v <> l_tbl.v) AND (b_tbl.v <> m_tbl.v) AND
10 (b_tbl.v <> s_tbl.v)
11 AND
12 (e_tbl.v <> g_tbl.v) AND (e_tbl.v <> l_tbl.v) AND
13 (e_tbl.v <> m_tbl.v) AND (e_tbl.v <> s_tbl.v)
14 AND
15 (g_tbl.v <> l_tbl.v) AND (g_tbl.v <> m_tbl.v) AND
16 (g_tbl.v <> s_tbl.v)
17 AND
18 (l_tbl.v <> m_tbl.v) AND (l_tbl.v <> s_tbl.v)
19 AND
20 (m_tbl.v <> s_tbl.v)
21 AND
22 (e_tbl.v + l_tbl.v = s_tbl.v + 10 * x1_tbl.v)
23 AND
24 (x1_tbl.v + s_tbl.v + l_tbl.v = e_tbl.v + 10 * x2_tbl.v)
25 AND
26 (x2_tbl.v + a_tbl.v + a_tbl.v = m_tbl.v + 10 * x3_tbl.v)
27 AND
28 (x3_tbl.v + b_tbl.v + b_tbl.v = a_tbl.v + 10 * x4_tbl.v)
29 AND
30 (x4_tbl.v = g_tbl.v)
```

2.2. Recommender Technologien

2.2.1. Übersicht

Recommender Systeme unterstützen Benutzer beim Identifizieren spezifischer Produkte in informationsreichen Domänen (siehe Felfernig and Burke, 2008). Der Hintergrund von Empfehlungsproblemen ist im Bereich der künstlichen Intelligenz zu finden. Einige dieser Grundlagen waren bereits vor deren Verwendung im Bereich der Recommender Systeme Gegenstand der Forschung. In diesem Kapitel werden verschiedene Arten von Recommender Systemen im Detail betrachtet.
Mitte der 1990er Jahre, mit der Verbreitung von e-Commerce (siehe Jannach et al., 2010), entstanden erste Empfehlungssysteme für Bücher, CDs und Filme. In diesen Bereichen trifft man oft *collaborative Recommender Systems*, welche Rating Daten von Benutzern verwenden, um eine Empfehlung zu erstellen. Amazon[†] verwendet unter anderem einen kollaborativen Ansatz zur Empfehlung von Büchern und mittlerweile vielen anderen Produkten (siehe Linden et al., 2003). Eine weitere Ausprägung von Recommender Systemen verwendet Meta Daten der Produkte; diese werden *Content-based Recommender Systems* genannt. Ihr Anwendungsbereich erstreckt sich von Webseiten, Restaurants, über Fernsehprogramme (siehe Pazzani and Billsus, 2007), bis hin zur Anwendung in Spamfiltern (siehe Seewald, 2007). Seit einigen Jahren gewinnt eine weitere Art von Recommender Systemen zunehmend an Bedeutung: *Knowledge-based Recommender Systems* stellen die dritte Ausprägung, neben bereits erwähnten inhaltsbezogenen und kollaborativen Ansätzen dar. Diese Art von Empfehlungssystem verwendet zusätzliches Wissen über die Zusammenhänge von Produkteigenschaften und Benutzeranforderungen, um eine Empfehlung zu generieren. Die Anwendungsbereiche sind vielfältig, wissensbasierte Recommender für Finanzdienstleistungen (siehe Felfernig, 2005a; Felfernig et al., 2007) wurden ebenso entwickelt, wie für Restaurants (siehe Burke, 2002) oder Digitalkameras (siehe Smyth et al., 2004).
Die verschiedenen Technologien besitzen je nach Anwendungsbereich Vor- und Nachteile. Damit kollaborative Systeme funktionieren, muss eine große Datenbank aufgebaut werden, welche das Nutzerverhalten speichert, wogegen inhaltsbasierte Recommender, die nur Eigenschaften der zu empfehlenden Produkte benötigen, kein sogenanntes *ramp-up* Problem besitzen. Der Anwendungsbereich betrifft komplexe Produkte und Dienstleistungen, etwa im Elektroniksektor die Auswahl der richtigen Digitalkamera oder im Finanzdienstleistungssektor die Empfehlung eines Produktes zum Vermögensaufbau. Dieses Kapitel zeigt eine Übersicht über bestehende Recommender Systeme und deren Unterschiede.

Definition eines Recommender Systems nach Burke und Felfernig *Any system that guides the user in a personalized way to interesting or useful objects in a large space of possible options or that produces such objects as output.* (siehe Felfernig and Burke, 2008, S. 1)

[†]http://www.amazon.com

2.2.2. Wissensquellen

Um eine Empfehlung generieren zu können, ist je nach Recommender System unterschiedliches Wissen über Produkt, Benutzer und Anwendungskontext notwendig. Wissen kann explizit erfasst werden, indem der Benutzer aufgefordert wird, ein Rating zu einem Produkt abzugeben. Aber auch implizit gewonnen werden, beispielsweise durch Speichern der Produkte, die der Kunde auf einer Webseite gekauft oder angesehen hat. Nach Felfernig and Burke (2008) stehen drei Bereiche zum Sammeln von Hintergrundinformationen zur Verfügung: der *social*, der *individual* und der *content* Bereich.

- Im *social* Bereich werden Meinungen von Benutzern gesammelt und für ein Voting verwendet. Ein Beispiel für ein kollaboratives Voting von Web Links für Webseiten ist der Page Rank Algorithmus (siehe Brin and Page, 1998)). Dieser Bereich ist für den kollaborativen Recommender besonders wichtig. Es wird ein Benutzerprofil erstellt, in welchem der Benutzer seine Meinung zu einem Produkt mitteilt. Eine Filmvertriebsplattform könnte verkaufte Filme auf einer Skala von 1-4 von Kunden raten lassen, wobei 1 = gefällt mir sehr gut, bis 4 = gefällt mir nicht bedeutet und erhält so ein gemeinschaftliches Voting für einen Film.
- *individual* Um personalisierte Empfehlungen zu erstellen muss individuelles Wissen über den Benutzer verwendet werden. Individuelle Daten sind aus Sicht eines anderen Benutzers wie zuvor beschrieben als soziale Daten zu sehen.
  Wesentlich für die Mächtigkeit eines Recommenders ist die Eingabe und Verarbeitung dieser individuellen Benutzerdaten. Der Benutzer verwendet das Empfehlungssystem mit einer bestimmten Absicht und das System muss darauf reagieren. Der Benutzer bringt seine Anforderungen gelenkt durch die Benutzeroberfläche in das System ein, er / sie könnte zum Beispiel ein Produktattribut suchen.
  Eine weitere Eingabemöglichkeit könnte eine Bedingung sein. Ein Wohnungssuchender mit Auto, in einer Großstadt würde dem System folgende Bedingung mitteilen: ”Zur Wohnung muss ein Parkplatz gehören.” Die Eingabemöglichkeit einer Bedingung wird kaum ausreichen für die Suche einer Wohnung, daher folgt eine Reihe weiterer Bedingungen wie ”Hunde müssen erlaubt sein”, ”Die Größe muss mindestens 55 qm betragen” und ”Sie darf nicht über 500 Euro kosten”.
  Ein weiterer Aspekt der individuellen Wissensgewinnung ist daher auch die Vergabe von Präferenzen an die gestellten Anforderungen. Hat der Preis niedrigere Priorität als der Parkplatz, könnte auch eine Wohnung empfohlen werden, die etwas teurer ist dafür den Parkplatz bietet.
  Aus dem Kontext des Benutzers lassen sich sehr detaillierte Rückschlüsse für Empfehlungen ziehen. Der Recommender *CoCo*, entwickelt für *Jimbo-cho* einen Buchmarkt mit über 150 Buchläden (siehe Si et al., 2005), verwendet zusätzlich zum Benutzerprofil dessen Umgebungsparameter wie Zeit, Ort und Status um Empfehlungen zu generieren.
- *content* Der Nutzen inhaltlicher Informationen über ein Produkt ist stark von der Produkt-

domäne abhängig. In Bereichen wie Musik und Film ist es schwierig, die Bedürfnisse eines Benutzers mit einem Produkt übereinzustimmen, da diese unter anderem von der Stimmung des Kunden, oder auch von seinem persönlichen Geschmack abhängig sind. Im Gegensatz dazu sind Produkte, die eher analytisch logisch gekauft werden, wie beispielsweise Digitalkameras, Webhosting-Dienstleistungen oder Finanzdienstleistungen, einfacher zu empfehlen. Die Produktattribute sind wesentlicher Bestandteil des Wissens über ein Produkt. Eine Digitalkamera würde beispielsweise die Attribute ”Megapixel = 8,0”, ”optischer Zoom = 3x”, Speicher = 128mb” besitzen, diese Key / Value Paare könnten in einer Datenbank abgelegt werden (siehe Smyth et al., 2004).

- *Domänenwissen* In manchen Fällen kann es für einen Recommender wichtig sein, zusätzlich zum Wissen über Attribute von Produkten genaueres Wissen über den Anwendungsbereich und den Zweck den ein Produkt erfüllt, zu besitzen.
  Means-Ends Wissen, definiert wie ein Produkt (*means*) ein bestimmtes Bedürfnis eines Benutzers (*ends*) erfüllen kann. Um das Bedürfnis ”familientauglich” zu erfüllen, muss das Produkt (in diesem Fall ein Auto) eine größere Menge von Eigenschaften erfüllen, etwa einen großen Kofferraum und viel Platz für Passagiere.
  Mit *Feature ontology* wird das Wissen über den Zusammenhang von Eigenschaften eines Produktes bezeichnet, beispielsweise ”Amarok” ist ein Pickup Modell des Herstellers Volkswagen.
  Domain Constraints bezeichnen jene Bedingungen, die von der Domain an die Produkte gestellt werden. Ein Rennrad beispielsweise wird einem Kunden nur angeboten, wenn dieser auch Rennfahrer ist (da für einen ”nicht” Rennfahrer ein Fahrrad ohne Beleuchtung für den Straßenverkehr nicht geeignet wäre). Eine bestimmte Versicherung wird nur angeboten, wenn ein Kunde Nichtraucher ist. Ein *Domain Constraint* im Kontext eines Recommenders für hosting Produkte sieht wie folgt aus: Möchte der Kunde Multimedia Inhalte auf dem Server platzieren, so legt ein *Domain Constraint* fest, dass für diese Inhalte ein hosting Produkt mit hoher Bandbreite gekauft werden muss, da ansonsten mit den Inhalten die Bandbreite überschritten wird. Die Kundenanforderung ”Multimedia Inhalte” und die Produkteigenschaft ”Bandbreite” werden also in Beziehung gesetzt.

2.2.3. Collaborative Recommendation

Collaborative Recommendation, oft auch als *Collaborative Filtering* (CF) bezeichnet, war eine Vorreitertechnik im Bereich der Empfehlungen im WWW. Erste Publikationen zum Thema CF erschienen in den 90er Jahren. Ein Beispiel dafür ist eine Arbeit über ein News Empfehlungssystem von Resnick et al. (1994). In Shardanand and Maes (1995) wird ein kollaborativer Musik Recommender beschrieben, eine weitere Arbeit aus den frühen 90er Jahren wurde von Goldberg et al. (1992) über das E-mail Filter System Tapestry verfasst. Seit dieser Zeit gewannen CF Technologien zunehmend an Bedeu-

tung.
Zwei Personen, Alice und Bob sahen ”Fluch der Karibik Teil 1” und ”Fluch der Karibik Teil 2” im Kino, beiden gefielen die Filme sehr gut. Bob gefällt der neueste Teil der Serie ”Fluch der Karibik 3”, den Alice noch nicht gesehen hat, ebenfalls sehr gut. Er empfiehlt Alice also den Film anzusehen. Bekommt Alice nicht nur von Bob diese Filmempfehlung, sondern auch von weiteren Freunden mit dem selben Filmgeschmack, so verstärkt sich die Wirkung der Empfehlung. Nach diesem Prinzip empfehlen sich Menschen untereinander viele Produkte des Alltags wie CDs, Bücher, Urlaube usw.
Dieses soziale Verhalten wird in ähnlicher Form als Grundlage für CF (siehe Aciar et al., 2007) verwendet. Es wird davon ausgegangen, dass eine Menge von Benutzern mit den selben Interessen, die in der Vergangenheit ähnlich handelten, dies auch in der Zukunft tun. Internet Plattformen bieten dem Kunden teilweise explizit die Möglichkeit, ein Kundenprofil zu erstellen und generieren anhand dieses Wissens eine individuelle Empfehlung. Zusätzlich wird meist auch implizites Wissen, also unbewusst hinterlassene Spuren im WWW, über Produkte, die angesehen bzw. gekauft wurden, für Empfehlungen verwertet.

User-based Collaborative Filtering

Das Erstellen CF basierter Empfehlungen geht meist auf Maschinelles Lernen zurück, ein bekannter Ansatz ist der *Nearest Neighbour* Algorithmus.
Für *User-based Collaborative Filtering* werden Produktevaluierungen von Benutzern aus der Vergangenheit in einer Datenbank abgelegt (siehe 2.1). Vom aktuellen Benutzer (Bob) werden ebenfalls die bereits evaluierten Produkte gespeichert. Nun werden jene Benutzer identifiziert, deren Bewertung Bobs am ähnlichsten ist. Für jedes Produkt, das Bob noch nicht gesehen hat, kann eine Prognose berechnet werden, basierend auf den Rating Werten ähnlicher Benutzer. Die Implementierung folgt dem Konzept, dass Benutzer mit ähnlichen Interessen in der Vergangenheit, auch in Zukunft ähnliche Interessen haben werden (siehe Jannach et al. (2010)).

User	Film 1	Film 2	Film 3	Film 4
Bob	2	4	3	?
User1	2	1	2	2
User2	1	4	3	3
User3	2	1	3	1

Tabelle 2.1.: Rating Matrix für *Collaborative Filtering*

Tabelle 2.1 zeigt, wie eine Datenbank für einen CF Recommender für Filme aussehen könnte. Mit dem Namen Bob wird jener Benutzer bezeichnet, der die Empfehlung erhält. User 1 - 3 sind Benutzer, welche Film 1, 2 und 3 ebenso gesehen haben, wie Bob. Die Datenbankeinträge bezeichnen das Rating der Benutzer für die einzelnen Filme. User 3 hat Film 1 zum Beispiel mit 3 von 4 Punkten

bewertet, wobei das Rating von ”5”, gefällt mir sehr gut, bis ”1” gefällt mir gar nicht reicht. Das ”?” für Film 4 bedeutet, dass Bob diesen Film noch nicht gesehen und bewertet hat. Die Aufgabe des Algorithmus ist es, herauszufinden, welches Rating Film 4 von Bob bekommen würde, im Falle eines hohen Ratings wäre es sinnvoll, Bob Film 4 zu empfehlen.
Zunächst wird ein *similarity Rating* erstellt, um herauszufinden welcher der Benutzer Bob am ähnlichsten ist, siehe Tabelle 2.2.

User	Durchschnittsbewertung
Bob	$(2+4+3)/3 = 3$
User1	$7/4 = 1,75$
User2	$11/4 = 2,75$
User3	$8/4 = 2$

Tabelle 2.2.: Berechnung der Durchschnittsbewertung von User 1, 2, 3 und Bob

$$sim(a,b) = \frac{\sum_{p \in P} (r_{a,p} - \overline{r_a})(r_{b,p} - \overline{r_b})}{\sqrt{\sum_{p \in P} (r_{a,p} - \overline{r_a})^2} \sqrt{\sum_{p \in P} (r_{b,p} - \overline{r_b})^2}} \tag{2.1}$$

Zum Berechnen der Ähnlichkeit zwischen zwei Benutzern a und b wird der *Pearson Correlation Coefficient* herangezogen. Die Berechnung der Ähnlichkeiten der Benutzerratings aus Matrix 2.1 erfolgt mittels Formel 2.1 (vgl. Jannach et al., 2010, S. 14) daraus ergeben sich die in Tabelle 2.3 dargestellten Ähnlichkeiten mit dem Benutzer Bob.

a, b	Similarity
Bob, User1	-0,45
Bob, User2	0,94
Bob, User3	-0,35

Tabelle 2.3.: Ähnlichkeiten von Bob mit User 1,2,3

Die Ergebnisse der *Pearson Korrelation* liegen zwischen -1, negative Korrelation und 1 positive Korrelation, die größte Ähnlichkeit, also den höchsten Wert im Vergleich mit Bob besitzt User 2 mit 0,94, User 1 und 2 besitzen negative Werte. Da User 2 die Filme in der Vergangenheit sehr ähnlich bewertete wie Bob, wird aus diesen Werten Bobs fiktives Rating für Film 4 berechnet.

$$pred(a,b) = \overline{r_a} + \frac{\sum_{b \in N} sim(a,b) * (r_{b,p} - \overline{r_b})}{\sum_{b \in N} sim(a,b)} \tag{2.2}$$

Die Berechnung des Vorhersagewertes für Bob und User 2 nach Formel 2.2 ergibt ein fiktives Rating für Film 4 von 2,7. In einer individuellen Empfehlungsliste auf einer E-Commerce Seite wäre es durchaus sinnvoll, Bob dieses Produkt zu empfehlen. Wie aus diesem Beispiel ersichtlich, werden zur Anwendung von CF Algorithmen keine Daten über die zu empfehlenden Produkte oder deren Eigenschaften benötigt. Daraus resultiert, dass auch neue Produkte sehr einfach eingefügt werden können. Kollaborative Strategien unterliegen jedoch dem *Ramp-up Problem*, daher ist deren Einsatz nur auf Produktplattformen mit einer großen Anzahl von aktiven Benutzern sinnvoll.

Item-based Collaborative Filtering

Der zweite Ansatz zum Berechnen kollaborativer Empfehlungen wird *Item-based CF* genannt. Dieser zieht die kollaborativen Ratings der Gegenstände bzw. Produkte heran, um eine Empfehlung zu generieren. Die Ausgangsbasis von *Item-based CF*: Für Film 4 gibt es noch kein Rating von Benutzer Bob (siehe Tabelle 2.1). Ziel ist es einen Film zu finden welcher ähnlich zu Film 4 ist aber von Bob schon bewertet wurde. Anschließend kann aus den Bewertungen des ähnlichsten Filmes ein Rating für Film 4 abgeleitet werden. Zum Berechnen einer *Item-basd Nearest Neighbour* Empfehlung (siehe Sarwar et al., 2001; Jannach et al., 2010), kann ebenso wie für *User-based CF* der *Pearson Correlation Coefficient* herangezogen werden. Dazu muss zunächst wieder der Durchschnittswert für jeden Film berechnet werden (siehe Tabelle 2.4). Anschließend wird in Formel 2.1 eingesetzt. Tabelle 2.5 zeigt die der berechneten Ähnlichkeiten zwischen Film 4 und den Filmen 1- 3. Film 2 ist Film 4 mit einem Wert von von 0,74 am ähnlichsten, das fiktive Rating für Film 4 wird nach Formel 2.3 berechnet.

User	Durchschnittsbewertung
Film 4	$(2+3+1)/3 = 3$
Film 1	$7/4 = 1,75$
Film 2	$10/4 = 2,5$
Film 3	$11/4 = 2,75$

Tabelle 2.4.: Berechnung der Durchschnittsbewertung von User 1, 2, 3 und Bob

a, b	Similarity
Film 4, Film 1	0,45
Film 4, Film 2	0,74
Film 4, Film 3	0,43

Tabelle 2.5.: Ähnlichkeiten von Film 4 mit Film 1,2,3. Die Bewertung von Film 2 ist jener von Film 4 mit einem Wert von 0,74 am Ähnlichsten.

$$pred(u,p) = \frac{\sum_{i \in ratedItems(u)} sim(i,p) * (r_{u,i})}{\sum_{i \in ratedItems(a)} sim(i,p)} \tag{2.3}$$

2.2.4. Content-based Recommendation

Für *Content-based Recommendation* werden Meta Daten der Objekte als Grundlage zum Erstellen von Empfehlungen verwendet. Adomavicius and Tuzhilin (2005) geben einen Überblick über inhaltsbasierende Recommender Technologien. Die notwenige Information wird dabei direkt aus den Objekten oder aus Beschreibungen der Objekte gewonnen, beispielsweise aus Dokumenten, welche mittels Methoden des *Information Filtering* bzw. *Information Retrieval* analysiert werden. Das Einfügen neuer Dokumente, sowie deren sofortige Einbindung in den Empfehlungsmechanismus stellt durch die automatische Analyse kein Problem dar. Dafür sind allerdings Methoden zur Extraktion von *Keywords* aus Beschreibungen notwendig. Andere Produktdaten wie zum Beispiel der Preis eines Films müssen explizit hinzugefügt werden. Mithilfe gewonnener Daten können Rankings nach bestimmten Kriterien erstellt werden.
Rating Informationen, wie sie für kollaborative Systeme verwendet werden, werden nicht benötigt, der Fokus liegt auf den inhaltlichen Daten der zu empfehlenden Objekte. Ein Beispiel dafür ist der Musik Recommender Pandora[‡], dessen Grundlage eine detaillierte Darstellung musikalischer Inhalte ist. Es muss sichergestellt werden, dass die Produktrepräsentation detailliert und einheitlich für das gesamte Produktportfolio ist.

Beispiel für Content-based Recommendation

Grundlage für eine *Content-based Recommendation* sind inhaltliche Eigenschaften, für Filme wie in Tabelle 2.6 aufgeführt, beispielsweise Genre, Regisseur und Preis. Beschreibt der Benutzer seine Anforderungen auf Grundlage derselben Objektattribute, so besteht die Aufgabe des Recommenders darin, die Benutzeranforderungen mit den Produktattributen zu matchen. Basierend auf den Attributen bereits konsumierter Items erfolgt die Identifikation potentieller Kandidaten für eine Empfehlung auf Basis der Ähnlichkeit neuer und konsumierter Items. Wären die Daten aus Tabelle 2.6 in einer relationalen Datenbank gespeichert, so könnten potentielle Items für eine Empfehlung mittels einer SQL Query abgefragt werden.

[‡]http://www.pandora.com

Title	Genre	Regisseur	Preis
Fluch der Karibik	*Abenteuer*	*Verbinski*	13,99
Hangover	Komödie	Todd Phillips	14,99
Fluch der Karibik 2	*Abenteuer*	*Verbinski*	16,99

Tabelle 2.6.: Eine Filmattribute Tabelle. Fett gedruckt: *Content basierte* Ähnlichkeiten der *Fluch der Karibik* und *Fluch der Karibik 2*

2.2.5. Knowledge-based Recommendation

Nach Burke (2000) werden alle Recommender als *knowledge-based* bezeichnet, welche auf Wissen basieren, das nicht durch Methoden des *Content-based* (2.2.4) oder *Collabriative Filtering* (2.2.3) gewonnen werden kann. Die Vorgangsweise in der Verwendung eines wissensbasierten Recommenders ist interaktiv, der Benutzer kann seine Anforderungen an das System weitergeben, welches ihm Produkte vorschlägt, bzw. kann die Eingabe der Anforderungen als Spezifikation eines Problems durch den Benutzer gesehen werden, die darauf folgende Ausgabe der Produkte als Lösungsvorschlag für das Problem.

Definition von *Knowledge-based* Recommendation nach Burke und Felfernig *Im Mittelpunkt des Wissensbasierten Ansatzes steht die Anforderung des Benutzers und wie ein Produkt oder eine Dienstleistung diese Anforderungen erfüllen kann.* (aus Felfernig and Burke, 2008, S. 4)

Wissensbasierte Empfehlungssysteme werden in Produktdomänen eingesetzt, in welchen es nicht möglich ist, auf eine große Anzahl von Benutzer Ratings zurückzugreifen, wie es für eine kollaborative Empfehlungen (siehe Kapitel 2.2.3) notwendig ist. Im Gegensatz zu CF Recommendern ist das Wissen, das für die Empfehlung benötigt wird, bereits vorhanden, daher gibt es kein *ramp-up* Problem. Nachteilig ist der wesentlich komplexere Aufbau des Domänenwissens, das hinter dem Recommender steht (siehe Kapitel 2.2.2). Zu den vielfältigen Einsatzbereichen gehören großteils Domänen, deren Produkte nur jährlich oder seltener gekauft werden. Beispiele dafür sind Produkte im Bereich Finanzdienstleistungen, in welchen Unterstützung für Kunden durch Online-Systeme, als auch für Vertriebsmitarbeiter geboten werden kann (siehe Felfernig, 2005a)). Produkte mit einer großen Anzahl von Attributen und Konfigurationsmöglichkeiten wie Computer (siehe Felfernig, 2005b) gehören ebenfalls zum Anwendungsbereich von wissensbasierten Systemen. Auch für Digitalkameras (siehe Smyth et al., 2004) und Mobiltelefone (siehe Chun and Hong, 2001) wurden bereits Empfehlungssysteme entwickelt.
In Felfernig and Gula (2006) wird eine Nutzerstudie vorgestellt, welche die Auswirkungen verschiedener Empfehlungsmechanismen auf die Kundenakzeptanz von Recommender Technologien beschreibt. Die Studie konzentriert sich im Detail auf den Einfluss wissensbasierter Recommender auf das Kaufverhalten von Benutzern. Als Grundlage der Studie wurden 8 verschiedene Versionen eines Internet-Provider Recommenders erstellt. Die Eigenschaften dieser Versionen sind: positiv, negativ formulier-

te, oder keine Erklärungen, mit, ohne oder mit automatischem Produktvergleich. Dazu wurde auch eine reine Produktliste, ohne jegliche Recommender Funktionalität angeboten. Die impliziten und expliziten Reaktionen von 116 online Benutzern auf diese Versionen wurden betrachtet. Verglichen mit der gewöhnlichen Produktliste ohne Recommender Funktionalität, zeigt die Studie eindeutige Vorteile bezüglich der Benutzerzufriedenheit während der Interaktion mit dem Recommender System. Auch das Vertrauen der Benutzer, die optimale Lösung gewählt zu haben ist beim Einsatz eines Recommender Systems wesentlich höher als beim Auswählen des Produktes aus der Produktliste. Die Arbeit kann als Guideline zum Erstellen wissensbasierter Recommender Systeme herangezogen werden.

Die Problemstellungen wissensbasierte Recommender zu entwerfen sind vielfältig: Zum einen muss ein Weg gefunden werden, die Spezifikation der Anforderungen durch den Benutzer möglichst intuitiv zu gestalten, zum anderen müssen die Anforderungen in eine Form gebracht werden, welche das System lösen kann. Zum Lösen des durch Benutzeranforderungen gegebenen Problems haben sich zwei Wege entwickelt: *Case-based Recommender* und *Constraint-based Recommender*. Beide Systeme gehen nach folgendem Grundprinzip vor:

1. Anforderungen des Benutzers sammeln
2. Lösungen zu den gesammelten Anforderungen finden
3. falls keine Lösung gefunden wurde: Anforderungen ändern

Case-based Recommendation

Es wird versucht ein Produkt zu finden, das jenem Produkt möglichst ähnlich ist, welches sich der Benutzer vorstellt. Im Empfehlungsprozess wird das Feedback (bzw. die Kritik, siehe *Critique-based* Recommender (Burke, 2000)), das der Benutzer dem System gibt, verwendet, um die Empfehlung zu verfeinern. In Smyth et al. (2004) wird ein System zur Empfehlung von Digitalkameras präsentiert. Der Recommender startet mit einem initialen Produktvorschlag, dessen Attribute über eine Benutzeroberfläche adaptiert werden können (siehe Abbildung 2.6). Die Attribute der Kameras wie Preis, Speicher, Gewicht, können mit den Buttons rechts neben der Bezeichnung der Eigenschaft nach oben bzw. nach unten korrigiert werden. Nach der Anpassung des Benutzers, berechnet das System erneut optimale Produkte zu den Anforderungen des Benutzers. Das Feedback des Anwenders kann im Kontext einer Digitalkamera unterschiedlicher Art sein, beispielsweise das Erhöhen bzw. Senken einer Einheit. Bezüglich der Speicherkapazität kann der Benutzer ”mehr Speicher” oder ”weniger Speicher” fordern. ”Mehr Speicher” ohne konkrete Einheit, bedeutet für das System eine Erhöhung des Speichers um eine Einheit bzw. die nächste verfügbare Größe einer Speicherkarte. Ein *Case-based Recommender* kann auch mit einem Lagermanagement verknüpft werden, wodurch nur noch Produktkonfiguration empfohlen werden, welche verfügbar sind. Über eine Benutzeroberfläche werden die entsprechenden Buttons zur Verfügung gestellt: beispielsweise weniger Leistung oder mehr Festplattenkapazität.

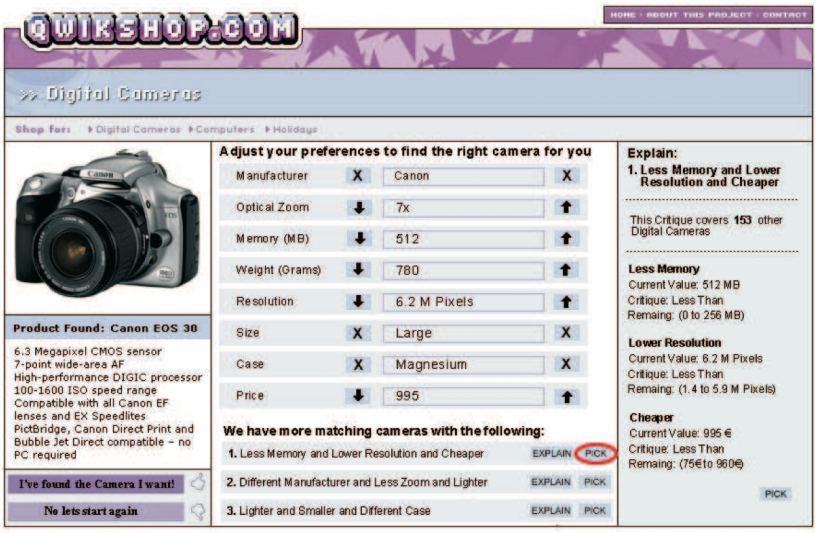

Abbildung 2.4.: Ein Beispiel für ein Case-based Recommender System aus Smyth et al. (2004)

Von Mirzadeh et al. (2005) wurde ein wissensbasiertes System für Reisevorschläge entwickelt. Dieses arbeitet ebenfalls *case-based*. Vom Benutzer wird eine initiale Query erstellt und mit zusätzlichen Queries ergänzt, um die Ergebnisse einzuschränken.

Constraint Based Recommender

Constraint-based Recommendation Systeme verwenden die Darstellungsweise von CSPs (siehe 2.1.2) damit ein Problem von *Constraint Solvern* wie zum Beispiel *JCOP*[§] gelöst werden kann. Die Erstellung eines solchen Recommenders erfordert meist hohen Aufwand, da Domänenexperten benötigt werden, die umfangreiches Wissen über die Eigenschaften der Produkte als auch über den Blickwinkel der Benutzer besitzen.
Zur Erstellung eines *Constaint-based Recommender* werden Domänenexperten benötigt, welche die Fähigkeit besitzen, die Beziehung zwischen explizit definierte Fragen und Produkteigenschaften herzustellen. Diese müssen mittels Constraints modelliert werden. Constraint Systeme bieten zusätzlich den Vorteil, falls keine Produkte gefunden wurden, dass die Kundenanforderungen repariert und automatische Erklärungen zur Verfügung gestellt werden können (siehe Felfernig and Burke, 2008).
Die modellierte Beziehung zwischen Produkten und Anforderungen wird als Wissensbasis eines Recommenders bezeichnet, im Sinne eines CSP gibt es dafür zwei Arten von Variablen (U, P) wobei U die Benutzer Variablen repräsentieren, P die Produktvariablen (siehe Felfernig and Burke, 2008). Das folgende Beispiel zeigt einen Recommender für Fahrräder:

$U =$

[§]http://www.jacop.eu/

$u_1 : streckenprofil(street, offroad)$,
$u_2 : transport(ja, nein)$,
$u_3 : downhill(ja, nein)$,
$u_4 : sport(ja, nein)$,
$u_5 : alter(integer)$,
$u_6 : max_preis(integer)$,

$P =$
$p_1 : name(text)$,
$p_2 : fullsuspension(ja, nein)$,
$p_3 : material(carbon, aluminium)$,
$p_4 :$ gepäckträger$(ja, nein)$,
$p_5 : type(kids, e-bike, mountain\_lady, mountain\_race)$,
$p_6 : preis(int)$

Die Produkt Constraints ($prod_i \in PROD$ siehe Tabelle 2.7) schränken die Instanziierungen der Variablen $p_i \in P$ ein und bestimmen damit die Eigenschaften der Produkte im Detail. Die Produkt Constraints sind disjunktiv verknüpft und bilden gemeinsam den *PROD* Constraint ($prod\_1$ OR $prod\_2$ OR $prod\_3$).

id	name	type	prod_fullsus	prod_material	prod_gepäcktr	prod_alter_min
prod_0	eCOOL	e-bike	ja	aluminium	nein	10
prod_1	eNICE	e-bike	nein	aluminum	ja	10
prod_2	RACE-cool	mount_race	ja	carbon	nein	10
prod_3	KID-BIKE	kids	nein	aluminium	ja	0

Tabelle 2.7.: Die Tabelle mit den Produkteigenschaften für einen Bike Recommender

Die *(in)Compatibility Constraints* ($comp_i \in COMP$) schränken die Anforderungen ein. Ein Beispiel dafür ist: Ein "downhill" Bike eignet sich nur für "offroad" Strecken. *Compatibility Constraints* stellen die Konsistenz der Anforderungen des Benutzers sicher.

id	constraint
comp_1	*knd_downhill* benötigt *knd_streckenprofil(offroad)*
comp_2	*knd_einsatzbereich(einkaufen)* benötigt *knd_race(nein)*

Tabelle 2.8.: Beispiele für *Compatibility Constraints eines einen Bike Recommender*

Filter Constraints ($filt_i \in FILT$) beschreibenden Zusammenhang zwischen Kundenanforderungen und der Produkteigenschaften. Einem Kunden, der ein Rad möchte, mit dem er etwas transportieren

kann, werden nur Räder mit Gepäckträger vorgeschlagen (siehe Tabelle 2.9) Ein weiterer Filter Constraint ist: Der Preis der vorgeschlagenen Produkte muss geringer oder gleich dem Betrag sein, den der Benutzer maximal ausgeben möchte.

id	constraint
filt_1	*knd_transport* benötigt *prod_gebäcktr*
filt_2	$knd\_max\_preis \leq prod\_preis$
filt_3	$knd\_sport \leq prod\_material(carbon)$

Tabelle 2.9.: Beispiele für *Filter Constraints* eines einen Bike Recommender

Zum Berechnen einer Empfehlung werden C und P als endliche Mengen von Variablen betrachtet, zusammen mit den gegebenen Mengen $PROD, COMP$ und $FILT$ von Constraints und CR, den Kundenanforderungen, kann das Berechnen einer Empfehlung als CSP betrachtet werden ($C, P, PROD \cup COMP \cup FILT \cup CR$). Die Aufgabe eine Lösung zu finden ist aus Sicht eines CSP eine Zuordnung zu den Variablen C und P zu finden, welche mit den Constraints ($PROD \cup COMP \cup FILT \cup CR$) konsistent ist. Ein Beispiel für Kundenanforderungen zeigt Tabelle 2.10. Die Konjunktion einzelnen Anforderungen (u_1 AND ... AND u_5) ergibt CR. Übergibt man das vollständige Problem $C, P, PROD \cup COMP \cup FILT \cup CR$ einem *Constraint Solver* so ergibt sich die Lösung: *RACE-cool*. Das Rad *RACE-cool*, bzw. dessen Eigenschaften stellen die einzige Lösung dar, welche bei gegebenen Kundenanforderungen keinen Konflikt mit der Wissensbasis erzeugt.

streckenprofil(u_1)	transport(u_2)	downhill(u_3)	sport(u_4)	alter(u_5)
offroad	nein	ja	ja	25

Tabelle 2.10.: Ein Beispiel für mögliche Anforderungen eines Kunden: das Ergebnis wäre das Rad "RACE-cool"

2.2.6. Intelligente Benutzeroberflächen für *Knowledge-based* Recommender Systems

Im folgenden werden einige Benutzeroberflächen, für wissensbasierte Recommender Systeme beschrieben. Die wesentlichen Aufgaben, die diese Benutzeroberflächen erfüllen müssen sind:

- Eingabemöglichkeit für Kundenanforderungen
- Darstellung des Ergebnisses (der Produkte die den Kundenanforderungen entsprechen)
- Falls keine Produkte gefunden wurden, Darstellung der Reparaturmöglichkeiten

Die NutKing Benutzeroberfläche

Mirzadeh et al. (2005) präsentierten in ihrer Arbeit einen wissensbasierten Recommender für Reisevorschläge. Die Weboberfläche bietet die Möglichkeit der Eingabe expliziter Benutzerpräferenzen. Im

Hauptbereich erfolgt die Ausgabe der Anzahl der aktuellen Ergebnisse. Falls die Benutzereingaben zu einer leeren Menge von Ergebnissen führen, berechnet das System mittels des *relax sub Moduls* des *interactive query Module* Möglichkeiten, Constraints abzuschwächen. Die Möglichkeiten dieser Abschwächung werden dem Benutzer unterhalb der Ergebnisse zur Auswahl angezeigt. Im Falle einer zu hohen Anzahl von Ergebnissen berechnet das System Verstärkungsmöglichkeiten für Constraints und präsentiert diese dem Benutzer zur Auswahl.

Abbildung 2.5.: Das Webbasierte NutKing User Interface (aus Mirzadeh et al. (2005))

Die Benutzeroberfläche des FSAdvisor

In der Benutzeroberfläche des *FSAdvisor* (Finance Service Advisor), entwickelt von Felfernig (2005a), werden sequentiell Fragen gestellt und nach Auswahl einer Antwort die Benutzeranforderungen entsprechend erweitert. Im Bereich unterhalb der Fragen werden bereits nach jeder Beantwortung provisorische Ergebnisse angezeigt, was dem Benutzer immer einen Überblick über Produktvorschläge

gibt. Im Falle inkonsistenter Anforderungen werden Fragen vorgeschlagen, deren Änderung wieder zu Lösungen, also zu Produkten führen würden. Nach Wahl einer vorgeschlagenen Frage folgt eine Auswahlmöglichkeit von Antworten, welche zu konsistenten Anforderungen führen. Zusätzlich bietet die Benutzeroberfläche Informationen zu Fragen sowie Detailinformationen zu Produkten.

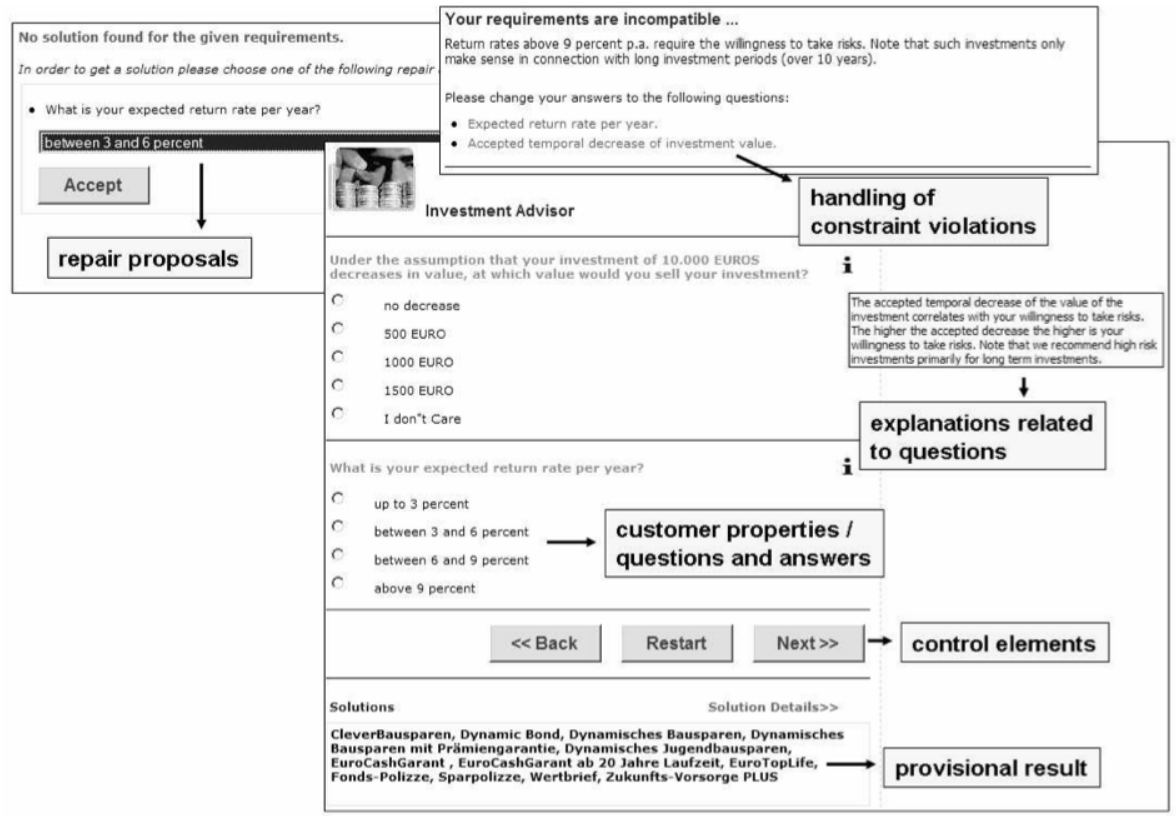

Abbildung 2.6.: Das Webbasierte User Interface des FSAdvisor (siehe Felfernig, 2005a)

Die Benutzeroberfläche von VITA

Eine weitere Anwendung im Finanzdienstleistungsbereich wird von Felfernig et al. (2007) gezeigt, die Benutzeroberfläche unterstützt ebenso, wie die Oberfläche des *FSAdvisor* eine sequenzielle Eingabe von Kundenanforderungen über die Beantwortung von Fragen, am Ende ist eine Ergebnispräsentation zu sehen (siehe Abb. 2.7). Diese wissensbasierte Applikation wir hauptsächlich im Bereich der Kundenberatung eingesetzt und bringt unter anderm große Vorteile im Vertrieb. Der wissensbasierte Ansatz bietet Reparaturen, Erklärungen bis hin zu einem MAUT (siehe Schmitt et al., 2003).

Benutzeroberflächen für mobile Geräte

Der mobile Recommender *MobiRek* (siehe Ricci and Nguyen, 2007)) ist *critique-based* und erweitert den zuvor beschriebenen *NutKing* Reise-Recommender. Die Intention von *MobiRek* ist, während einer

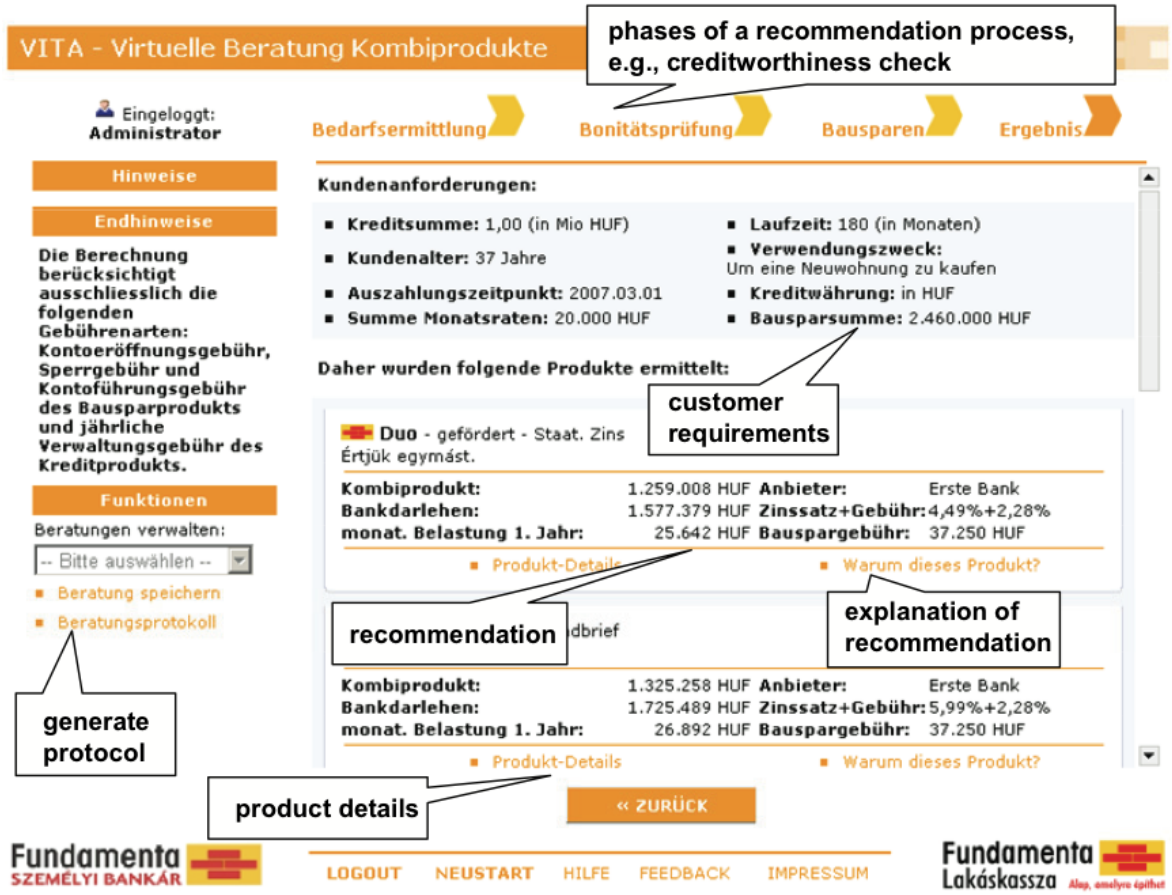

Abbildung 2.7.: (Das VITA User Interface aus Felfernig et al. (2007))

Reise über ein Mobiltelefon Attribute des Reiseplans abzuändern. *MobiRek* berechnet Empfehlungen (siehe 3.3(a)), zeigt die Eigenschaften des ausgewählten Produktes (siehe 3.3(b)) und ermöglicht dem Benutzer Kritiken einzugeben (siehe 3.3(c)).

Ricci (2011) und Ricci and Nguyen (2007) zeigen, wie Recommender Applikationen auf mobilen Geräten umgesetzt werden können. Die Technologien, welche in aktuelle Devices eingebaut werden, (z.B. GPS und weitere personalisierte Informationen) bieten eine Reihe neuer Möglichkeiten für Recommender Applikationen. Zur Empfehlung eines Restaurants im Nahen Umkreis können die Koordinaten des Benutzers herangezogen werden. Die Arbeit Ricci (2011) fokussiert auf die Bereiche Tourismus und Reisen und beschreibt die Vor- und Nachteile von mobilen Recommendern. Dem mobilen Benutzer muss die optimale Menge von Informationen zur Verfügung gestellt werden, um mit dem System interagieren zu können und dennoch muss die Applikation mit der geringen Auflösung und Bildschirmdiagonale auskommen. Gewöhnliche Webinterfaces, ausgelegt für die Bildschirmauflösung und große von Desktop PCs, produzieren beim Verwenden auf mobilen Geräten einen Informationsüberfluss.

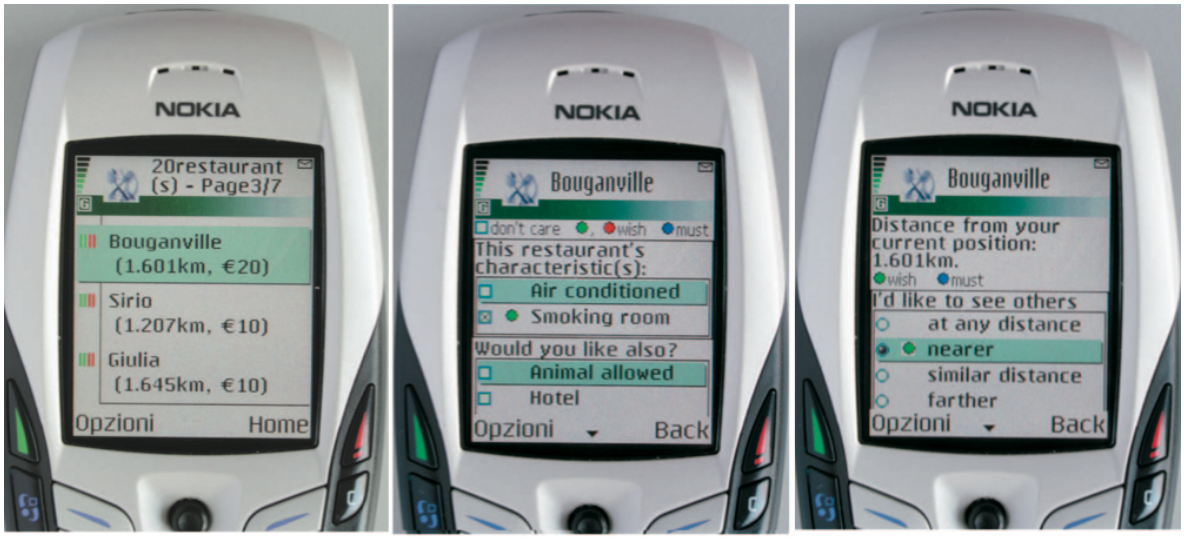

Abbildung 2.8.: Das MobiRek Userinterface. (a) Empfohlene Produkte, (b) Details der Produkte, (c) Eingabe von Krtiken (aus 3.3)

Alternative Eingabemöglichkeiten

Thompson et al. (2004) setzt auf eine unkonventionellere Variante, ohne grafische Oberfläche, der Kunde gibt die Anforderungen in natürlichsprachlicher Form ein. Wäre da nicht die Barriere der schwierigen Eingabe käme diese Methode auch dem knapp bemessenen Platz auf dem Bildschirm eines mobilen Gerätes entgegen.

2.2.7. Hybride Recommender

Hybride Recommender kombinieren in unterschiedlicher Art und Weise bereits genannte Empfehlungsalgorithmen.

Ziel ist es, die Vorteile unterschiedlicher Algorithmen zur Entwicklung besserer Algorithmen (z.B. im Sinne der Vorhersagequalität) zu nutzen. In Burke (1999) und Zanker (2008) werden *knowledge-based* Recommender *collaborative filtering* Ansatz verbunden.
Zanker (2008) kombiniert die Recommenderstrategien indem die komplexe Wissensgewinnung, die ein *Knowledge-based* Recommender zum Aufbau der Wissensbasis benötigt, durch einen *Collaborative* Ansatz erweitert wird. Dazu werden Benutzerdaten aus der Vergangenheit verwendet und mittels einem *nearest-neighbour* Algorithmus ausgewertet. Die Evaluierung des Algorithmus an Daten zeigt, dass dieser Hybrid Recommender eine bessere Vorhersagegenauigkeit besitzt, als eine von Domain Experten erzeugte Wissensbasis.
Burke (1999) gleicht das *ramp-up* Problem von CF mit dem *knowledge-based* Ansatz aus. So kann

das *collaborative filtering* auch wenn nur wenige Rating Daten zur Verfügung stehen bereits gute Empfehlungen liefern.
Eine weitere Möglichkeit für einen Hybrid Recommender wird in Ricci et al. (2006) beschrieben. Die Arbeit stellt ein Reiseplanungssystem vor, welches den *Knowledge-based* Ansatz mit einem CF Ansatz kombiniert. Der Reiseplaner empfiehlt dem Benutzer nur Services, die von anderen Benutzern positiv evaluiert wurden.

Kapitel 3

Knowledge-based Recommendation mittels *Conjunctive Queries*

3.1. Übersicht

Der Kern eines wissensbasierten Systems ist eine Wissensbasis, aufgebaut aus Produkten und Kundenanforderungen. Die Beziehung zwischen den Produkten und den Kundenanforderungen wird durch Bedingungen bzw. Einschränkungen modelliert. Für *Constraint-based Recommenders* werden diese Bedingungen in *Compatibilty Constraints* und *Filter Constraints* aufgeteilt (siehe Kapitel 2.2.5). Während des Empfehlungsprozesses kommen die Kundenanforderungen hinzu, welche die Gesamtmenge der Produkte auf die Menge der kompatiblen Produkte einschränken. Der Programmablauf von *Constraint-based* Recommendern wird durch den Prozessfluss (siehe Felfernig et al., 2006) dargestellt. Dieser schließt während des Empfehlungsprozesses aufgrund bereits eingegebener Anforderungen künftige Fragen aus. Reparaturoptionen ermöglichen es dem Benutzer in Situationen, in denen keine Lösung gefunden werden kann, seine Anforderungen einfach zu korrigieren, um Produktempfehlungen zu erhalten.
In Felfernig et al. (2006) wird eine integrierte Umgebung (*CW Advisor Designer*) beschrieben, mit der es möglich ist, Recommender für verschiedenste Anwendungsbereiche wie Digitalkameras oder Finanzdienstleistungen zu erstellen. Der *CW Advisor Designer* bietet sowohl umfangreiche grafische Tools zum Erstellen des Prozessflusses, als auch Unterstützung beim Erstellen von Wissensbasen.
Die wesentlichen Bestandteile eines *Constraint-based* Recommenders sind die folgenden:

- die **Wissensbasis:** bestehend aus den *Produkt-Constraints*, *Compatibility-Constraints* und den *Filer-Constraints*
- die **Customer Requirements:** (Kundenanforderungen), in Form von Fragen
- dem **Prozessfluss:** beschreibt den Ablauf des Programms

- das **Management widersprüchlicher Anforderungen:** im Falle von Inkonsistenz der Kundenanforderungen mit der Wissensbasis müssen Alternativen präsentiert werden

Die Darstellung von Wissensbasis und Kundenanforderungen eines *Constraint-based* Recommenders als *conjunctive Query* ermöglicht das Lösen des CSP auf einer Datenbank. Aufgrund der freien Verfügbarkeit von Datenbanken bringt dies einen deutlichen Vorteil gegenüber dem Einsatz eines *Constraint Solvers*. SQL als Darstellungssprache für Constraints bringt weiters den Vorteil, dass SQL geläufiger ist als andere Logik Programmiersprachen.

3.2. Darstellung eines Empfehlungsproblems als konjunktive Query

Für die Darstellung eines wissensbasierten Empfehlungsproblems als *conjunctive Query* müssen alle Teile der Problembeschreibung in SQL Syntax beschrieben werden. Zunächst muss die Problemdefinition, die Wissensbasis (C_{KB}) bestehend aus den Produkt Constraints (C_{PROD}), den *Compatibility Constraints* (C_c) und den Filter Constraints (C_f) in SQL Syntax beschreiben werden. Den zweiten dynamischen Teil stellen die *Customer Requirements* C_R dar. Zum Ausführen des Statements auf der Datenbank müssen schließlich noch die verwendeten Variablen als Datenbanktabellen und deren Domänen durch Inhalte der Datenbanktabellen repräsentiert werden (siehe Felfernig and Burke (2008)).

- V_C stellt eine Menge von Variablen dar, welche die Kundenanforderungen repräsentieren,
- V_{PROD} stellt eine Menge von Variablen dar, welche die Produkteigenschaften beschreiben,
- C_{PROD} stellt die Menge von Constraints dar, welche die Produkte beschreiben,
- C_R stellt eine Menge von Constraints dar, welche mögliche Kombinationen von Kundenanforderungen darstellen,
- C_F stellt jene Menge von Constraints dar, welche die Kundenanforderungen mit den Produkteigenschaften in Beziehung setzen,
- C_c stellt eine Menge von unären Constraints dar, welche konkrete Kundenanforderungen beschreiben

Die Definition eines Produktes, in diesem Fall eines Fahrrades innerhalb einer *conjunctive Query* erfolgt durch Konjunktionen von Eigenschaften des Produktes. Die konjugierten Constraints der Produkteigenschaften werden disjunktiv verknüpft und ergeben einen gesamten Produkt Constraint. Listing 3.1 zeigt den Aufbau eines solchen Produkt Constraints für die beiden Fahrräder *eCOOL* und *eNICE* aus Tabelle 2.8.

Listing 3.1: Aufbau einer *conjunctive Query* für Fahrräder

```
1 (
2 (prod_name.v = 'eCOOL' AND prod_type.v = 'e-bike' AND prod_ fullsuspension.v='
      ja'       AND prod_price.v = 8888 AND prod_transport.v ='nein' AND
      prod_min_age.v = 10)
3 OR
4 (prod_name.v = 'e-NICE' AND prod_type.v = 'e-bike' AND prod_ fullsuspension.v=
      'nein'  AND prod_price.v = 2449 AND prod_transport.v ='nein' AND
      prod_min_age.v = 10)
5 OR
6  ... OR additional Products OR ....
7 )
```

Die *Compatibility Constraints* und *Filter Constraints* werden durch Implikation von Produkteigenschaften mit Kundeneigenschaften dargestellt. Ein natürlichsprachliches Beispiel für ein *Compatibility Constraint*: Um mit einem Rad etwas transportieren zu können, wird ein Gepäckträger benötigt. Als *Compatibility Constraint* formuliert bedeutet dies: ”Transport” impliziert ”Gepäckträger” (vgl. Listing 3.2 Z3). Diese Einschränkungen werden konjunktiv mit den zuvor definierten *Product Constraint* verbunden. Da SQL keine Implikationen versteht, müssen diese in ”not”, ”OR” Statements umgewandelt werden. Listing 3.2 zeigt die Bike Constraints aus Kapitel 2.2.5.

Listing 3.2: Aufbau der *Filter / Compatibility Constraints* einer *conjunctive Query* für Fahrräder

```
1 ... Product Constraint ...
2 AND
3 (not(transport) OR (gebäckträger))
4 AND
5 (not(kinderbike) OR (alter<10))
6 AND
7  ... AND ... additional Compatibility / Filter Constraints ... AND ...
```

Während der Kunde mit dem Recommender interagiert, wird die Wissensbasis um Kundenanforderungen (C_R) erweitert. Ein Beispiel für eine Kundenanforderung ist: Der Kunde möchte das Fahrrad im Gelände benutzen. Der zugehörige Constraint sieht wie folgt aus: ”streckenprofil = offroad” (vgl. Listing 3.3 Z5). Die Wissensbasis wird um die eingebenen Kundenanforderungen erweitert (siehe 3.3).

Listing 3.3: Beispiel für das Hinzufügen von Kundenanforderungen zur *conjunctive Query* für Fahrräder

```
1 ... Product Constraint ...
2 AND
```

```
3  ...  Compatibility / Filter Constraints ...
4  AND
5  streckenprofil = offroad
6  AND
7  transport = nein
8  AND
9  downhill= ja
10 AND
11 sport = ja
12 AND
13 knd_alter_genau = 25
```

Listing 3.4 zeigt den Aufbau einer fertigen *conjunctive Query* für einen wissensbasierten Recommender. Selektiert werden die Namen der Produkte ”prod_name.v” (siehe Listing 3.4 Z1). Das Statement liefert die Namen der Produkte welche zu den gegebenen Kundenanforderungen passen.

Listing 3.4: Aufbau einer vollständigen *conjunctive Query* für Fahrräder

```
1  SELECT prod_name.v FROM ALLTABLES
2  WHERE
3  /* Product Constraint */
4  (
5  (prod_name.v = 'eCOOL' AND prod_type.v = 'e-bike' AND prod_ fullsuspension.v='
       ja'  AND prod_price.v = 8888 AND prod_transport.v ='nein' AND prod_min_age
       .v = 10)
6  OR
7  (prod_name.v = 'e-NICE' AND prod_type.v = 'e-bike' AND prod_ fullsuspension.v=
       'nein'  AND prod_price.v = 2449 AND prod_transport.v ='nein' AND
       prod_min_age.v = 10)
8  )
9  AND
10 /* Compatibility / Filter Constraints */
11 (not(transport) OR (gebäcksträger))
12 AND
13 (not(kinderbike) OR (alter<10))
14 AND
15 /* Customer Requirements */
16 streckenprofil = offroad
17 AND
18 transport = nein
19 AND
20 downhill= ja
21 AND
22 sport = ja
23 AND
24 knd_alter_genau = 25
```

3.3. Der Empfehlungsprozess

Die Interaktion mit dem Recommender erfolgt über die Ausgabe von Fragen und deren Beantwortung durch den Benutzer. Eine solche Frage mit Antwort wird im wissensbasierten Recommender durch

einen Constraint repräsentiert. Zur Erstellung eines solchen Systems sind Domänenexperten notwendig, welche mit den Eigenschaften der Produkte und den Fragen, die den Benutzern gestellt werden, vertraut sind. Die Domänenexperten modellieren auch die Beziehung zwischen den Kundenanforderungen und den Eigenschaften der Produkte.
In einem Empfehlungsprozess kann zusätzlich zu den in der Wissensbasis festgelegten Beziehungen auch ein Ablauf festgelegt werden, der die Beziehung der Kundenanforderungen zueinander bestimmt.

In Felfernig and Shchekotykhin (2006) wurde der Aufbau des Prozesses mittels des ”predicate augmented finite state recognizer” (PFSR) modelliert. Im PFSR werden die Variablen im Constraint System durch Zustände im Graphen repräsentiert, die Übergänge durch Constraints (Belegungen der Variablen). Der *CW Advisor Designer* übersetzt diesen Prozessgraphen direkt in einen ausführbaren Recommender.

Als Beispiel zeigt Abb. 3.1 den Programmablauf für einen Bike Recommender. Der Ablauf legt fest, dass zunächst nach dem Alter (*knd_alter*) gefragt wird. Anschließend wird nach Preisvorstellung des Kunden (*knd_max_preis*) gefragt und danach ob das Rad zu Transportzwecken (*knd_transport*) verwendet werden soll. Wird das Rad nicht zum Transport von Gegenständen genutzt, folgt die Frage, ob das Bike für den Radsport (*knd_sport*) eingesetzt werden soll. Da ein Sportrad sowohl im Gelände als auch auf der Straße eigesetzt werden kann folgt die Frage nach dem Streckenprofil (*knd_streckenprofil*). Falls der Kunde ein ”street” Bike möchte gelangt er mit dieser Antwort zum Ergebnis. Da ein offroad Bike auf downhill Strecken eingesetzt werden kann oder nicht, wird nach Beantwortung mit ”offroad” schließlich nach der ”downhill” Fähigkeit gefragt. Nach Beantwortung der ”downhill” Frage (*knd_downhill*) werden dem Kunden Produkte als Resultat vorgeschlagen.

3.4. Management widersprüchlicher Anforderungen

Sowohl während der Erstellung, als auch während der Verwendung von *Constraint-based* Recommendern oder Konfiguratoren können Inkonsistenzen auftreten (siehe Felfernig and Schubert, 2010; Felfernig et al., 2009; O’Sullivan et al., 2007). Mit steigender Menge und Komplexität von Produkten wächst auch die Komplexität der Wissensbasis, um die Produkte und deren Beziehung zu den Kundenanforderungen darzustellen. Schwieriger wird auch die Wartung und Weiterentwicklung solcher Wissensbasen. Diagnosealgorithmen helfen Entwicklern komplexer Recommender und Konfiguratoren beim Testen von Wissensbasen. Durch den Einsatz von Testfällen können Konflikten aufgedeckt und Diagnosen erstellt werden. Diagnostizieren von Wissensbasen mit Testfällen ist unter anderem auch äußerst wichtig für Wissensbasen die häufigen Änderungen unterliegen, wie es im Finanzdienstleistungssektor (siehe Felfernig, 2005a), oder für jegliche Art von Produktkonfigurator (z.B. PC Konfigurator) der Fall ist.
Diagnosealgorithmen werden jedoch nicht nur zum Testen von Wissensbasen verwendet, auch während

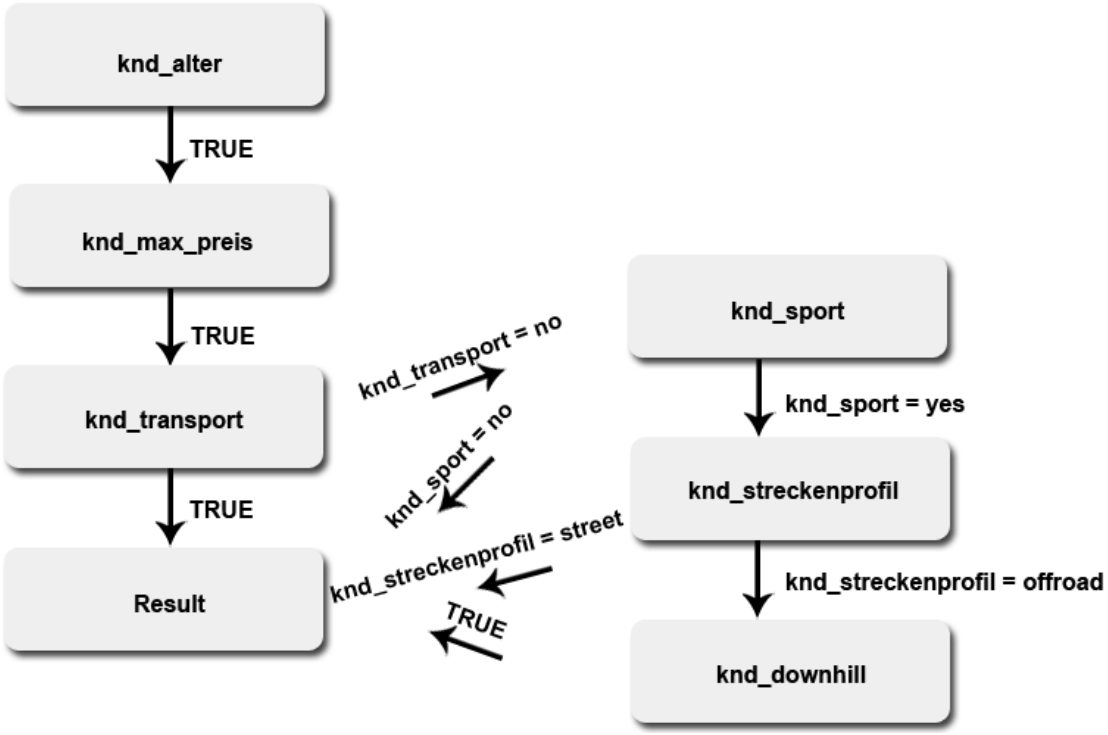

Abbildung 3.1.: Der Prozessfluss eines Bike Recommenders

der Interaktion mit *Constraint-basierten* Recommendern werden sie angewandt, um Widersprüche von Kundenanforderungen mit der Wissensbasis zu finden. Dieses Problem wird im Kontext eines *Knowledge-based* Recommenders als *customer requirements diagnosis problem* bezeichnet (siehe Felfernig et al., 2004).
Während der Interaktion mit der Benutzeroberfläche müssen Diagnosen oft *on-the-fly* generiert werden. Um allgemein gültigen *usability* Anforderungen gerecht zu werden, müssen Grenzen für die Reaktionszeiten eines User Interfaces basierend auf Empfehlungen, wie zum Beispiel von (Card et al., 1991) publiziert gewährleistet sein. Daher ist es äußerst wichtig einen entsprechend schnellen Diagnosealgorithmus zur Erstellung der Diagnosen und in weiterer Folge zum Erzeugen der Reparaturen zu wählen.

Eine *minimale* Diagnose bedeutet, eine minimale Menge von Constraints in den Kundenanforderungen zu finden, nach deren Entfernung aus den Kundenanforderungen die Gesamtheit aus Wissensbasis und den um die Diagnose reduzierten Kundenanforderungen wieder konsistent ist. Ein verbreiteter Ansatz, minimale Diagnosen zu identifizieren ist die Kombination eines *Conflict Detection* Algorithmus mit einem *Hittingset* Algorithmus (siehe de Kleer et al., 1992)). Dazu wird zum Beispiel *QuickXplain* (siehe Junker, 2004) verwendet, um die minimalen Konflikte zu identifizieren. Das Lösen der Konflikte erfolgt durch Konstruktion des *hitting set directed acyclic graph* (siehe Reiter, 1987). Die Lösung der minimalen Konfliktmengen entspricht der minimalen Diagnose.
Der Diagnose Algorithmus *FastDiag* (siehe Felfernig and Schubert, 2010)) liefert in einem Aufruf ge-

nau eine minimale Diagnose und benötigt nur eine logarithmische Anzahl von Konsistenzüberprüfungen. Da sowohl *FastDiag* als auch *QuickXplain* nach dem *Divide and Conquer* Prinzip vorgehen, besitzen deren *worst-case* Konsistenzchecks logarithmische obere Schranken. Der Vorteil von *FastDiag* liegt darin, dass dieser die fertigen Diagnosen berechnet, während *QuickXplain* zusätzlich zur Berechnung der Konflikte noch den HSDAG Algorithmus für die Diagnose benötigt.
Ein weiterer Vorteil von FastDiag ist die Fähigkeit, eine präferierte Diagnose erstellen zu können. Diese hängt von der Reihenfolge der Input Constraints ab.

3.4.1. Der Algorithmus Fastdiag

Die Operationen von FastDiag sind im Detail in Listing 3.2 beschrieben. Es wird davon ausgegangen, dass C_{KB} konsistent, die Menge $AC = C_{KB} \cup C_R$ inkonsistent ist. Das heißt, C_R, also die Kundenanforderungen beinhalten mindestens eine minimale Diagnose. Nach *Divide and Conquer* Strategie wird C_R in 2 Mengen geteilt z.B. $C_{R1} = c_4, c_5$ und $C_{R2} = c_6, c_7, c_8$. Darauf folgt nun die Konsistenzüberprüfung: Ist $AC - C_{R1}$ konsistent, liegt die Diagnose in C_{R1} und C_{R2} muss nicht mehr betrachtet werden. Rekursiv werden die beiden Mengen geteilt, bis die minimale Diagnose gefunden wurde.

Algorithm 1 – FastDiag

```
1  func FastDiag(C ⊆ AC, AC = {c1..ct}) : Δ
2  if isEmpty(C) or inconsistent(AC − C) return ∅
3  else return FD(∅, C, AC);

4  func FD(D, C = {c1..cq}, AC) : diagnosis Δ
5  if D ≠ 0 and consistent(AC) return ∅;
6  if singleton(C) return C;
7  k = q/2;
8  C1 = {c1..ck}; C2 = {ck+1..cq};
9  D1 = FD(C1, C2, AC − C1);
10 D2 = FD(D1, C1, AC − D1);
11 return(D1 ∪ D2);
```

Abbildung 3.2.: Der FastDiag Algorithmus (vgl. Felfernig and Schubert, 2010)

3.4.2. Beispiel: Kundenanforderungen für den Fahrradrecommender diagnostizieren

Das gegebene Diagnoseproblem besteht aus einem Tupel (C_{KB}, C_R) - zum einen aus der Wissensbasis C_{KB}, zum anderen aus den Kundenanforderungen C_R. Eine Diagnose zu finden bedeutet jene Constraints in C_R zu finden, welche gelöscht oder geändert werden müssen, damit Konsistenzz mit der Wissensbasis hergestellt werden kann. Die Menge der Constraints der Diagnose sind ein Teil der Benutzeranforderungen ($\Delta \subseteq C_R$) . Die Minimalität einer Diagnose ist gewährleistet, wenn es keine

Diagnose Δ' gibt für die gilt $\Delta' \subset \Delta$. Werden die Constraints der Benutzeranforderungen C_R um jene der Diagnose Δ reduziert, so ist das gegebene Problem wieder konsistent $(C_{KB} \cup (C_R - \Delta))$.

Angenommen für die gegebene Wissensbasis des Fahrradrecommenders gibt es folgende Kundenanforderungen: $c_1 = (knd\_streckenprofil = offroad)$, $c_2 = (knd\_transport = ja)$ und $c_3 = (knd\_sport = ja)$. Der Recommender kann kein Ergebnis liefern, da ein Konflikt mit der Wissensbasis besteht.

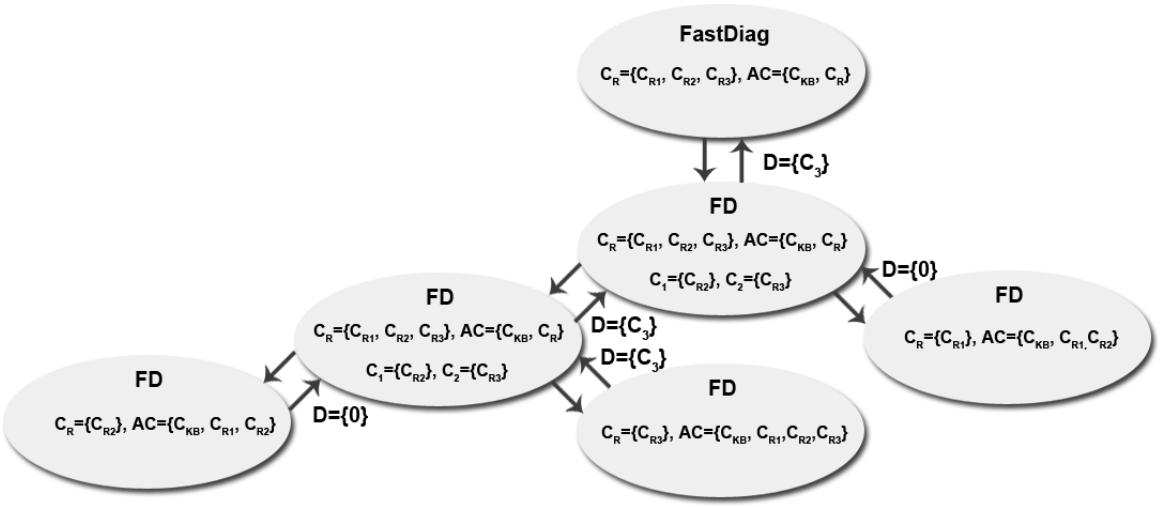

Abbildung 3.3.: Execution Trace des *FastDiag* Algorithmus mit Wissensbasis der Fahrräder

3.4.3. $n > 1$ Diagnosen berechnen

Um alle Diagnosen zu berechnen sind mehrere Aufrufe von *FastDiag* notwendig. Der Suchbaum wird dazu nach dem Prinzip der Breitensuche ausgedehnt. Ein Element der ersten Diagnose wird dazu aus der Menge C_R entfernt. Der erste Aufruf des FastDiag Algorithmus erfolgt mit den Kundenanforderungen (CR) und der Wissensbasis (C_{KB}). Ein Beispiel mit einer ersten Diagnose $\Delta_1 = c_1, c_5$ auf der ersten Ebene der Breitensuche, deren Ergebnisse $Delta_2$ und $Delta_3$, sind:
$\Delta_1 = c_1, c_5 \longleftarrow FastDiag(C_R, C_{KB})$
$\Delta_2 = c_5, c_2 \longleftarrow FastDiag(C_R - c_1, C_{KB} + c_1)$
$\Delta_3 = c_1, c_2, c_8 \longleftarrow FastDiag(C_R - c_5, C_{KB} + c_5)$

Kapitel 4

Das AdviSense Framework

4.1. Übersicht

Dieses Kapitel beschreibt das Design der Benutzeroberfläche, sowie die Architektur des *AdviSense* Framework. Das *AdviSense* Framework ist für den einfachen, kostengünstigen Entwurf von Recommendern für Touch Devices konzipiert. Der erste Prototyp wurde unter Objektive-C für das Apple iPad entwickelt, danach folgte die Entwicklung eines Prototyps für Android, um das Framework für einen größtmöglichen Marktanteil der Tablet Devices zur Verfügung stellen zu können. Den Kern der Mehrplattformfähigkeit des Frameworks stellen SQLite* Datenbanken dar, welche sowohl unter Android, als auch unter iOS lokal, ohne Datenbank Server verwendet werden können. SQLite Datenbanken bieten die optimale Basis, um die Konfigurationen, also Oberflächenparameter der Recommender und die Constraints für den Empfehlungsprozess zu speichern. Die folgenden Eckpunkte werden in diesem Kapitel beschrieben:

- Erklärung des User Interfaces
- Komponenten der Architektur
- Implementierung des *FastDiag* Algorithmus in Objective C
- Generierung mehrerer Reparaturen
- Entwicklung von Vorbedingungen auf Basis des Empfehlungsprozesses
- Abstraktion der Applikation auf Datenbankebene

Im *AdviSense* Framework wurde größter Wert auf Bedienbarkeit mittels Touchscreen sowie die Übersichtlichkeit gelegt. Das Empfehlungssystem nützt das Display von Tablets optimal und stellt während des Empfehlungsprozesses sämtliche Veränderungen dar.

*http://www.sqlite.org/

4.2. Arbeitsbeispiel Freizeit-Vermögensberater

Als Arbeitsbeispiel dienen die Daten eines Freizeit-Vermögensberaters. Anhand dieses werden die Funktionalitäten des *AdviSense* Framework erklärt. In Tabelle 4.1 sind die verwendeten Produkte und deren Eigenschaften zu finden. Die Tabelle 4.2 zeigt den Fragenkatalog für den Kunden, sowie mögliche Antworten.
In Tabelle 4.1 sind die Produkte eines Freizeit-Vermögensberaters dargestellt. Die verschiedenen Finanzprodukte (prod_1 ... prod_7) besitzen Eigenschaften, wie eine Auszahlungsvariante (prod_auszahlung), eine Einzahlungsvariante (prod_einzahlung), eine minimum Einzahlungssumme. Bestimmte Produkte können nicht in Kombination mit einem Bausparvertrag verwendet werden, dafür ist die Eigenschaft prod_bausparpraemie zuständig.

Produkt / Eigenschaft	prod_name	prod_ auszahlung	prod_ bauspar praemie	prod_ einzahlung	prod_ hoehe_ min
prod_1	Flexibles Bausparen	kapital	ja	einmalig regelmäßig	0
prod_2	Super Bausparen	Übertragung	nein	einmalig regelmäßig	0
prod_3	VorsorgeBonus (6,10J.)	kapital Pension	nein	einmalig regelmäßig	0
prod_4	Bausparen ohne Prämie	kapital	nein	einmalig regelmäßig	0
prod_5	Flexible Bond (2, 4, 6 J.)	kapital	nein	einmalig	1000
prod_6	Flexible Rend	Pension	nein	einmalig	5000
prod_7	Instantpension	Pension	nein	einmalig	0

Tabelle 4.1.: Die Produkte des Freizeit-Vermögensberaters und deren Eigenschaften.

Ein Kernstück des Beraters sind die Filterbedingungen in Tabelle 4.3. Sie stellen sicher, dass Kundenanforderungen und Produkteigenschaften im Einklang stehen. Diese zu finden, ist Aufgabe der *Knowledge Engineers*. Um Filterbedingungen zu definieren, ist detailliertes Wissen über die Produktdomäne (*Means-ends* Wissen) erforderlich. Eine Benutzeranforderung muss auf eine Produkteigenschaft gemapt werden (siehe Kapitel 2.2.2).

Name	Frage	Antworten
knd_alter_genau	Wie alt sind Sie?	16 - 99 Jahre
knd_auszahlung	In welcher Form soll die Auszahlung erfolgen?	einmalig / regelmäßig
knd_bausparpraemie_genutzt	Wird die Bausparprämie bereits genutzt?	ja / nein
knd_einzahlung	Wie möchten Sie ihren Kapitalaufbau gestalten?	einmalig / regelmäßig
knd_startkapital	Bitte geben Sie an wieviel Kapital Sie investieren möchten?	0 - 15000 Euro

Tabelle 4.2.: Die Fragen und Antworten des Freizeit-Vermögensberaters

Bedingungen
filt_1: $(knd\_alter\_genau < 55) => (prod\_name <> \text{"}Sofortpension\text{"})$ Unter 55 wird "Sofortpension" nicht angeboten! filt_2: $(knd\_auszahlung = \text{"}einmalig\text{"}) => (prod\_auszahlungcontains\text{"}kapital\text{"})$ Es werden nur Produkte angeboten, bei denen am Ende der Laufzeit das ganze Kapital auf einmal ausbezahlt wird. filt_3: $(not(knd\_auszahlung.v =' regelmäßig')OR(prod\_auszahlung.v =' pension'))$ Es werden jene Produkte angeboten, bei denen am Ende der Laufzeit eine regelmäßige Pensionsauszahlung möglich ist. filt_4: $(knd\_bausparpraemie\_genutzt = \text{"}ja\text{"}) => (prod\_bausparpraemie = \text{"}nein\text{"})$ Wenn ein Kunde bereits einen Bausparvertrag hat, werden nur Produkte ohne staatliche Förderung angeboten. filt_5: $(knd\_einzahlung = \text{"}regelmäßig\text{"}) => (prod\_einzahlung = \text{"}regelmäßig\text{"})$ Es werden jene Produkte angeboten, bei denen regelmäßige Einzahlungen möglich sind. filt_6: $(knd\_startkapital = \text{"}0\text{"}) => (prod\_hoehe\_min = \text{"}0\text{"})$ Es werden jene Produkte angeboten, die ein Startkapital unter 1000 Euro, oder keines) erfordern filt_7: $(not(knd\_einzahlung.v =' einmalig')OR(prod\_einzahlung.v =' einmalig'))$ Es werden jene Produkte angeboten, bei denen einmalige Einzahlungen möglich sind.

Tabelle 4.3.: Die Filter des Freizeit-Vermögensberaters

Listing 4.1: Die Produkte für den Freizeit-Vermögensberater

```
1 /*prod_1:*/ ((prod_name.v = 'Flexibles Bausparen' AND prod_auszahlung.v = '
    kapital' AND prod_bausparpraemie.v = 'ja' AND (prod_einzahlung.v = '
    einmalig' OR prod_einzahlung.v = 'regelmäßig') AND prod_hoehe_min.v = '0')
     OR
2 /* prod_2:*/ (prod_name.v = 'Super Bausparen' AND prod_auszahlung.v = '
    übertragung' AND prod_bausparpraemie.v = 'nein' AND (prod_einzahlung.v = '
    einmalig' OR prod_einzahlung.v = 'regelmäßig') AND prod_hoehe_min.v = '0')
     OR
3 /* prod_3:*/ (prod_name.v = 'VorsorgeBonus (6, 10 J.)' AND (prod_auszahlung.v
    = 'kapital' OR prod_auszahlung.v = 'pension') AND prod_bausparpraemie.v =
    'nein' AND (prod_einzahlung.v = 'einmalig' OR prod_einzahlung.v = '
    regelmäßig') AND prod_hoehe_min.v = '0') OR
4 /*prod_4:*/ (prod_name.v = 'Bausparen ohne Prämie' AND prod_auszahlung.v = '
    kapital' AND prod_bausparpraemie.v = 'nein' AND (prod_einzahlung.v = '
    einmalig' OR prod_einzahlung.v = 'regelmäßig') AND prod_hoehe_min.v = '0')
     OR
5 /*prod_5:*/ (prod_name.v = 'Flexible Bond (2, 4, 6 J.)' AND prod_auszahlung.v
    = 'kapital' AND prod_bausparpraemie.v = 'nein' AND prod_einzahlung.v = '
    einmalig'  AND prod_hoehe_min.v = '1000') OR
6 /*prod_6:*/ (prod_name.v = 'Flexible Rend' AND prod_auszahlung.v = 'pension'
    AND prod_bausparpraemie.v = 'nein' AND prod_einzahlung.v = 'einmalig'  AND
     prod_hoehe_min.v = '5000') OR
7 /*prod_7:*/ (prod_name.v = 'Instantpension' AND prod_auszahlung.v = 'pension'
    AND prod_bausparpraemie.v = 'nein' AND prod_einzahlung.v = 'einmalig'  AND
     prod_hoehe_min.v = '0'))
```

Listing 4.2: Die Filter Constraints für den Freizeit-Vermögensberater

```
1 /*filt_1:*/ AND (not(knd_alter_genau.v > 55) OR not(prod_name.v = '
    Sofortpension'))
2 /*filt_2:*/ AND (not(knd_bausparpraemie_genutzt.v = 'ja') OR (
    prod_bausparpraemie.v = 'nein'))
3 /*filt_3:*/ AND (not(knd_auszahlung.v = 'einmalig') OR (prod_auszahlung.v = '
    kapital' OR prod_auszahlung.v = 'übertragung'))
4 /*filt_4:*/ AND (not(knd_auszahlung.v = 'regelmäßig') OR (prod_auszahlung.v =
    'pension'))
5 /*filt_5:*/ AND (not(knd_einzahlung.v = 'einmalig') OR (prod_einzahlung.v = '
    einmalig'))
6 /*filt_6:*/ AND (not(knd_einzahlung.v = 'regelmäßig') OR (prod_einzahlung.v =
    'regelmäßig'))
7 /*filt_7:*/ AND (not(knd_startkapital.v = '0') OR (prod_hoehe_min.v = '0'))
8 /*filt_8:*/ AND (not(knd_startkapital.v = 1000) OR (prod_hoehe_min.v = 1000))
9 /*filt_9:*/ AND (not(knd_startkapital.v = 5000) OR (prod_hoehe_min.v = 5000))
```

4.3. Der Empfehlungsprozess im *AdviSense* Framework

Verglichen mit anderen Recommendern (siehe Kapitel 2.2.6) zeigt das *AdviSense* Framework im Anfangszustand alle Produkte sowie alle Fragen. Beantwortet der Kunde eine Frage und erzeugt damit eine Anforderung, so wird überprüft, ob zu gegebener Anforderung Produkte verfügbar sind. Falls dem so ist, werden abhängig von den Vorbedingungen (siehe Abb. 4.3) Fragen ein bzw. ausgeblendet und die Liste der Produkte den Anforderungen entsprechend angepasst. Sind keine Produkte zu den

gegebenen Anforderungen des Kunden verfügbar, so erfolgt ein Vorschlag für eine Reparatur. Nach akzeptierter Reparatur erfolgt die Anzeige der entsprechenden Produkte. Abbildung 4.1 zeigt eine graphische Darstellung des Empfehlungsprozesses.

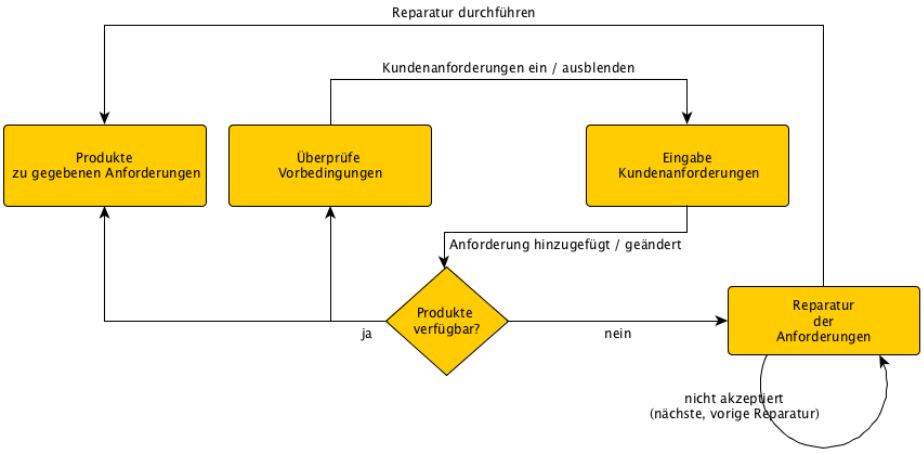

Abbildung 4.1.: Der Empfehlungsprozess des *AdviSense* Framework

Vom Prozessfluss-Graphen zu den Vorbedingungen

Der Empfehlungsprozess im *AdviSense* Framework besitzt keine Reihenfolge für die Eingabe der Benutzeranforderungen, diese werden beim Start der App vollständig in einer Liste dargestellt. Die Ergebnis- bzw. Produktdarstellung erfolgt direkt nach Eingabe einer Anforderung anstatt einer Präsentation am Ende des Empfehlungsprozesses. Eine Prozessdarstellung für den ”Freizeit” Recommender, wie sie in Felfernig (2005a) verwendet wird, zeigt Abbildung 4.2. Die Fragen werden als Knoten im Graphen dargestellt, die Kanten repräsentieren die Bedingungen zwischen den Knoten. Um diese Darstellung, wie Sie für die Zusammenhänge zwischen den Kundenanforderungen verwendet wird in *AdviSense* umzusetzen, werden die Knoten (Fragen die gestellt werden) und Kanten (Bedingungen für die Ausgabe der Fragen) im Graphen betrachtet. Für jeden Knoten wird falls notwendig ein Constraint als Vorbedingung erstellt.
Eine Vorbedingung ist eine Implikation, diese legt fest: Wenn *Frage A* mit *Antwort 1* beantwortet wurde, dann werden zum Beispiel *Frage D, Frage E* und *Frage F* ausgeblendet. Im Prozessflussgraphen werden alle Knoten (Fragen) und die Auswirkungen der Beantwortung dieser Fragen betrachtet. Soll eine Frage immer angezeigt werden, wie in Abbildung 4.2 die Fragen *knd_alter_genau*, *knd_bausparpraemiegenutzt* und *knd_einzahlung* muss dafür keine Vorbedingung erstellt werden.
Wie im Prozessfluss ersichtlich sind *knd_startkapital* und *knd_auszahlung* von der Beantwortung der Frage *knd_einzahlung* abhängig. Die beiden Fragen, in Abbildung 4.2 (1) und (2) sollen ausgeblendet werden, wenn *knd_einzahlung* mit regelmäßig beantwortet wurde (4.3 Z3). Die Frage *knd_auszahlung* ist nur von Frage *knd_startkapital* abhängig. Diese wird ausgeblendet falls das Startkapital nicht 5000

Euro beträgt. Listing 4.3 Zeile 5 zeigt dieses Statement.

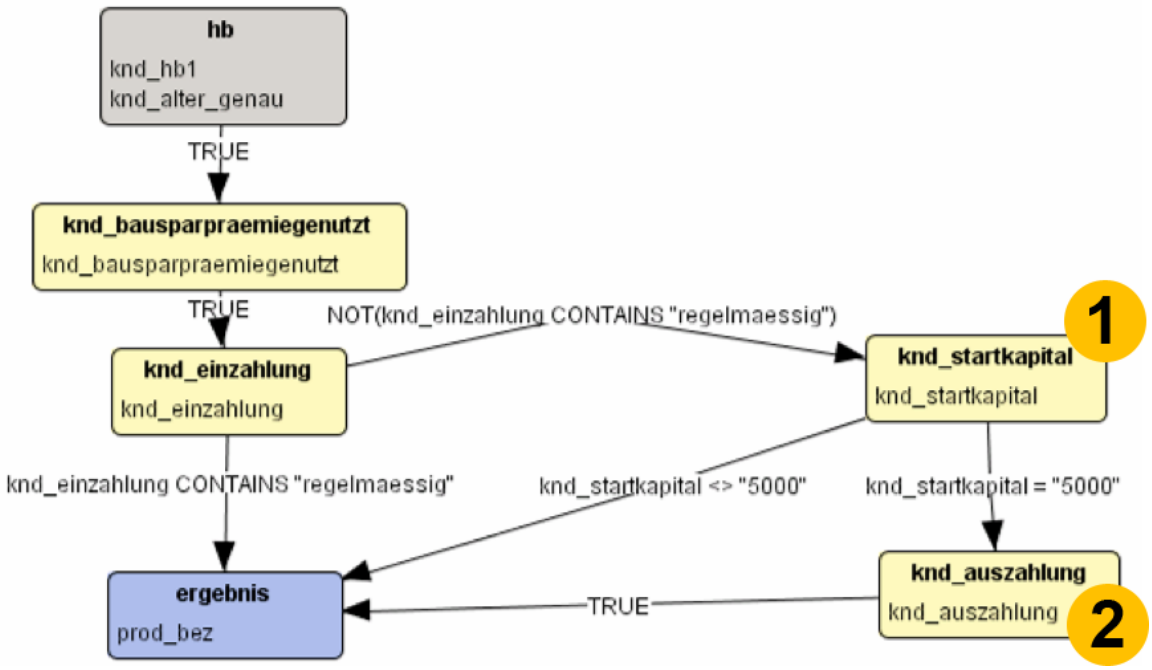

Abbildung 4.2.: Der Prozessfluss des Freizeit-Vermögensberaters: (1) Diese Frage wird nur gestellt, wenn die Kunde als Einzahlungsform „einmalig“ ausgewählt hat. (2) Diese Frage wird nur gestellt, wenn das Startkapital mehr als ”5000 Euro” beträgt.

Diese Vorbedingungen werden wiederum als *conjunctive Query* dargestellt (siehe Listing 4.3). Die Vorbedingungen für den Prozess werden in Listing 4.3 Zeile 3 und 5 gezeigt, ab Zeile 7 in diesem Beispiel werden die Kundenanforderungen als Konjunktion angehängt. Das Statement gibt die *Keys* jener Fragen zurück die zu den aktuellen Kundenanforderungen angezeigt werden sollen. Jene Fragen deren Keys nicht zurückgegeben werden, verletzen Vorbedingungen und müssen aus der Fragenliste ausgeblendet werden.

Listing 4.3: Die Vorbedingungen für die Fragen im Prozessfluss als konjunktive Datenbank Query

```
1  SELECT questions.key FROM QUESTIONTABLES
2  WHERE
3  (not(knd_einzahlung.v = 'regelmäßig')  OR (not(questions.tbl_name = '
       knd_startkapital') AND not(questions.tbl_name = 'knd_auszahlung')))
4  AND
5  (not(knd_startkapital.v <> '5000')  OR not(questions.tbl_name = '
       knd_auszahlung'))
6  AND
7  Kundenanforderung 1
8  AND
9  Kundenanforderung 2
10 AND
11 ...
```

4.4. Das User Interface

Die Benutzeroberfläche des *AdviSense* Framework lässt sich in 4 Bereiche unterteilen (siehe Abb. 4.3):

1. Die **Titelleiste** mit dem Button für die Beraterauswahl.
   Für die Titelleiste wird eine Grafik hinterlegt, welche individuell angepasst werden kann. Neben dem Bereich für das Logo, wird der Titel des aktiven Beraters ausgegeben. Das Antippen des ”Berater Auswahl” Button öffnet ein Popup mit den Icons der im System verfügbaren Berater (siehe Abb. 4.4).
2. Eine scrollbare **Liste der verfügbaren Produkte**.
   Passend zu den zu den bereits eingegebenen Kundenanforderungen (siehe 4.4.2).
3. Eine scrollbare **Liste der Benutzeranforderungen**.
4. **Banner** für *cross-selling* bzw. *up-selling* oder Werbung.
   Beim Antippen kann auf die Webseite der betreffenden Produkte verlinkt werden.

Abbildung 4.3.: Die *AdviSense* Benutzeroberfläche rechts für Android, links fürs iPad: (1) Titelleiste mit ”Berater Auswahl” , (2) Produkte, (3) Benutzeranforderungen, (4) Werbefläche

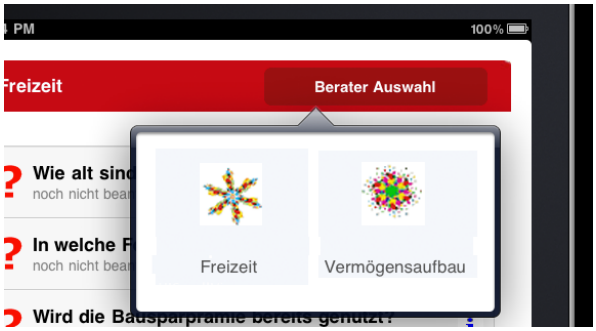

Abbildung 4.4.: Auswahl der verfügbaren Berater im System.

4.4.1. Liste der Kundenanforderungen

Beim Design der Benutzeroberfläche wurde auf effiziente und intuitive Eingabemöglichkeiten geachtet. Die Eingabe der Kundenanforderungen kann als Zahl (siehe Abb. 4.5), oder als Text (siehe Abb. 4.6) erfolgen. Nach Bestätigung der Antwort durch Antippen des ”ok” Buttons erfolgt ein Update des Systems. Der zuvor mit einem Fragezeichen versehene Eintrag, erhält einen Haken und die gegebene Antwort als Untertitel (siehe 4.6b bzw. 4.5b). Durch Drücken des ”reset” Buttons wird die Frage wieder zum Anfangszustand zurückgesetzt.

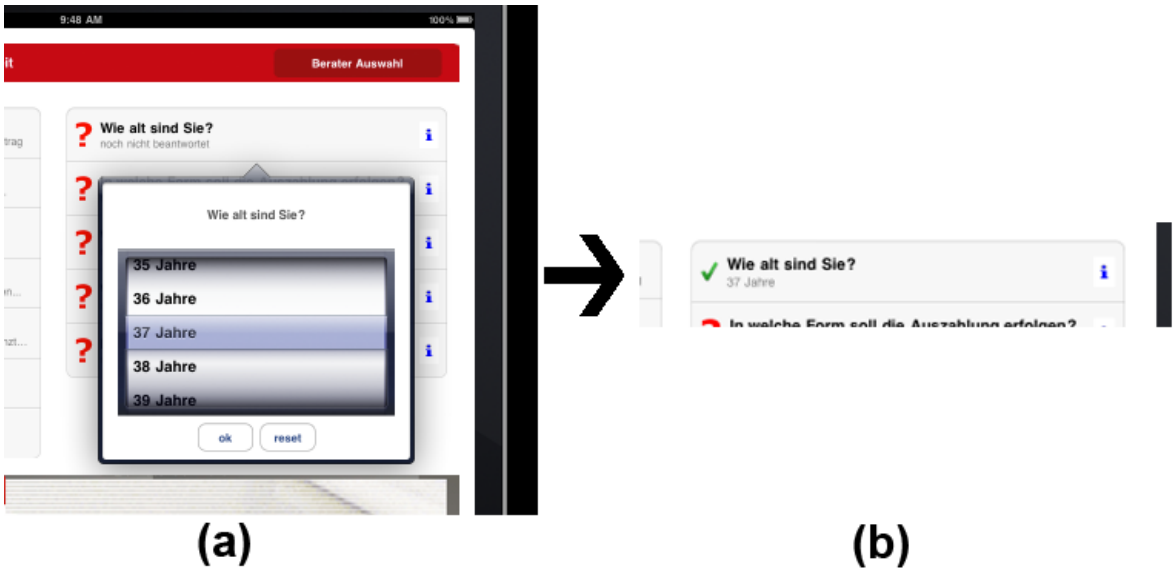

Abbildung 4.5.: (a) Auswahl eines Wertes aus einer Zahlenfolge (b) Nach Bestätigung des Wertes mit ”ok”

Die Beantwortung einer Frage mittels Textantwort (siehe Abb. 4.6) erfolgt durch Auswahl einer Antwort mittels Antippen des *Radio Button* der Antwort im Menü. Da Radio Buttons kein Standard User Interface Element sind, wurde die Open Source Bibliothek von ”iPhone UIButton Tutorial: Radio

Buttons verwendet”[†].

Abbildung 4.6.: (a) Auswahl einer Antwort (b) Nach Bestätigung der Antwort mit ”ok”

In der Android Benutzeroberfläche wurde das *AlertDialog* Element mit *SingleChoice Items* für die Darstellung der Eingabe der Kundenanforderungen verwendet (siehe Abb. 4.7). Unter Android wurde zwischen der Darstellung der Textantworten und Zahlenantworten nicht unterschieden, die Liste der *SingleChoice Items* lediglich länger und scrollbar.

Um Fragen bzw. Details zu Antwortmöglichkeiten zu erklären, gibt es die Möglichkeit, zu einer Frage zusätzlichen Informationstext zu hinterlegen, welcher dem Kunden beim Antippen des *i-Buttons* mittels eines *Popup* angezeigt wird (siehe Abb. 4.8).

[†]http://www.mobisoftinfotech.com/blog/iphone/iphone-uibutton-tutorial-radio-buttons/

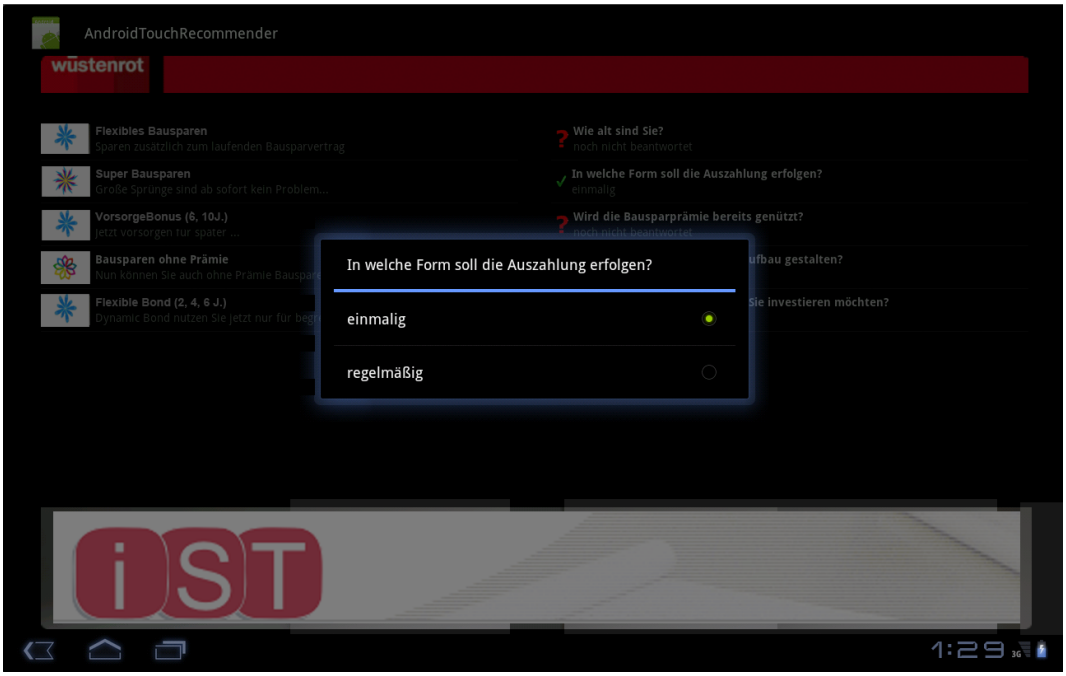

Abbildung 4.7.: Die Antwortauswahl unter Android - bestätigt wird die Auswahl durch Antippen des Bildschirms außerhalb des *Popup*

Abbildung 4.8.: Anzeige von Details zu Fragen durch *antippen* des *i-Button* einer Frage.

4.4.2. Liste der verfügbaren Produkte

Alle verfügbaren Produkte werden beim Programmstart in einer scrollbaren Liste gezeigt, die nach jeder Eingabe einer Benutzeranforderung aktualisiert wird. Falls keine Produkte zu den Anforderungen gefunden werden, wird die Produktliste durch die Anzeige der Reparaturmöglichkeiten ersetzt. Um mehr über ein Produkt zu erfahren, kann der Kunde ein Produkt aus der Liste antippen und erhält damit detaillierte Informationen (siehe Abb. 4.9).

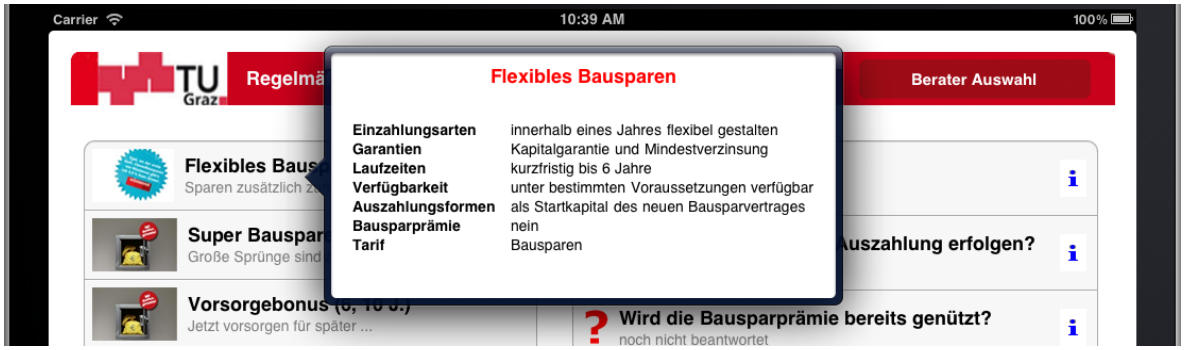

Abbildung 4.9.: Anzeige von Produktdetails durch Antippen eines Produkts.

4.4.3. Anzeige der Reparaturen

Ein wichtiges Instrument wissensbasierter Recommender ist die Erstellung von Reparaturoptionen, welche dem Benutzer auf einfache und verständliche Art und Weise Möglichkeiten aufzeigen, Anforderungen zu reparieren, zu denen keine Produkte gefunden wurden. Beim Auftreten inkonsistenter Anforderungen wird der Benutzer durch ein aufklärendes *Popup* auf die Reparatur hingewiesen (siehe Abb. 4.10). Tippen auf das Touchscreen schließt das *Popup* und die gesamte Ansicht ist wieder verfügbar (siehe Abb. 4.11).

Die visuelle Darstellung der Reparatur (Abb. 4.11 (1)) soll eine Frage symbolisieren mit der Erweiterung der Auswahlmöglichkeit einer Antwort auf diese Frage, um mindestens ein Produkt als Ergebnis zu erhalten. Die Symbolik wird dadurch verstärkt, dass die betreffende Frage aus der Fragenliste (Abb. 4.11 (2)) ausgeblendet wird. Falls weitere Reparaturen zur Verfügung stehen wird der *nächste Reparatur* Button angezeigt. Das Ergebnis nach Antippen des Buttons zeigt Abbildung 4.12. Die vorherige Reparatur wird wieder in die Fragenliste eingeblendet, die neue Reparaturoption wird aus der Liste ausgeblendet und links angezeigt. Drücken des Buttons *Änderung durchführen* (Abb. 4.12 (2)) führt dazu, dass die entsprechende Frage mit der vorgeschlagenen Antwort belegt wird. Die reparierte Frage scheint wieder in der Fragenliste auf und anstatt der Reparatur wird wieder die Produktliste angezeigt.

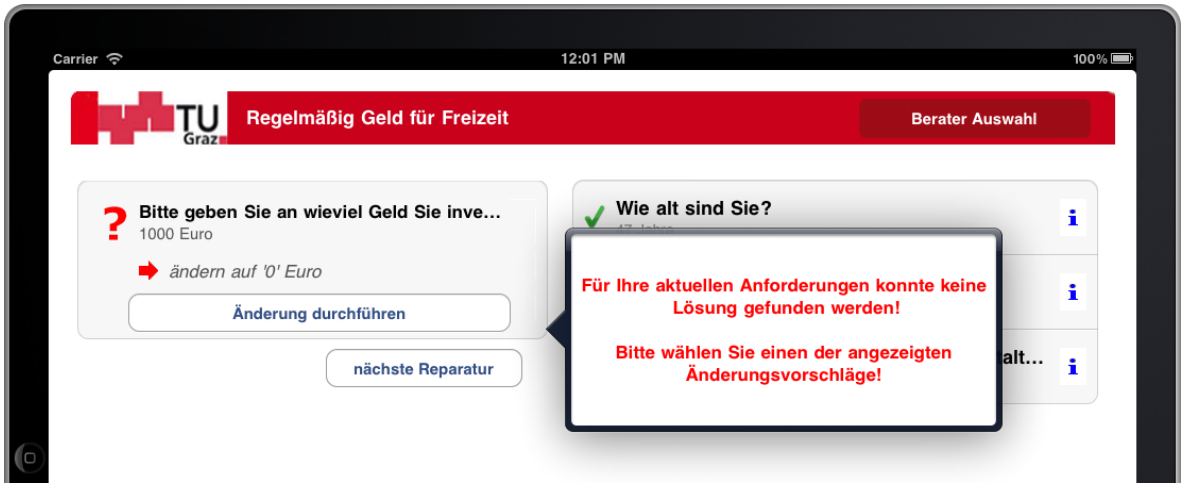

Abbildung 4.10.: Sind keine Produkte zu den eingegebenen Kundenanforderungen verfügbar, so erscheint links die erste Reparaturoption, mit einem PopUp welches darauf hinweist, dass der Benutzer nun über die Reparatur weiter navigieren muss.

Die Oberfläche des *AdviSense* Framework ermöglicht zusätzlich zu den zuvor beschriebenen Reparaturoptionen, zwei Fragen gleichzeitig zu korrigieren, diese Situation tritt auf, wenn die Minimaliagnose zwei Constraints beinhaltet (siehe Abb. 4.13(b)). Eine Reparaturerweiterung stellt auch Abbildung 4.13(a) dar: Die Auswahl einer der beiden Antworten führt zu Produktvorschlägen.

4.5. Systemarchitektur

Die Entwicklung von Prototypen für das *AdviSense* Framework erfolgte für iOS und Android. Die Entwicklungsframeworks, das *Cocoa Framework* für iOS und das Android SDK sind sehr unterschiedlich. Nicht nur im Bereich der Oberflächen der Betriebssysteme, auch die verwendeten Programmiersprachen unterscheiden sich mit *Objective C* (Cocoa) und *Java* (Android) grundlegend. Dennoch wurde darauf geachtet, dass beide Prototypen bzgl. ihrer Komponenten auf einer Linie liegen. Im folgenden Kapitel werden die Komponenten der Systemarchitektur des *AdviSense* Framework im Detail beschrieben (siehe Add. 4.14):

- **Config Komponente:** Die Konfigurations-Komponente hält in der Datenbank *config.sqlite* sämtliche Daten zur Definition der Oberfläche der Berater, sowie die Constraints der Wissensbasis in SQL Syntax, die das System definieren.
- **View Komponente** Die View Komponente hält die für das jeweilige Framework (Android / iOS) notwendigen Oberflächen Klassen, welche die Benutzeroberfläche des Systems definieren, diese sind in *Android* und *iOS* Version unterschiedlich. Beide Frameworks unterstützen jedoch das *Model View Controller* Konzept welches damit auch das Konzept sämtlicher Views in *AdviSense* darstellt.

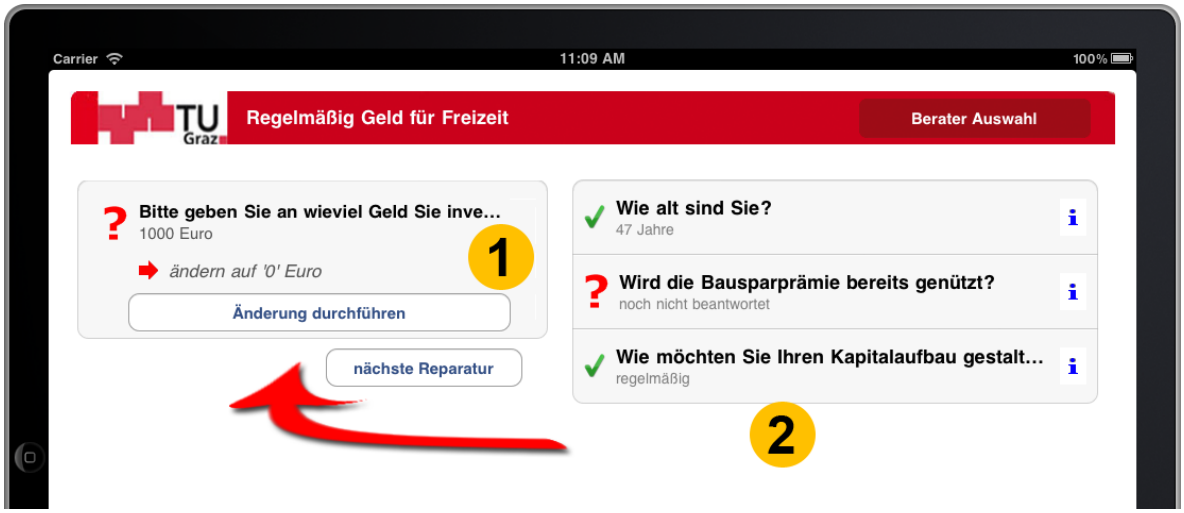

Abbildung 4.11.: Sind keine Produkte zu den eingegebenen Kundenanforderungen verfügbar, so erscheint links die erste Reparaturoption (1), die zu reparierende Frage wird rechts ausgeblendet (2).

- ***Conjunctive Query* Komponente:** Die *Conjunctive Query* Komponente ist für den Aufbau der konjunktiven Datenbank Queries, sowie deren Ausführung auf der Datenbank zuständig und liefert die entsprechenden Ergebnisse der Queries zurück.
- **Reparatur / Diagnose Komponente:** Die Reparatur-Komponente wird aufgerufen, sobald keine Produkte gefunden werden die den Kundenanforderungen entsprechen. Die Berechnung der Diagnosen mittels *FastDiag* Algorithmus sowie die Erstellung der Reparaturen sind Teil dieser Komponente.

Aus Sicht des *Model View Controller Pattern* gibt es für die Standard Recommender Ansicht einen View Part und einen Controller Part. Die einzelnen Berater Datenbanken können als Model Part gesehen werden. Auch der Reparatur / Diagnose Teil besitzt wiederum einen View Part und einen Controller Part, der Model Part wird von den Datenbanken übernommen.

4.5.1. Config Komponente

Nach Programmstart erfolgt durch die *Config Komponente* der Zugriff auf die *config.sqlite* Datenbank (Datenbank Schema siehe Abb. 4.16), aus welcher sämtliche Oberflächendaten für die Darstellung des ersten Beraters geladen werden. Die folgenden Parameter (Spalten der Tabelle *advisors*) können durch entsprechende Datenbankeinträge bestimmt werden:

- **name:** Kurzname des Beraters
- **database:** Name der Datenbank, auf welcher die *conjunctiven Queries* ausgeführt werden.
- **title** Der Titel des aktiven Beraters, der in der Titelleiste (siehe 4.3 (1)) angezeigt wird.

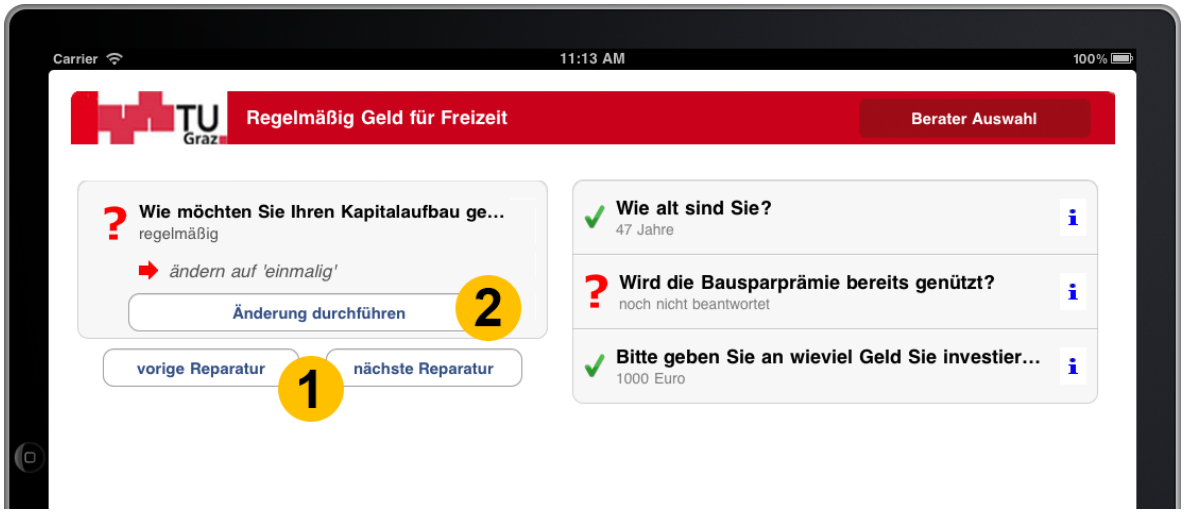

Abbildung 4.12.: Die ”vorige” und ”nächste” Buttons dienen der Navigation zwischen den verfügbaren Reparaturoptionen. Drücken des ”Änderung durchführen” Buttons führ die vorgeschlagene Reparatur durch.

Abbildung 4.13.: Die erweiterten Reparaturmöglichkeiten

- **thumbnail:** Die Grafik welche in der Beraterauswahl (siehe 4.4) angezeigt wird.
- **bgcolor:** Hintergrundfarbe RGB 0 - 255
- **type:** Art der Produkt Darstellung, ”0” große Produktbilder, ”1” nur Produkt Icons (siehe Abb. 4.4)
- **headerimg:** Dateiname des Hintergrundbildes der Titelleiste (siehe 4.3 (1))
- **footerimg:** Dateiname des Banners (siehe 4.3 (4))
- **advisorchoiceimg:** Hintergrundbild des ”Berater Auswahl” Buttons (siehe 4.3 (1))

Jeder Berater, zu dem über den ”Berater auswählen” (siehe Abb. 4.4) gewechselt werden kann, muss in die Tabelle *advisors* eingetragen werden. Beim Starten der App wird jener Berater der Tabelle *advisors* mit *key = 1* geladen. Für jeden Berater gibt es eine Detailtabelle (siehe Abb. 4.16, *advisordetail* Tabelle), in deren Spalten die folgenden Attribute des jeweiligen Beraters eingetragen sind:

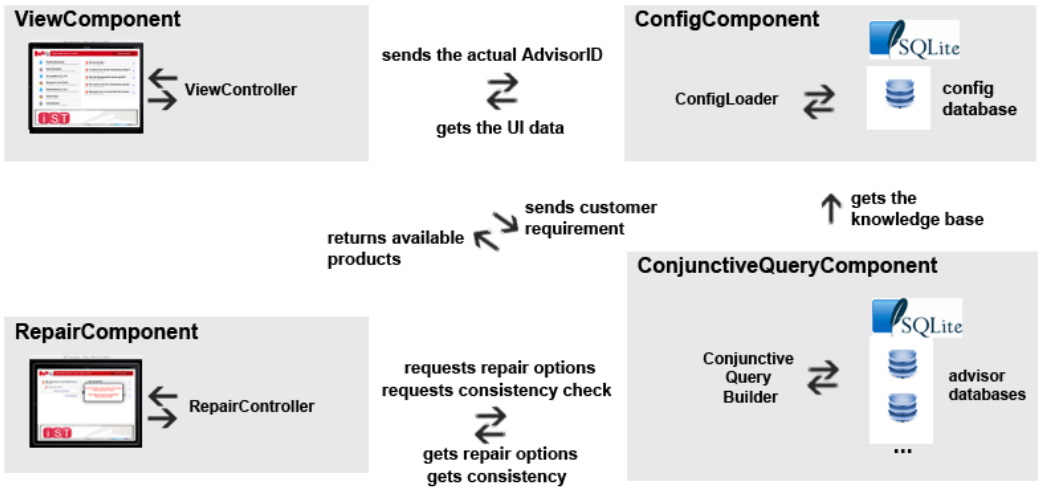

Abbildung 4.14.: Die vier Komponenten des *AdviSense* Framework (vereinfacht dargestellt)

- **products:** Die Produkt Constraints in SQL Syntax
- **filters:** Die Filter Constraints
- **preconditions:** Die Vorbedingungen für den Prozessfluss
- **product_tables:** Die Namen der Tabellen der Produkte
- **customer_tables:** Die Namen der Tabellen der Kundenanforderungen
- **repair_excluded:** Die von der Reparatur ausgenommen Kundenanforderungen

Die *advisor_detail* Tabelle enthält also sämtliche Constraints, die für den Aufbau der Wissensbasis des Recommenders notwendig sind. Die *Config Komponente* stellt der *Conjunctive Query Komponente* die Daten der Wissensbasis des aktuellen Beraters über ein Objekt zur Verfügung. Bei Beraterwechsel über das *Advisor Auswahl Menü* (siehe Abb. 4.4) werden von der *Config Komponente* die Daten des neuen Beraters geladen und der *Conjunctive Query Komponente* zur Verfügung gestellt.

4.5.2. View Komponente

Die View Komponente kommuniziert mit der *Config Komponente* zum Wechseln des aktiven Recommenders und zum Laden der Informationen bezüglich dessen Oberfläche. Da über die View Komponente die Eingabe der Kundenanforderungen erfolgt, wird von der View Komponente eine neue Kundenanforderung an die *Conjunctive Query Komponente* übermittelt. Die entsprechenden Produkte zu den bereits gegebenen Anforderungen werden retourniert.
Nachfolgend wird der Aufbau der View Komponente unter iOS und Android beschrieben:

- **iOS:** Unter iOS besteht die Standard View Komponente aus zwei Cocoa *UITableView* Elementen. Diese stellen die Oberfläche der Listen für Fragen und Produkte dar. Weiters sind

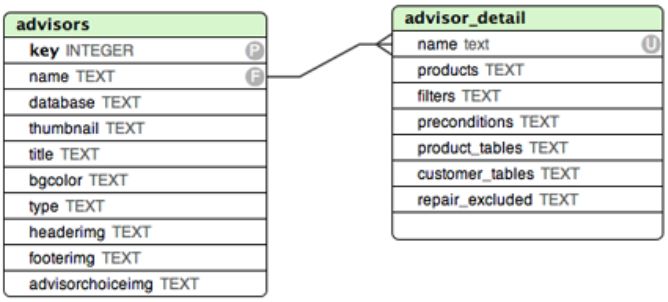

Abbildung 4.15.: Die Konfigurationsdatenbank *config.sqlite* bestehend aus der *advisors* Tabelle mit den Eigenschaften der Berateroberfläche sowie einer Detailtabelle je Berater (*advisor_detail*), zum Speichern der Constraints des Beraters.

die *Popover Controller* zum Anzeigen der Antwortmöglichkeiten, der Produktinformationen und der Informationen zu den Fragen Teil der View Komponente. Auch als Interface für den Wechsel zwischen den Recommendern im System wurde ein *Popover Controller* verwendet.

- **Android:** Die Android Oberfläche besteht ebenfalls aus den beiden Listen, Android besitzt jedoch keine mit den ”Popover Controllern” vergleichbaren View Objekte. Daher wurde hier das *AlertDialog* Element mit *SingleChoice* Items zur Auswahl der Antworten verwendet. Die Listen für Fragen und Produkte wurden mit dem *ListView* umgesetzt.

Sowohl das Android als auch das iOS Entwicklungsframework unterstützen die Verwendung des *Model View Controller* Konzeptes. Das *Model View Controller* und das Komponentenkonzept dieses Kapitels sind wie folgt integriert: Die View Komponente enthält einen View und einen Controller Teil. Das Modell, also das zur Verfügung Stellen der Daten wird von der *Conjunctive Query Komponente* übernommen.

4.5.3. Conjunctive Query Komponente

Die *Conjunctive Query Komponente* wickelt den Zusammenbau der Datenbank Queries ab. Sie benötigt Wissensbasis, Vorbedingungen und die Tabellennamen der Produkte und Fragen des aktuellen Recommenders von der *Config Komponente*. Damit ist der *Conjunctive Query Komponente* der Bau folgender Queries möglich:

- die Datenbank Queries zum Generieren der Empfehlungen für Kundenanforderungen,
- das Erzeugen der Queries für die Vorbedingungen der Fragen,
- das Erzeugen der Queries für die Reparaturoptionen,
- das Durchführen der Konsistenzüberprüfungen für den *FastDiag* Algorithmus

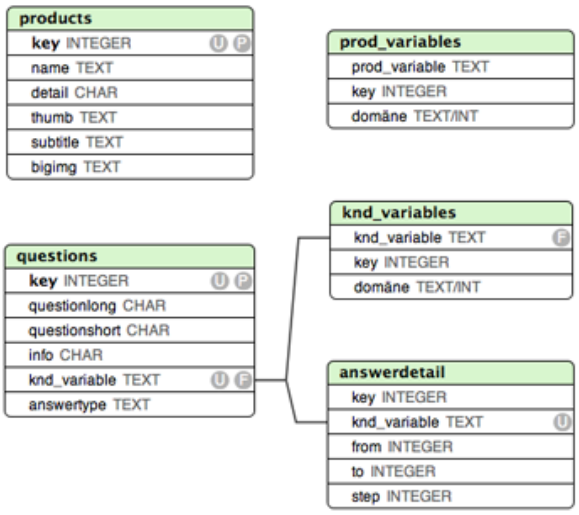

Abbildung 4.16.: Aufbau der Beraterdatenbank **advisor1.sqlite**, bestehend aus einer **products** Tabelle, einer Tabelle für jede Produkt-Variable **prod_variables**, eine Tabelle für jede Kundenanforderung **knd_variables**, die Tabelle **questions** mit Detailinformationen zu jeder Kundenanforderung und die Tabelle **answerdetail** zur Darstellung großer Zahlen Domänen im User Interface.

Nach Eingabe einer neuen Benutzeranforderung (siehe 4.6) wird das System aktualisiert. Dies passiert durch den Aufruf der *setQuestionAnswered* Methode in Listing 4.4 mit den Parametern *qid* (ID der Frage) und *aid* (ID gegebenen Antwort). Falls möglich wird die Liste der angezeigten Produkte über die View Komponente aktualisiert (4.4 Zeile 6). Wurde eine Vorbedingung für eine oder mehrere Fragen verletzt, so wird eine Nachricht an die View Komponente gesendet und die Liste der Fragen entsprechend aktualisiert. Im Falle einer Inkonsistenz zwischen Kundenanforderungen und Wissensbasis können keine Produkte angezeigt werden und es erfolgt der Aufruf der *Reparatur / Diagnose Komponente*.

Listing 4.4: Ein Frage wurde beantwortet

```
1  - (void) setQuestionAnswered:(int)qid :(int)aid
2  {
3    if(![self updateSystemStateWithAnswer:qid :aid])
4      NSLog(@"ERROR With Updating System State %i, %i",qid, aid);
5
6    if (![[[Config getConfig] getProductsData] updateProducts:crdict_])
7    {
8    // es werden keine Produkte gefunden -> Reparieren!
9
10     repair_ = [[Repair alloc] init];
11     [repair_ setQuestionData:self];
12     [repair_ createRepairOptions:crdict_];
13
14     [ViewManager showRepairOptions];
15     [ViewManager showRepairInfo];
16   }
17   else
18   {
```

```
19      [ViewManager unShowRepairOptions];
20
21      [self checkpreconditions];
22    }
23  }
```

Beispiel: Eine Kundenanforderung hinzufügen

Ausgehend von der Konfiguration in Abb. 4.6, die Frage ”Wie alt sind Sie?” wurde mit ”37 Jahre” beantwortet, die Frage ”In welcher Form soll die Auszahlung erfolgen?” steht vor der Beantwortung durch den Kunden. Mit Einloggen der Antwort ”einmalig” durch Drücken des ”ok” Buttons erfolgt im *QuestionController* der Aufruf der *setQuestionAnswered* Methode (Listing 4.4). Diese ruft die *Conjunctive Query Komponente* auf, wo die gegebene Antwort in Form der Constraint *knd_auszahlung = 'einmalig'* in die *conjunctive Query* integriert wird. Die *Conjunctive Query Komponente* führt anschließend den Datenbankaufruf mit der vollständigen Query aller Kundenanforderungen durch. Als Ergebnis der Query sind in Produkte in Abbildung 4.18 zu sehen.

Listing 4.5: Schematische Darstellung der generierten Datenbank Query mit der eingefügten neuen Constraint in Zeile 9

```
1  SELECT prod_name.v FROM ALLTABLES
2  WHERE
3  ... 'Product Constraint' ...
4  AND
5  ... 'Filter Constraints' ...
6  AND
7  knd_alter_genau = 37
8  AND
9  knd_auszahlung = 'einmalig'
```

Beispiel: Eine Kundenanforderung hinzufügen (Auslösen einer Precondition)

Eine Vorbedingung tritt ein, wenn die Frage ”Wie möchten Sie ihren Kapitalaufbau gestalten?” anstatt wie in Kapitel 4.5.3 mit ”einmalig”, mit ”regelmäßig” beantwortet wird. Nun wird ein Aufruf der *Conjunctive Query Komponente* getriggert, in welcher die gegebene Antwort als Constraint *knd_einzahlung = 'regelmäßig'* zur konjunktiven Datenbank Query hinzugefügt wird. Im Falle eines erfolgreichen Ausführens der Datenbank Query, d.h. es werden Produkte retourniert und keine Reparatur ausgelöst, erfolgt der Aufruf der Methode *checkpreconditions* (siehe Listing 4.4 Z21). Diese Methode triggert wiederum einen Aufruf innerhalb der *Conjunctive Query Komponente*, welche die Query für den Precondition Check zusammenbaut. Das Ergebnis der Query liefert jene Fragen die angezeigt werden sollen. Sie werden im *QuestionController* mit den aktuell angezeigten Fragen verglichen. Jene Fragen die eine Vorbedingung verletzen werden ausgeblendet. In diesem Fall sind dies

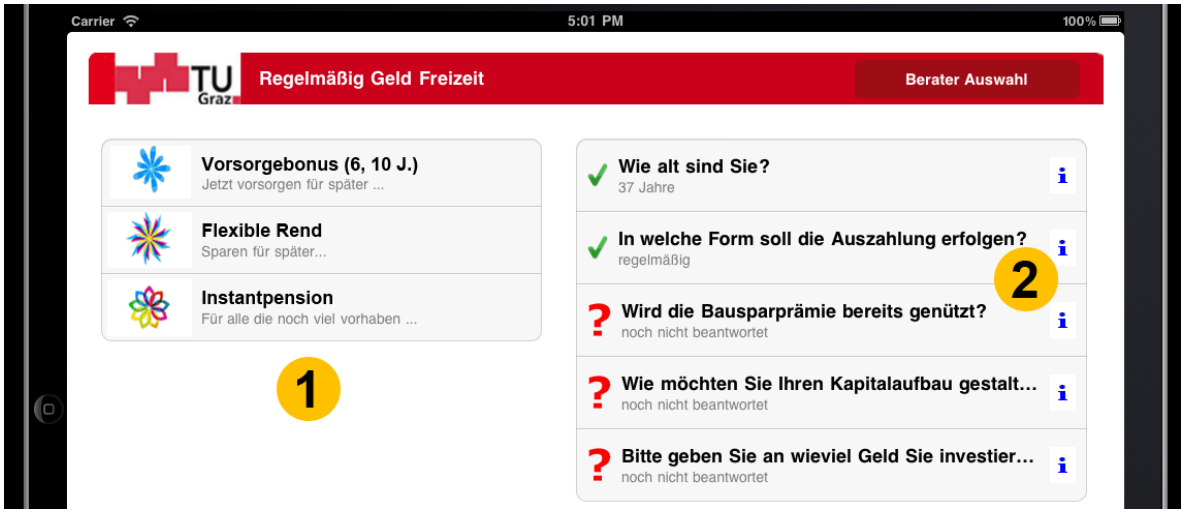

Abbildung 4.17.: Der Systemzustand nach Setzen der Kundenanforderungen *knd_alter_genau = 37* und *knd_auszahlung = 'einmalig'*, die Produkte *Super Bausparen, Flexibles Bausparen, Bausparen ohne Prämie und Flexible Bond* entsprechen nicht den Kundenanforderungen und wurden daher ausgeblendet.

die Fragen "In welcher Form soll die Auszahlung erfolgen?" und " Bitte geben Sie an wieviel Geld Sie investieren möchten! ". Das Ergebnis der Query wird nach Auswertung über den *QuestionController* der View zum Anzeigen zur Verfügung gestellt.

Listing 4.6: Schematische Darstellung der generierten Datenbank Query mit dem neu hinzugefügten Constraint *knd_einzahlung= regelmäßig*

```
1 SELECT questions.key FROM ALLQUESTIONTABLES
2 WHERE
3 ... 'PRECONDITIONS' ...
4 AND
5 ( knd_alter_genau = 37 )
6 AND
7 ( knd_einzahlung = 'regelmäßig' )
```

4.5.4. Reparatur / Diagnose Komponente

Die *Reparatur / Diagnose Komponente* wird aufgerufen, wenn die *Conjunctive Query Komponente* für die aktuellen Kundenanforderungen keine Produkte liefert - die Kundenanforderungen C_R sind mit der Wissensbasis C_{KB} inkonsistent. Ein *Repair* Objekt (siehe Listing 4.4 Z10) übernimmt nun die Erzeugung der Reparaturoptionen mittels der von *FastDiag* (siehe 3.4) berechneten Diagnosen.
Teil der Reparatur Komponente ist eine eigene View und ein Controller. Der View Teil ist für die Darstellung der Reparaturoption und der notwendigen Buttons für Navigation und "Reparatur durchführen" zuständig. Das Durchführen einer Reparatur erfolgt wie das Hinzufügen neuer Kun-

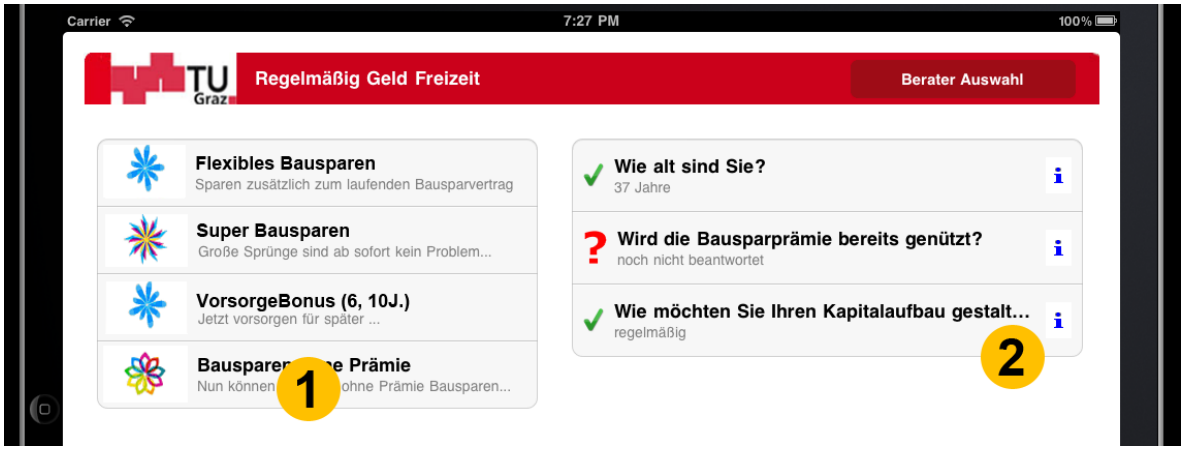

Abbildung 4.18.: Der Systemzustand nach Setzen der Kundenanforderungen *knd_alter_genau = 37* und *knd_einzahlung = 'regelmäßig'*. Die den Anforderungen ansprechenden Produkte werden in der Produktliste angezeigt. Da eine Vorbedingung verletzt wurde, wurden die Fragen "In welcher Form soll die Auszahlung erfolgen?" und "Bitte geben Sie an wie viel Geld Sie investieren möchten!" ausgeblendet.

denanforderungen zum System durch Aufruf der Methode *setQuestionAnswered* in Listing 4.4 mit den Parametern der zur Reparatur angebotenen Frage und der angebotenen Reparaturoption.

Berechnung der Reparaturen

Zur Berechnung von mehreren Reparaturen ist es zunächst notwendig, alle minimalen Diagnosen mittels *FastDiag* zu berechnen (siehe Kapitel 3.4.3). Die Reparatur Komponente berechnet zu jeder Diagnose eine Reparaturmöglichkeit.
Ausgangspunkt für die Reparatur sind die inkonsistenten Kundenanforderungen, $C_R = knd\_startkapital = 1000$, $knd\_bausparpraemie = nein$, $knd\_auszahlung = regelmäßig$. Von *FastDiag* wird die erste Diagnose $\Delta_1 = knd\_startkapital = 1000$ und die zweite Diagnose $\Delta_2 = knd\_auszahlung = regelmäßig$ ermittelt. Die *conjunctive Query* Komponente erhält die entsprechenden Parameter der Diagnose zum Bau der Query für die Reparatur zu Diagnose Δ_1. Das Ergebnis des Statements in Listing 4.7, sind eine oder mehrere alternative Beantwortungsmöglichkeiten zur Frage "Bitte geben Sie an wieviel Geld Sie investieren möchten?". In diesem Fall gibt die *conjunctive Query* Komponente als ersten Wert "0" zurück. Die Reparatur Komponente baut aus den erhaltenen Reparaturoptionen für alle Diagnosen ein Model von Reparaturoptionen. Der Reparatur Controller kontrolliert die Navigation zwischen den Reparaturen und setzt das *Durchführen* einer Reparatur in Gang. Abbildung 4.19 zeigt die Reparaturdarstellung für die beiden Diagnose Δ_1 und Δ_2.

Listing 4.7: Darstellung der generierten Datenbank Reparatur Query

```
SELECT knd_startkapital.v FROM knd_startkapital, knd_bausparpraemie.v,
    knd_auszahlung,
WHERE
 (knd_bausparpraemie=nein)
AND
 (knd_auszahlung=regelmäßig)
AND
 not(knd_startkapital = 1000)
```

Abbildung 4.19.: Zwei Reparaturen in *AdviSense*: (a) für Diagnose $knd\_startkapital = 1000$ und (b) für Diagnose $knd\_auszahlung = regelmäßig$

Attribut ohne Reparaturmöglichkeit

In der Umsetzung des *AdviSense* Frameworks wurde auch darauf geachtet, dass es Kundenanforderungen geben kann, welche nicht Teil des Reparaturprozesses sind. Im Arbeitsbeispiel des Finanzdienstleistungsrecommenders wird die Frage nach dem Alter von der Reparatur ausgenommen. Wird die Frage vom Benutzer beantwortet, so ist sie eine fixe Größe. Zum Berechnen der Reparaturen mittels *FastDiag* werden die von der Reparatur ausgenommenen Fragen als Constraint an die Wissensbasis C_{KB} angeschlossen.

4.5.5. Optimierungen

Für *conjunctive Query* Statements werden in den Variablen und Domänen konkrete Werte für jede mögliche Anforderung benötigt. Tabellen und deren Werte in der ”v” Spalte repräsentieren diese Variablen und ihre Domänen. Soll das System dem Kunden eine Auswahl von 0 - 15000 Euro anbieten, zum Beispiel in 500er Schritten, so wären dafür 30 Werte in der Datenbank Tabelle notwendig. Diese Vorgehensweise geht jedoch deutlich zu Lasten der Geschwindigkeit der Datenbankabfrage. Daher werden in die ”knd_max_preis” Tabelle nur jene Werte eingetragen, welche für ein Produkt Relevanz besitzen z.B. Prod 1, Einzahlung: 0, Prod 2, Einzahlung: 7000, Prod 3, Einzahlung: 10000. Die *AnswerDetail* Tabelle wurde eingeführt um für große Zahlenauswahlen einen Startwert, einen Endwert und eine Schrittweite angeben zu können. Bei Eingabe eines Wertes, z.B. 7500 Euro, wird auf den

nächsten Wert in der Datenbank abgerundet, in diesem Fall ”7000 Euro”. Betrifft eine Reparatur einen solchen Wert, z.B. falls kein billigeres Bike als 7500 Euro gefunden wurde, so wird der nächst größere Wert aus der Datenbankabelle verwendet. So wäre ”knd_max_preis = 10000 Euro” eine mögliche Reparaturoption.

Kapitel 5

Entwicklungsprozess eines Recommenders im AdviSense Framework

5.1. Übersicht

Dieses Kapitel beschreibt die Vorgehensweise zur Entwicklung eines neuen Beraters auf Basis des *AdviSense* Frameworks. Als Grundlage dafür werden die Constraints des Bike-Recommenders verwendet, der in den Kapiteln 2.2.5 und 3 als Beispiel verwendet wurde. Die Erstellung des iPad Recommenders muss unter MacOS X und XCode* mindestens in Version 4.1 erfolgen. Das Framework für Android ist durch die Entwicklung unter Eclipse† v.3.5 mit dem Android SDK plugin‡ v.12.0.0 plattformunabhängig. Weiters muss ein SQLite Datenbankmanagement Programm auf dem lokalen Rechner installiert werden, dafür stehen einerseits *command line shells* für alle Systeme auf der SQLite Webseite§, andererseits auch Tools mit grafischer Benutzeroberfläche wie das *sqlite-manager* Plugin¶ für den Firefox‖ Browser und SQLiteSpy** zur Verfügung.
Vorweg müssen für die Oberfläche von *AdviSense* die *header* und *footer* Grafik entworfen werden. Die Gestaltung dieser Grafiken, sowie die Wahl der Hintergrundfarbe könnte an eine bereits bestehende Produkt Webseite angelehnt werden. *AdviSense* baut auf einem bereits durchgeführten *Knowledge Engineering Prozess* auf. Die Wissensbasis für den Recommender muss in SQL Syntax Vorliegen.
Basis für die Entwicklung eines neuen Recommenders im *AdviSense* Framework stellen zwei *SQLite* Datenbank Dateien dar. Die Beiden Datenbankdateien stehen in einer Anfangskonfiguration zur Verfügung, welche mit den Parametern des neuen Recommenders ergänzt werden:

*http://developer.apple.com/xcode/
†http://www.eclipse.org
‡http://developer.android.com/sdk/
§http://www.sqlite.org Sektion ”Download”
¶http://code.google.com/p/sqlite-manager/
‖http://www.firefox.com
**http://www.yunqa.de/delphi/doku.php/products/sqlitespy/index

- Die Konfigurationsdatenbank *config.sqlite*
  - Vorlage der Tabelle *advisors* mit den Spalten: key, name, database, thumbnail, title, bgcolor, type, headerimg, footerimg, advisorchoiceimg
  - Vorlage für eine Detail Tabelle eines Beraters *advisor_detail* mit den Spalten: name, products, filters, preconditions, product_tables, customer_tables, repair_excluded. Diese Vorlage muss für jeden neuen Berater kopiert und als Basis verwendet werden.
- Die Beispieldatenbank für einen Recommender *advisorexample.sqlite*
  - Die Tabelle *products* mit den Spalten key, name, detail, thumb, subtitle, bigimg
  - Die Tabelle *questions* mit den Spalten key, questionlong, questionshort, info, knd_variable, answertype
  - Die Tabelle *answerdetail*
  - Vorlage für die Tabelle einer Produkt Variable (*prod_variable*) und Kunden Variable (*knd_variable*) mit den Spalten: *key, v*. Diese Vorlagen können für jede neue Variable kopiert und verwendet werden.

Folgende Schritte müssen nun zur Erstellung durchgeführt werden:

1. Die Konfigurationsdatenbank wird mit den Oberflächenparametern *config.sqlite* gefüllt.
2. Die Produkt-Constraints und Filter-Constraints werden in die Detailtabellen der einzelnen Berater in der *config.sqlite* Datenbank eingefügt.
3. Die Datenbanken für die einzelnen Recommender im System, mit den Tabellen und Inhalten für die Abfragen der *conjunctive Queries* müssen erstellt werden.
4. Die Details und Zusatzinformationen müssen nun in die einzelnen Datenbanken der Recommender eingefügt werden.
5. Die *config.sqlite* Datenbank und die Beraterdatenbanken werden mithilfe der jeweiligen Entwicklungsumgebung in das iPad bzw. Android Projekt eingefügt.
6. Für das Ausführen in im jeweiligen Emulator müssen die Datenbanken in die entsprechenden Emulator Verzeichnisse kopiert werden.
7. Kompilieren des Projektes für Android bzw. iPad.
8. Den Recommender im *Android Market* bzw. den *Appstore* veröffentlichen.

5.2. Erstellung der Konfigurationsdatenbank

Tabelle 5.1 zeigt die Belegung der Datenbank für den Fahrrad Recommender. Zunächst wird das Attribut *name* mit einem Kurznamen des Recommenders belegt, dieser wird in der GUI unter dem Icon

im Beraterwechsel *Popup* angezeigt, falls mehr als ein Recommender im System ist. Die Eigenschaft *title* wird mit einem längeren Titel belegt, welcher in der Titelzeile dargestellt wird. Die beiden Bilder für den Kopfbereich (bike_header.png) und den Banner im Fußbereich (bike_footer.png) müssen unter diesem Namen auch im Basisverzeichnis der Entwicklungsumgebung des Recommenders liegen. Falls diese in einem Unterverzeichnis liegen, ist der entsprechende Pfad anzugeben. In die Spalte *database* wird der Dateiname für die Datenbank, auf der die *conjunctive Queries* ausgeführt werden eingetragen, hier: *bike.sqlite*. Die Hintergrundfarbe (*bgcolor*) wird im RGB Farbschema eingetragen und ist in diesem Beispiel *0.0.0*. Der *type* soll *0* sein, für die Darstellung mit großem Bild, da Fahrräder für den Kunden auch visuell von Interesse sind. Für mehrere Recommender kann in die Datenbank eine zweite Zeile mit Eigenschaften des zweiten Recommenders eingefügt werden.

Spalte	Wert
name	bikes
database	bike.sqlite
thumbnail	bike_thumb.tiff
title	Bike Recommender
bgcolor	0.0.0
type	0
headerimg	bike_header.png
footerimg	bike_footer.png
advisorchoice	bikechoice.png

Tabelle 5.1.: Die Tabelle *advisors* befüllt mit den Daten für einen Bike Recommender

Tabelle 5.2 zeigt die Detail Tabelle, belegt mit den Werten für den Bike-Recommender. Die entsprechenden konjunktiven SQL Constraints für *products*, *filters* und *preconditions* werden in die Datenbank Tabelle eingetragen. Die Tabellennamen der Produkte werden in *product_tables* und jene der Kundenanforderungen in *customer_tables* eingetragen. In diese Tabellen werden die Werte eingetragen, die die Eigenschaften der Produkte annehmen können. Die *customer_tables* beinhalten alle Antworten auf die Frage für die die Tabelle steht. Schließlich folgt das Eintragen der Namen der *Fragen*, die von der Reparatur ausgenommen werden.

Spalte	Wert
name	bikes
products	siehe Listing 5.2
filters	siehe Listing 5.3
preconditions	siehe Listing 5.1
product_tables	siehe Listing 5.4
customer_tables	siehe Listing 5.5
repair_excluded	knd_alter_genau

Tabelle 5.2.: Die Detail Tabelle des Bike Recommenders

5.3. Erstellen der Vorbedingungen aus dem Prozessflussgraphen des Bike Recommender

Als Prozessflussgraph für den hier konstruierten Bike Recommender wird jener Graph (Abb. 3.1) herangezogen, der bereits in Kapitel 3.3 erklärt wurde. Aus dem Empfehlungsprozess müssen nun die entsprechenden Vorbedingungen als *conjunctive Queries* für das *AdviSense* Framework erzeugt werden. Die Fragen *knd_alter*, *knd_max_preis*, *knd_transport* werden immer gestellt, also werden für diese Fragen keine Vorbedingungen benötigt. Die Vorbedingung für *knd_sport* ist *knd_transport = nein* - ein Rad das für Sport verwendet wird, kann nicht für den Transport von Gegenständen eingesetzt werden. Entscheidet sich der Kunde also für ein Rad mit welchem er Gegenstände transportieren kann, so werden die drei Fragen *knd_sport*, *knd_streckenprofil* und *knd_downhill* ausgeblendet (Listing 5.1 Z1). Nur wenn ein Rad für Sport eingesetzt wird, ist die Frage nach dem Streckenprofil (*knd_streckenprofil*) und der Downhillfähigkeit von Interesse, andernfalls werden diese beiden Fragen ausgeblendet (Listing 5.1 Z3). Die dritte Vorbedingung in Listing 5.1 Zeile 5 behandelt die Frage, ob das Bike downhill Eigenschaften besitzt, diese Frage ist nicht von Interesse und wird daher ausgeblendet, wenn der Benutzer zuvor bereits Rad für das Streckenprofil "street" gewählt hat. Die vollständige *conjunctive Query* zeigt Listing 5.1.

Listing 5.1: Die Vorbedingungs Constraints des Bike-Recommenders

```
1 (not(knd_transport.v = 'ja')  OR (not(questions.tbl_name = 'knd_sport') AND
      not(questions.tbl_name = 'knd_streckenprofil') AND not(questions.tbl_name
       = 'knd_downhill')))
2 AND
3 (not(knd_sport.v = 'nein')  OR (not(questions.tbl_name = 'knd_streckenprofil'
      ) AND not(questions.tbl_name = 'knd_downhill')))
4 AND
5 (not(knd_streckenprofil.v = 'street')  OR not(questions.tbl_name = '
      knd_downhill'))
```

5.4. Erstellung der Datenbanken für den Recommender

Die Datenbankinhalte der Datei *bike.sqlite*, werden in den folgenden Abbildungen in der GUI des Firefox Plugin *SQLite Manager* dargestellt. Abbildung 5.1 zeigt die Tabellen im Bike-Recommender (1) und die Tabelle *products* mit einem Eintrag je Produkt (2). Da die Bilder der Fahrräder in Großansicht präsentiert werden sollen, (siehe Tab. 5.1 Parameter *type = 0*) muss ein Eintrag des großen Bildes in Spalte *bigimg* erfolgen. Auf Einträge in Spalte *thumb* kann verzichtet bzw. *none* eingetragen werden. Außer der Tabelle *products* sind auch Tabellen *questions* und *answerdetail* fixer Bestandteil jedes mit *AdviSense* umgesetzten Recommenders. Die Daten der einzelnen Fragen werden durch Einträge in der Tabelle *questions* repräsentiert (siehe Abb. 5.2). Die Einträge der Spalte *tbl_name* stellen die Zuordnungen der Fragen zu den Tabellen für die *conjunctive Queries* dar (Abb. 5.1(1)). Diese

und die Produkt Tabellen (Abb. 5.1(1)) müssen angelegt und mit Werten belegt werden. Fragen mit Zahlenantworten besitzen den Eintrag *number* in Spalte *answertype*. Zusätzlich werden die Eigenschaften der Antwortmöglichkeiten in Tabelle *answerdetail* eingetragen: Das Alter (*knd_alter_genau*) reicht in diesem Fall von 6 bis 99 Jahren mit einer Schrittweite von 1, der Preis (knd_preis) reicht von 1500 bis 15000 Euro mit einer Schrittweite von 500. Wie in dieser Tabelle vorgegeben erscheint in der Benutzeroberfläche die Auswahl der Antwortmöglichkeiten. Die Kundentabellen und Produkttabellen sind in den Abbildungen 5.5 bzw. 5.4 dargestellt. Die Datenbanken und Bilder können in den Entwicklungsumgebungen *XCode* bzw. *Eclipse* per *Drag and Drop* zum Projekt hinzugefügt werden.

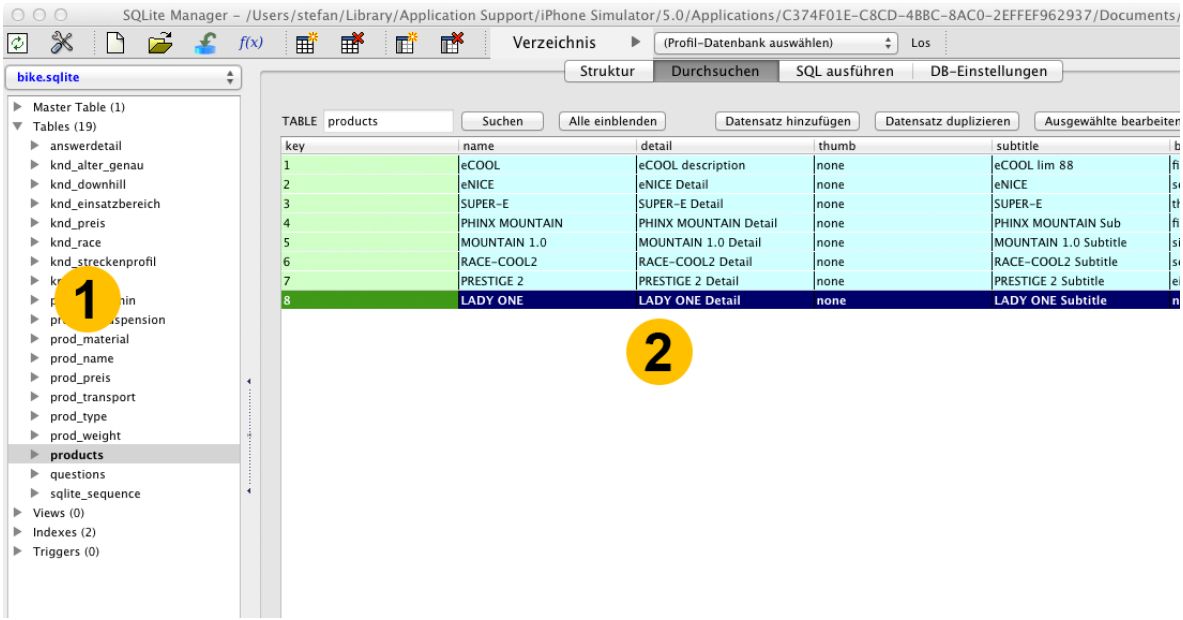

Abbildung 5.1.: Screenshot der Oberfläche des Firefox plugin *SQLite Manager* der *products* Tabelle in der *bike.sqlite* Datenbank. (1) zeigt die vorhandenen Tabellen der Datenbank, (2) zeigt den Inhalt der markierten *products* Tabelle

Abbildung 5.2.: Screenshot der *questions* Tabelle in der *bike.sqlite* Datenbank

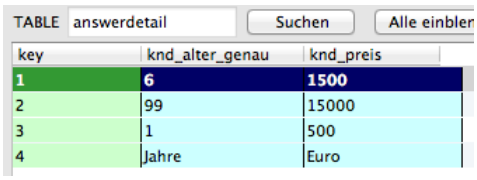

Abbildung 5.3.: Screenshot der *answerdetail* Tabelle in der *bike.sqlite* Datenbank. *key = 1*: Startwert; *key = 2*: Endwert; *key = 3*: Schrittweite; *key = 4*: Einheit

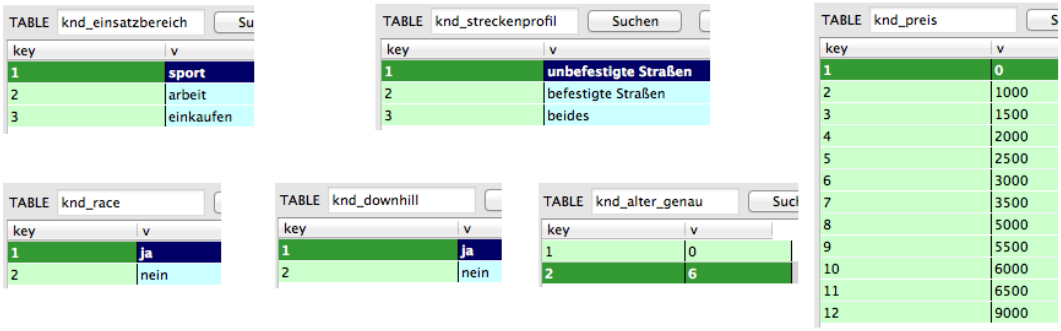

Abbildung 5.4.: Screenshot der Kundentabellen der *bike.sqlite* Datenbank des Bike Recommender. In Tabelle *knd_einsatzbereich* sind beispielsweise die Domänenwerte des Einsatzbereiches gespeichert: sport, Arbeit, einkaufen

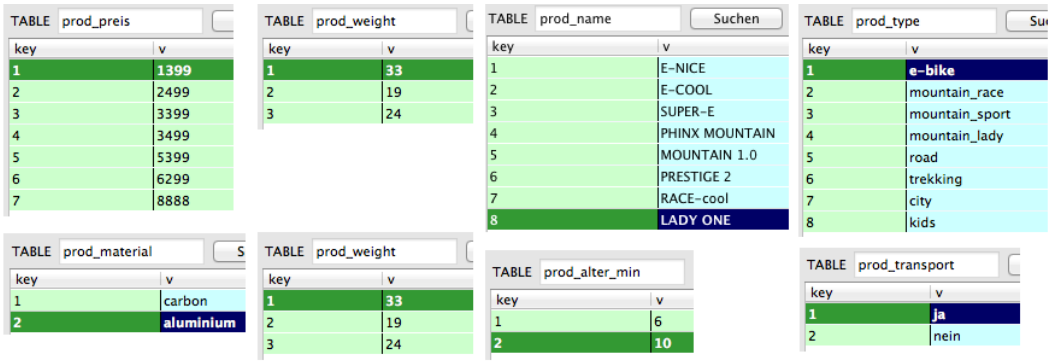

Abbildung 5.5.: Screenshot der Produkttabellen der *bike.sqlite* Datenbank des Bike Recommender. In Tabelle *knd_material* sind beispielsweise die Domänenwerte der Materialien aus denen Fahrräder bestehen können gespeichert: carbon und aluminium

5.5. Ausführen des fertigen Produktes

Nachdem alle Daten im Projekt Ordner liegen muss das Projekt nur noch ausgeführt werden. Die folgenden Bilder zeigen den Bike Recommender im iPad (Abb. 5.6) und im Android Emulator (Abb. 5.7).

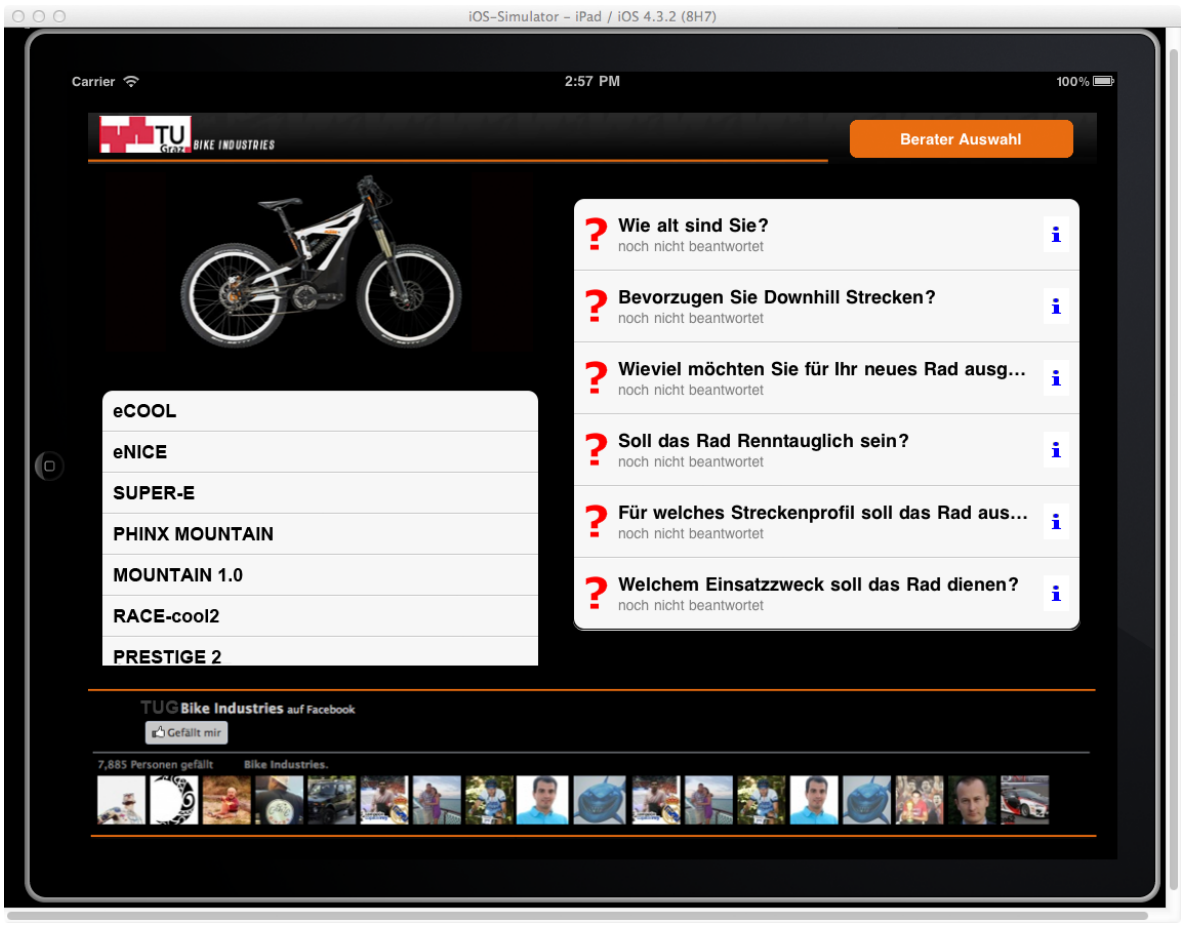

Abbildung 5.6.: Das Screen des fertigen Bike Recommender im iPad Emulator

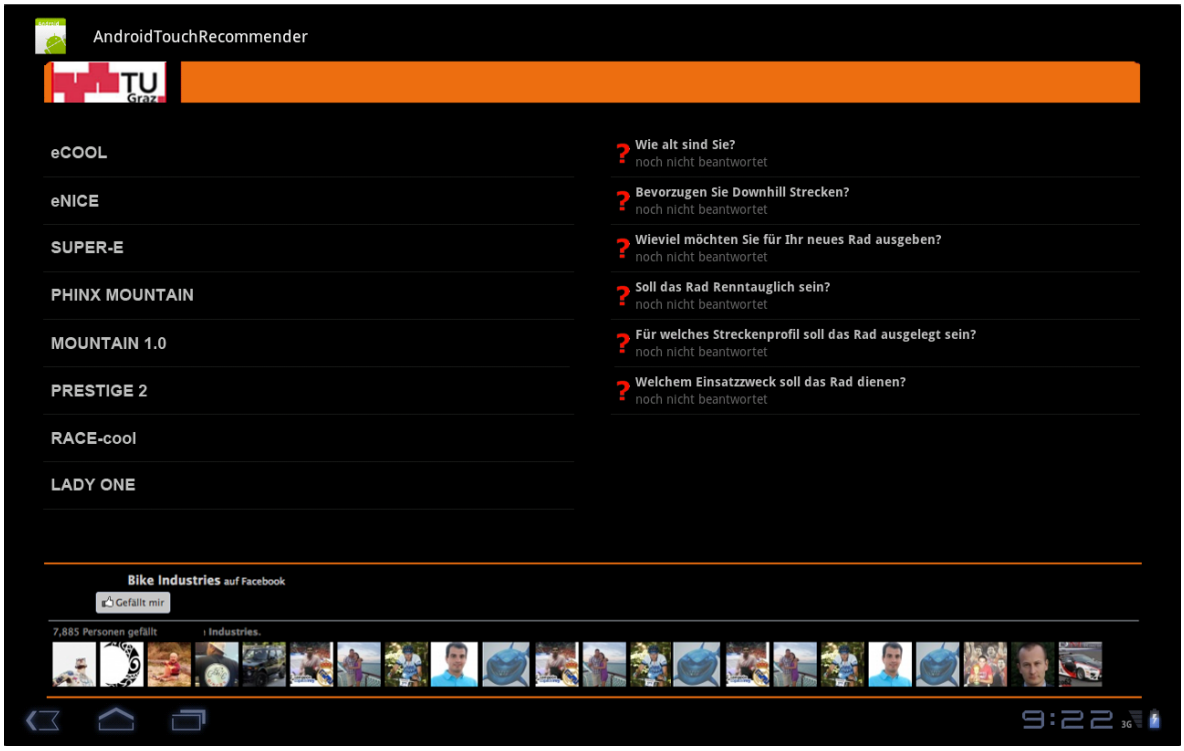

Abbildung 5.7.: Das Screen des fertigen Bike Recommender im Android Emulator

5.6. Die Constraints des Bike-Recommenders

Listing 5.2: Die Produkt Constraint des Bike-Recommenders

```
1  ((prod_name.v = 'eCOOL' AND prod_type.v = 'e-bike' AND prod_fullsuspension.v =
       'ja' AND prod_alter_min.v = 10 AND prod_preis.v = 8888 AND prod_transport
      .v = 'nein'  AND prod_material.v = 'aluminium')
2  OR
3  (prod_name.v = 'E-NICE' AND prod_type.v = 'e-bike' AND prod_fullsuspension.v =
       'ja' AND prod_alter_min.v = 10 AND prod_preis.v = 3499 AND prod_transport
      .v = 'nein' AND prod_material.v = 'aluminium')
4  OR
5  (prod_name.v = 'SUPER-E' AND prod_type.v = 'e-bike' AND prod_fullsuspension.v
      = 'nein' AND prod_alter_min.v = 10 AND prod_preis.v = 2499 AND
      prod_transport.v = 'ja'  AND prod_material.v = 'aluminium')
6  OR
7  (prod_name.v = 'PHINX MOUNTAIN' AND prod_type.v = 'mountain_race' AND
      prod_fullsuspension.v = 'ja' AND prod_alter_min.v = 10 AND prod_preis.v =
      6299 AND prod_transport.v = 'nein' AND prod_material.v = 'carbon')
8  OR
9  (prod_name.v = 'MOUNTAIN 1.0' AND prod_type.v = 'mountain_race' AND
      prod_fullsuspension.v = 'ja' AND prod_alter_min.v = 10 AND prod_preis.v =
      3399 AND prod_transport.v = 'nein' AND prod_material.v = 'aluminium')
10 OR
11 (prod_name.v = 'PRESTIGE 2' AND prod_type.v = 'mountain_race' AND
      prod_fullsuspension.v = 'nein' AND prod_alter_min.v = 10 AND prod_preis.v
      = 5399 AND prod_transport.v = 'nein' AND prod_material.v = 'carbon')
12 OR
```

```
13 (prod_name.v = 'RACE-cool' AND prod_type.v = 'mountain_sport' AND
       prod_fullsuspension.v = 'nein' AND prod_alter_min.v = 10 AND prod_preis.v
       = 1399 AND prod_transport.v = 'nein' AND prod_material.v = 'aluminium')
14 OR
15 (prod_name.v = 'LADY ONE' AND prod_type.v = 'mountain_lady' AND
       prod_fullsuspension.v = 'ja' AND prod_alter_min.v = 10 AND prod_preis.v =
       1399 AND prod_transport.v = 'nein' AND prod_material.v = 'aluminium'))
```

Listing 5.3: Die Filter Constraints des Bike-Recommenders

```
1  AND
2  (prod_alter_min.v <= knd_alter_genau.v)
3  AND
4  (prod_preis.v <= knd_preis.v)AND(not(knd_streckenprofil.v = 'unbefestigte
       Straßen') OR ((prod_type.v = 'mountain_race') OR (prod_type.v = '
       mountain_sport') OR (prod_type.v = 'mountain_lady') OR (prod_type.v = '
       trekking') OR (prod_type.v = 'kids')))
5  AND
6  (not(knd_streckenprofil.v = 'befestigte Straßen') OR ((prod_type.v = '
       mountain_lady') OR (prod_type.v = 'kids') OR (prod_type.v = 'city') OR (
       prod_type.v = 'road') ))
7  AND
8  (not(knd_streckenprofil.v = 'beides') OR (not((prod_type.v = 'city') OR (
       prod_type.v = 'road'))))AND(not(knd_downhill.v = 'ja') OR (
       prod_fullsuspension.v = 'ja'))AND(not(knd_race.v = 'ja') OR (prod_material
       .v = 'carbon'))AND(not(knd_einsatzbereich.v = 'einkaufen') OR (
       prod_transport.v = 'ja'))
9  AND
10 (not(knd_einsatzbereich.v = 'arbeit') OR ((prod_transport.v = 'ja')
11 AND
12 (prod_type.v = 'e-bike')))
```

Listing 5.4: Die Produkt Tabellennamen des Bike-Recommenders

```
1  prod_name, prod_type, prod_fullsuspension, prod_weight, prod_material,
       prod_preis, prod_alter_min, prod_transport
```

Listing 5.5: Die Kundenanforderungs Tabellen des Bike-Recommenders

```
1  questions, knd_alter_genau, knd_downhill, knd_preis, knd_race,
       knd_streckenprofil, knd_einsatzbereich
```

Kapitel 6

Ergebnisse

In dieser Arbeit wurde mit *AdviSense* ein Framework zur Erstellung von wissensbasierten Recommendern vorgestellt. Aufbauend auf einem *Knowledge Engineering* Prozess können damit für verschiedene Produktdomänen Recommender erstellt werden. Die Eingabe der Kundenanforderungen, sowie die Darstellung der Produkte erfolgt auf einem Screen. Zur Darstellung der Produkte werden zwei Varianten angeboten, einerseits die Listendarstellung mit Icons welche sich für Dienstleistungen wie Finanzdienstleistungen oder Internet Provider Angebote gut eignet, andererseits die Listendarstellung mit großem Bild für reale Produkte wie Fahrräder, Motorräder und andere visuell interessante Produkte. Zur Integration der Reparaturen in das Gesamtkonzept der Benutzeroberfläche wird die zu reparierende Frage ausgeblendet und symbolisch an Stelle der leeren Produktliste eingeblendet. Zur Berechnung der Diagnosen wurde ein innovativer Algorithmus zur Berechnung von präferierten Diagnosen (FastDiag) in das Recommender Framework integriert. Die Unterstützung mehrerer Recommender in einem System wird durch das *Berater wechseln* Menü unterstützt. Mittels *Popups* wurde die Anzeige von Detailinformationen für Produkte und Fragen umgesetzt. Durch die Entscheidung für SQLite Datenbanken zur Konfiguration der Oberfläche und zum Ablegen und Ausführen der *conjunctive Queries* kann ein erstellter Recommender sowohl auf dem iPad als auch auf Android Tablet betrieben werden.

Kapitel 7

Future Work

Auf Basis der erreichten Ziele in dieser Arbeit gibt es mehrere Richtungen in welche dieses Framework weiterentwickelt werden kann:

- **Präferierte Diagnosen:** Da FastDiag die Möglichkeit bietet, präferierte Diagnosen zu erstellen wäre es naheliegend diese in einer Erweiterung in *AdviSense* zu integrieren. Reparaturen könnten dann in einer für den Benutzer interessanten Reihenfolge ausgeben ausgegeben werden.
- **Erweiterung der Werbefläche:** Die Werbefläche für Cross/Upselling ist momentan ein Statischer Banner mit einer Verlinkung, die Webseite hinter der Verlinkung ist zwar veränderbar, eine bessere Einbindung der Fläche in den Empfehlungsprozess wäre sicher sinnvoll. Der Banner könnte angepasst an die aktuelle Produktempfehlung spezifische Cross/Upselling Möglichkeiten bieten.
  Alternativ könnte die Werbefläche, falls sie als solche nicht benötigt wird für die Darstellung von Produktdetails beim Antippen der Produkte verwendet werden.
- **Einführung eines Multi-Attribute Utility Theory (MAUT) Systems** (siehe Schmitt et al. (2003) und Felfernig et al. (2006)) in *AdviSense*. Durch den Einsatz eines MAUT Systems könnten die Produktvorschläge gerankt werden.
- **Unterschiedliche Auflösungen unter Android unterstützen:** Sowohl iPad 1 als auch iPad 2 besitzen die selbe Bildschirmauflösung. Die verwendeten Grafiken der Benutzeroberfläche können also fixe Werte besitzen. Android Tablets ab Version 3.2 besitzen dagegen unterschiedliche Auflösungen. Das *relative Layout* der *AdviSense* Android Oberfläche müsste für zukünftige Versionen dahingehend weiterentwickelt werden von 800 x 480 Pixel bis 1.280 x 800 Pixel mehrere Auflösungen zu unterstützen.
- **Erweiterung der Darstellungsmöglichkeiten der Produkte:** Momentan unterstützt das System 2 Produktdarstellungen: Typ *0* die Darstellung mit großem Bild, und Typ *1* Darstellung in Liste mit Icons. Die Darstellungsvariante mit großem Bild könnte, zum Beispiel bei antippen

des Bildes um dessen Vergrößerung auf Fullscreen unterstützen. Eine weitere Möglichkeit wäre beim Antippen das Einblenden von technischen Daten des Produktes anstatt der Anzeige eines Bildes.

- **Beschleunigung der Reparatur Berechnung:** Das Berechnen der Reparaturen kann beschleunigt werden, wenn beim Feststellen inkonsistenter Anforderungen nicht sofort alle Diagnosen und deren Reparaturen berechnet würden, sondern wenn deren Anzeige durch antippen des ”nächste” Buttons notwendig ist.
- **Anpassen der Android Implementierung an die iPad Implementierung:**
  - Ein Ersatzelement für das unter iOS verwendete *Popover* Menü finden, um die Produkt Details und Fragen Infos darstellen zu können.
  - Implementierung der Produkt Darstellung mit großem Bild für Android
  - Anpassung der Darstellung der Reparaturmöglichkeiten an jene am iPad (Design wie eine Frage aus der rechten Fragenliste).

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

ACIAR, S., ZHANG, D., SIMOFF, S., AND DEBENHAM, J. 2007. Informed recommender: Basing recommendations on consumer product reviews. *Intelligent Systems, IEEE 22,* 3 (may-june), 39 –47. (Cited on page 14.)

ADOMAVICIUS, G. AND TUZHILIN, A. 2005. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on 17,* 6 (june), 734 – 749. (Cited on page 17.)

BARTÁK, R. 1999. Constraint programming: In pursuit of the holy grail. In *Proceedings of Week of Doctoral Students (WDS99)*. 555–564. (Cited on page 3.)

BRAILSFORD, S. C., POTTS, C. N., AND SMITH, B. M. 1999. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research 119,* 3 (December), 557–581. (Cited on pages 4, 5, 6, 7, and 8.)

BRIN, S. AND PAGE, L. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems 30,* 1-7, 107 – 117. Proceedings of the Seventh International World Wide Web Conference. (Cited on page 12.)

BURKE, R. 1999. Integrating knowledge-based and collaborative-filtering recommender systems. In *In AAAI Workshop on AI in Electronic Commerce*. AAAI, 69–72. (Cited on page 26.)

BURKE, R. 2000. Knowledge–based recommender systems. *Encyclopedia of Library and Information Science 69,* 32. (Cited on pages 18 and 19.)

BURKE, R. 2002. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction 12,* 4, 331–370. 10.1023/A:1021240730564. (Cited on page 11.)

CARD, S. K., ROBERTSON, G. G., AND MACKINLAY, J. D. 1991. The information visualizer, an information workspace. In *Proceedings of the Conference on Human Factors in Computing Systems: Reaching Through Technology*. ACM, New York, NY, USA, 181–186. (Cited on page 34.)

CASTILLO, L., BORRAJO, D., AND SALIDO, M. 2005. *Planning, Scheduling and Constraint Satisfaction: From Theory to Practice.* IOS Press, 2005. (Cited on page 3.)

CHUN, I.-G. AND HONG, I.-S. 2001. The implementation of knowledge-based recommender system for electronic commerce using java expert system library. In *Industrial Electronics, 2001. Proceedings. ISIE 2001. IEEE International Symposium on*. Vol. 3. 1766 –1770 vol.3. (Cited on page 18.)

DE KLEER, J., MACKWORTH, A. K., AND REITER, R. 1992. Characterizing diagnoses and systems. *Artif. Intell. 56,* 2-3 (August), 197–222. (Cited on page 34.)

FELFERNIG, A. 2005a. Knowledge-based interactive selling of financial services using fsadvisor. In *17 th Innovative Applications of Artificial Intelligence Conference (IAAI'05*. AAAI Press, 1475–1482. (Cited on pages 11, 18, 23, 24, 33, and 41.)

FELFERNIG, A. 2005b. Koba4ms: Selling complex products and services using knowledge-based recommender technologies. In *CEC '05: Proceedings of the Seventh IEEE International Conference on E-Commerce Technology*. IEEE Computer Society, Washington, DC, USA, 92–100. (Cited on page 18.)

FELFERNIG, A. AND BURKE, R. 2008. Constraint-based recommender systems: technologies and research issues. In *Proceedings of the 10th international conference on Electronic commerce*. ICEC '08. ACM, New York, NY, USA, 3:1–3:10. (Cited on pages 11, 12, 18, 20, and 30.)

FELFERNIG, A., FRIEDRICH, G., JANNACH, D., AND STUMPTNER, M. 2004. Consistency-based diagnosis of configuration knowledge bases. *Artif. Intell. 152,* 2 (February), 213–234. (Cited on page 34.)

FELFERNIG, A., FRIEDRICH, G., JANNACH, D., AND ZANKER, M. 2006. An integrated environment for the development of knowledge-based recommender applications. *Int. J. Electron. Commerce 11,* 2 (December), 11–34. (Cited on pages 29 and 71.)

FELFERNIG, A., FRIEDRICH, G., SCHUBERT, M., MANDL, M., MAIRITSCH, M., AND TEPPAN, E. 2009. Plausible repairs for inconsistent requirements. In *Proceedings of the 21st international jont conference on Artifical intelligence*. IJCAI'09. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 791–796. (Cited on page 33.)

FELFERNIG, A. AND GULA, B. 2006. An empirical study on consumer behavior in the interaction with knowledge-based recommender applications. In *E-Commerce Technology, 2006. The 8th IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services, The 3rd IEEE International Conference on*. 37. (Cited on page 18.)

FELFERNIG, A., ISAK, K., SZABO, K., AND ZACHAR, P. 2007. The vita financial services sales support environment. In *Proceedings of the 19th national conference on Innovative applications of artificial intelligence - Volume 2*. AAAI Press, 1692–1699. (Cited on pages 11, 24, and 25.)

FELFERNIG, A. AND SCHUBERT, M. 2010. A diagnosis algorithm for inconsistent constraint sets. In *Proceedings of the 21st International Workshop on the Principles of Diagnosis*. 31–38. (Cited on pages 33, 34, and 35.)

FELFERNIG, A. AND SHCHEKOTYKHIN, K. 2006. Debugging user interface descriptions of knowledge-based recommender applications. In *Proceedings of the 11th international conference on Intelligent user interfaces*. IUI '06. ACM, New York, NY, USA, 234–241. (Cited on page 33.)

FLEISCHANDERL, G., FRIEDRICH, G., HASELBOCK, A., SCHREINER, H., AND STUMPTNER, M. 1998. Configuring large systems using generative constraint satisfaction. *Intelligent Systems and their Applications, IEEE 13,* 4 (jul/aug), 59 –68. (Cited on page 3.)

FREUDER, E. C. 1997. In pursuit of the holy grail. *Constraints 2,* 1, 57–61. (Cited on page 3.)

GOLDBERG, D., NICHOLS, D., OKI, B. M., AND TERRY, D. 1992. Using collaborative filtering to weave an information tapestry. *Communications of the ACM 35,* 12, 61–70. (Cited on page 13.)

HENTENRYCK, P. V. AND SARASWAT, V. A. 1997. Constraint programming: Strategic directions. *Constraints 2,* 1, 7–33. (Cited on page 4.)

JANNACH, D., ZANKER, M., FELFERNIG, A., AND FRIEDRICH, G. 2010. *Recommender Systems: An Introduction*, 1 ed. Cambridge University Press. (Cited on pages 11, 14, 15, and 16.)

JUNKER, U. 2004. Quickxplain: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th national conference on Artifical intelligence*. AAAI'04. AAAI Press, 167–172. (Cited on page 34.)

LINDEN, G., SMITH, B., AND YORK, J. 2003. Amazon.com recommendations: item-to-item collaborative filtering. *Internet Computing, IEEE 7,* 1 (jan/feb), 76 – 80. (Cited on page 11.)

MIRZADEH, N., RICCI, F., AND BANSAL, M. 2005. Feature selection methods for conversational recommender systems. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05) on e-Technology, e-Commerce and e-Service*. EEE '05. IEEE Computer Society, Washington, DC, USA, 772–777. (Cited on pages 20, 22, 23, and 73.)

O'SULLIVAN, B., PAPADOPOULOS, A., FALTINGS, B., AND PU, P. 2007. Representative explanations for over-constrained problems. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1*. AAAI Press, 323–328. (Cited on page 33.)

PAZZANI, M. AND BILLSUS, D. 2007. Content-based recommendation systems. In *The Adaptive Web*, P. Brusilovsky, A. Kobsa, and W. Nejdl, Eds. Lecture Notes in Computer Science, vol. 4321. Springer Berlin / Heidelberg, 325–341. 10.1007/978-3-540-72079-9_10. (Cited on page 11.)

REITER, R. 1987. A theory of diagnosis from first principles. *Artif. Intell. 32,* 1 (April), 57–95. (Cited on page 34.)

RESNICK, P., IACOVOU, N., SUCHAK, M., BERGSTROM, P., AND RIEDL, J. 1994. *GroupLens: an open architecture for collaborative filtering of netnews*. Vol. pp. ACM, 175–186. (Cited on page 13.)

RICCI, F. 2011. Mobile recommender systems. *International Journal of Information Technology and Tourism. 12,* 3, 205–231. (Cited on page 25.)

RICCI, F., CAVADA, D.AND MIRZADEH, N., AND VENTURINI, A. 2006. Case-based travel recommendations. In *Destination Recommendation Systems: Behavioural Foundations and Applications*. CABI, 67–93. (Cited on page 27.)

RICCI, F. AND NGUYEN, Q. N. 2007. Acquiring and revising preferences in a critique-based mobile recommender system. *Intelligent Systems, IEEE 22,* 3 (may-june), 22 –29. (Cited on pages 24 and 25.)

RUSSELL, S. J. AND NORVIG, P. 2003. *Artificial Intelligence: a modern approach*, 2nd international edition ed. Prentice Hall. (Cited on pages 5 and 6.)

SARWAR, B., KARYPIS, G., KONSTAN, J., AND REIDL, J. 2001. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*. ACM, New York, NY, USA, 285–295. (Cited on page 16.)

SCHMITT, C., DENGLER, D., AND BAUER, M. 2003. Multivariate preference models and decision making with the maut machine. In *User Modeling 2003*, P. Brusilovsky, A. Corbett, and F. de Rosis, Eds. Lecture Notes in Computer Science, vol. 2702. Springer Berlin / Heidelberg, 148–148. 10.1007/3-540-44963-9_40. (Cited on pages 24 and 71.)

SEEWALD, A. K. 2007. An evaluation of naive bayes variants in content-based learning for spam filtering. *Intell. Data Anal. 11*, 497–524. (Cited on page 11.)

SHARDANAND, U. AND MAES, P. 1995. Social information filtering: algorithms for automating 'word of mouth'. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. CHI '95. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 210–217. (Cited on page 13.)

SI, H., KAWAHARA, Y., KURASAWA, H., MORIKAWA, H., AND AOYAMA, T. 2005. A context-aware collaborative filtering algorithm for real world oriented content delivery service. In *Ubicomp Metapolis and Urban Life Workshop*. (Cited on page 12.)

SINZ, C., HAAG, A., NARODYTSKA, N., WALSH, T., GELLE, E., SABIN, M., JUNKER, U., O'SULLIVAN, B., RABISER, R., DHUNGANA, D., GRUNBACHER, P., LEHNER, K., FEDERSPIEL, C., AND NAUS, D. 2007. Configuration. *IEEE Intelligent Systems 22,* 1 (January), 78–90. (Cited on page 3.)

SMYTH, B., MCGINTY, L., REILLY, J., AND MCCARTHY, K. 2004. Compound critiques for conversational recommender systems. In *Web Intelligence, 2004. WI 2004. Proceedings. IEEE/WIC/ACM International Conference on*. 145 – 151. (Cited on pages 11, 13, 18, 19, and 20.)

SUTHERLAND, I. E. 1963. Sketchpad: A Man-Machine Graphical Communication System. In *Proceedings of the 1963 Spring Joint Computer Conference*, E. C. Johnson, Ed. AFIPS Conference Proceedings, vol. 23. American Federation of Information Processing Societies, Spartan Books Inc., Baltimore, MD, 329–346. (Cited on page 3.)

THOMPSON, C. A., GÖKER, M. H., AND LANGLEY, P. 2004. A personalized system for conversational recommendations. *J. Artif. Int. Res. 21,* 1 (March), 393–428. (Cited on page 26.)

ZANKER, M. 2008. A collaborative constraint-based meta-level recommender. In *Proceedings of the 2008 ACM conference on Recommender systems*. RecSys '08. ACM, New York, NY, USA, 139–146. (Cited on page 26.)