FAKULTÄT FÜR !INFORMATIK

Design and Implementation of a Generic Recommender and Its Application to the Music Domain

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Roman Cerny
Matrikelnummer 9725401

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber

Wien, 15.10.2008 ____________________ ____________________
(Unterschrift Verfasser) (Unterschrift Betreuer)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Acknowledgements

I would like to dedicate this thesis to my great-aunt Anna Zanetos, who always cared for me like a grandmother and wishes nothing more than seeing me become the first member of our family to ever graduate.

My greatest appreciation belongs to both, my mother Anna Cerny and my beloved wife Daniela Cerny, who have always supported me throughout my studies and were always confident of me. During hard times in my life their unconditional love always rejuvenated my mental powers.

I want to thank the *Smart Agent Technologies Studio* (SAT)[1] (which is part of the *Research Studios Austria Forschungsgesellschaft mbH* (RSA)[2]) for creating a context that made this work possible. Many thanks go to my colleagues Erich Gstrein and Brigitte Krenn who guided me during this project in countless discussions with many useful technical and research-based advices. Furthermore I want to thank my colleague Stephan Zavrel for providing his in-depth knowledge of Web services. Special thanks go to my colleague and dear friend Florian Kleedorfer for many precious hints and suggestions concerning all issues one runs through when writing a masters thesis, no matter if they were technical, research-based, methodical or even psychological. Thanks also go to my long-time friend Christian Zanoni for sharing his time and know-how on the conversion of some picture files.

Deep gratitude is owned by Erich Gstrein for providing an adapted implementation of one of the recommender methods he has been researching on recently (the SAT *Association Rule Miner*) for the integration as offline item association generator.

Moreover I want to thank Erich Gstrein, Florian Kleedorfer, Brigitte Krenn, Gernot Sattler, Patrick Wertitsch, and Stephan Zavrel for proofreading and feedback on the work at hand.

Special thanks also go to my supervisor Prof. Dr. Andreas Rauber for guidance and constructive criticism.

Furthermore I want to thank the developers of the client applications *Music Explorer*[3] (Peter Hlavac), *RASCALLI Environment*[4] (David Mann) and *Visual Browser*[5] (Xiwen Cheng and Peter Adolphs) for the good cooperation during the integration of the *Recommender System* (RS).

[1]`http://sat.researchstudio.at`, as of 2008/08/21

[2]`http://www.researchstudio.at`, as of 2008/08/21

[3]`http://rascalli.researchstudio.at/searcher.form`, as of 2008/09/30

[4]`http://intralife.researchstudio.at/rascalli`, as of 2008/09/30

[5]`http://rascalli.dfki.de/live/`, as of 2008/09/30

Additionally I would like to thank Sean Owen (the author of the *Taste*[6] framework) for assisting me during the integration of *Maven2* builds into the *Taste* framework and the evaluation of a *SlopeOne* algorithm.

Further thanks go to the *Verisign Communications GmbH*[7] for their good cooperation and especially for providing a large dataset of user ratings on music items (in order to evaluate a rating-based recommender algorithm).

The work is partially funded from the *EC Cognitive Systems Project FP6-IST-027596-2004 RASCALLI* [8], which is funded under the 6th FWP (*Sixth Framework Programme*). Parts of this thesis were incorporated in *RASCALLI* deliverables.

Finally kindly regards go to all of my fellow students I shared time with during my studies: Arash Amiri, Martin Beinhart, Markus Cozowicz, Wolfgang Deix, Peter Hlavac, Jörg Irran, Florian Kleedorfer, Christoph Leder, Andreas Müller, Bernhard Pollak, Stefan Pomajbik, Stefan Rank, Robert Stepanek and Stephan Zavrel. Due to all of you it was possible to survive boring lectures and pass incredibly difficult courses without ever losing sight of the funny sides of life.

[6] `http://taste.sourceforge.net/`, as of 2008/06/20
[7] `http://www.3united.com`, as of 2008/06/20
[8] `http://www.ofai.at/rascalli`, as of 2008/06/21

Kurzfassung

Durch den stetigen Anstieg der Anzahl zur Verfügung stehender Datenquellen durch technologische Entwicklungen steigt der Bedarf an sinnvollen Techniken zur Filterung großer Datenmengen ebenso konstant. Für diesen Zweck wurden bisher diverse existierende Recommender Systeme eingesetzt. Oftmals liegt der Fokus solcher Systeme auf einem eingeschränkten Datenbestand oder auf einer bestimmten Menge von Vorschlagsmethoden.

Diese Arbeit präsentiert ein generisches Recommender Framework, welches einfach an neue Domänen adaptiert werden kann und das leicht mit unterschiedlichsten Recommender Algorithmen erweitert werden kann. Algorithmen werden unterteilt in: *offline* Generatoren, die aus Benutzeraktionen Regeln ableiten, und *online* Recommender Services. Um Mandantenfähigkeit zu gewährleisten beinhaltet das Konzept verschiedene Typen für Produkte, Aktionen und Verbindungen zwischen Produkten. Weiters unterstützt das System die Einbindung von fremden Daten und bietet eine Menge an Web Services für domänenunabhängige und domänenabhängige Vorschläge.

Wir zeigen die technische Architektur sowie ein Real-Life Szenario, in das unser System integriert wurde. Außerdem geben wir einen Ausblick über zukünftige Verbesserungen wie ein Administrations-Tool, weitere Vorschlagsmethoden oder neue Ansätze die Benutzer-Tags verwenden.

Abstract

Since the amount of available data sources increases steadily with technological developments, the need for useful filtering techniques for large data sets rises constantly. Various existing recommender systems have been used for this task. Often the focus lies on a specific content, or a defined set of recommender techniques.

This work introduces a generic recommender framework that can be easily adapted to new domains and extended with different recommender algorithms. It divides algorithms into *offline* generators computing business rules out of user actions, and *online* recommender services. In order to service multiple tenants the concept consists of various types for items, actions and item associations. Furthermore it allows for the integration of third-party data and provides a set of Web services for domain-independent as well as domain-specific recommendations.

We demonstrate the technical architecture as well as a real-life scenario where we integrated our system and give a prospect to future enhancements like an administration tool, additional recommender techniques, or novel approaches that use custom user tags.

Contents

Chapter 1
Introduction

1.1 Motivation

During the past years we have worked together with business partners of the *Smart Agent Technologies Studio* (SAT)[1], researching the field of recommender systems and introducing new approaches for the customers' recommenders. Our tasks focused mainly on the improvement of the quality of recommendations used in a large music download platform in order to increase in sales. We ran evaluations on prototypical algorithm implementations over the customers' content and provided our learnings and findings on a conceptual level to our partners.

Before we launched this project and started working on this master's thesis more customers with applications in different domains showed interest in our research. Since the type of data varies (and is strongly bound to the customer's domain), a single best suiting recommender technique cannot be determined for different needs. Suitable methods can vary from rating-based over content-based approaches to various hybrid combinations.

To adapt to the customers' requirements we decided to implement our own RS that would be able to meet different types of requirements.

1.2 Goals

Since recommenders are mainly used in Web applications the requirements of such systems are mostly domain-specific and tend to change quite often, as the market adapts to new domains rapidly when a profitable outcome can be expected.

In order to adjust to the fast development of different recommender approaches we focused our work on the following research questions:

- Which components and their implied features are necessary for a generic recommender system to be easily adaptable to new algorithms and methods, as well as to

[1] `http://sat.researchstudio.at`, as of 2008/08/21

allow for the operation as a managed system for many applications, tenants, and domains?

- What is necessary to provide a basic implementation of such components and how do they work together in a real-life application?

Thus we have derived the following goals for the work presented in this diploma thesis:

- Identify the major features of a generic RS.
- Design and implement components providing such features, and furthermore combine these components to a RS.
- Exemplarily integrate this RS into a real-life application for a specific use case, and demonstrate the functional efficiency.

We would like to note that it is not our goal to implement and evaluate different recommender algorithms but rather to show that our RS concept works by implementing one prototypical algorithm and applying it to the RS.

1.3 Structure of the Work

Starting with a brief introduction in the field of recommender systems, Chapter 2 presents different types of recommender techniques as well as various evaluation measures and discusses common issues. The subsequent chapter (Chapter 3) lists some existing recommender frameworks, giving an overview of academic recommender projects and commercial recommender products as well as some open-source recommender frameworks. The Chapters 4 and 5 are the main chapters of this work, presenting the concept, picturing the most important modules, and describing the implementation details of the RS. We demonstrate a real-life showcase and applications that use the RS in Chapter 6. We summarise our learnings and conclusions in Chapter 7 and give an outlook on possible enhancements and further research in Chapter 8. In Appendix A we present the evaluation of a rating-based recommender algorithm implementation using an open-source recommender framework while Appendix B presents the *Software* (SW) infrastructure. Finally Appendix C lists the existing Web service interfaces.

Chapter 2
Theory of Recommender Systems

Abstract

This chapter introduces the field of recommender systems. After explaining the basic mode of operation of a RS in Section 2.1 and giving a brief overview of the history in Section 2.2, we present common methods as well as their differences in Section 2.3 and evaluation measures for them in Section 2.4. Section 2.5 points out typical problems that are often observed in a RS. Finally Section 2.6 gives a survey of common issues related to recommender systems.

2.1 Basic Functionality

Recommender systems are used for the task of information filtering. Users are confronted with a large amount of information in many applications. In order to reduce the information presented to the user, recommender systems usually filter out a set of items that should fit the user's preferences best.

The most common use case for recommender systems is to suggest items which are of interest to users. These compiled item lists can be based on explicit and implicit user preferences, the preferences of other users, and users' as well as items' attributes. For example a music related recommender may compute the recommendations combining explicit user ratings (e.g. User A rated track X a 4 out of 5), implicit data (e.g. User B purchased track Y), user demographic information (e.g. User A is 32 years old) and item content information (e.g. Track X belongs to genre pop, or track Y runs at a speed of 90 BPM[1]).

Technically, recommender systems are assigned to tasks of *Regression* (the prediction of real values, e.g. ratings), *Classification* (the classification to one class out of a finite set of classes, e.g. ”recommend” and ”do not recommend”), and *Correlation* (computing the similarity between two ordered lists of items, e.g. user preferences and item features).

[1]stands for *Beats Per Minute*

Recommender systems can provide a useful alternative to common search algorithms since they help users to access items fitting to their preferences effortlessly, and even discover items they might not have found by themselves.

2.2 Historical Overview

The basic roots of recommender systems are derived from the fields of *Information Filtering* and *Information Retrieval* (Belkin and Croft, 1992) while alterations were introduced using methods from the field of *Data Mining*.

The first system (that would nowadays be called a *Recommender System*) was introduced by Goldberg et al. (1992) and was called *Tapestry*. (The authors used the phrase *Collaborative Filtering* (CF) and many others followed doing so for some years.) We prefer the more general term *Recommender System* as it was established later by Resnick and Varian (1997).

Only two years later the *GroupLens*[2] Group presented their first recommender system (Resnick et al., 1994) and the system *Ringo* (Upendra, 1994; Upendra and Maes, 1995) was launched. Both used findings and learnings propagated by Goldberg et al. (1992).

Fab presented by Balabanović and Shoham (1997) is considered the first hybrid RS combining a content-based filtering approach with traditional CF.

Although recommender systems were quite popular in the academic sector in the mid nineties, only the introduction of *item-based CF* by Sarwar et al. (2001) lead to a breakthrough in commercial systems as well. This approach was extended by Linden et al. (2003) for the usage in *Amazon.com*'s[3] Web portal. The success of this approach led to a wide acceptance of recommender systems. Moreover academic research was driven further as Lemire and Maclachlan (2005) introduced a set of *Slope One* predictors for online rating-based CF and provided an implementation of the algorithm (Lemire and McGrath, 2005).

Further topics of research were, among others, hybrid recommender systems (Burke, 2002), attacks in CF systems (Burke et al., 2006), explanations (Herlocker et al., 2000) and trust (Massa and Avesani, 2004).[4]

[2]`http://www.grouplens.org`, as of 2008/08/29
[3]`http://www.amazon.com`, as of 2008/10/07
[4]also see Section 2.6 on page 15 et seq.

2.3 Methodical Overview

In this section we give a brief introduction to the most common recommender methods.[5]

Adomavicius and Tuzhilin (2005) classify recommender systems into three different categories (*collaborative* recommendations, *content-based* recommendations, and *hybrid* approaches), based on how recommendations are made.[6] This section describes the formal recommendation problem and furthermore introduces methods for recommendations of the categories (just mentioned).

2.3.1 The Formal Recommendation Problem

The common formulation reduces the recommendation problem to the estimation of ratings for unseen items, based on the given user ratings to seen items and some additional information depending on the type of recommender system. Being able to estimate such ratings, a recommender system can then provide items with the highest estimated ratings to the user.

> ”More formally, the recommendation problem can be formulated as follows: Let C be the set of all users and let S be the set of all possible items that can be recommended, such as books, movies, or restaurants. (...) Let u be a utility function that measures the usefulness of item s to user c, i.e., $u : C \times S \rightarrow R$, where R is a totally ordered set (e.g., nonnegative integers or real numbers within a certain range). Then, for each user $c \in C$, we want to choose such item $s' \in S$ that maximizes the user's utility.”[7]

$$\forall c \in C, s'_c = \arg\max_{s \in S} u(c, s) \tag{2.1}$$

[5] We want to note that we found more than 250 publications concerning either CF or RS and therefore compiling a full list of all types of recommender methods ever introduced to the community absolutely exceeds the capacity of this thesis. Actually we believe that a RS for publications as well as improved visualisations, e.g. computing the correlation between publications (using methods from the fields of *Natural Language Processing* and *Data Mining*) could drastically reduce the amount of time required for the exploration of any research field.

[6] We would like to note that there exist different classifications of recommender systems in the literature (Resnick and Varian, 1997; Schafer et al., 1999; Burke, 2002), but we rather use the classification provided by Adomavicius and Tuzhilin (2005).

[7] cp. Adomavicius and Tuzhilin (2005, 734-735)

The spaces S (set of possible items) and C (set of all users) can be very large ranging up to millions of items and users. The utility u is usually presented by a *rating* in recommender systems. For each user (from the user space C) additional characteristics (e.g. personal or demographic information) can be added. Similarily all items are defined with a set of characterisics as well. Since the utility u is usually not defined ion the whole $C \times S$ space, it needs to be *extrapolated*, for e.g. estimating the rating value for unseen items.

2.3.2 Collaborative Methods

Collaborative methods basically recommend items to the user that users with similar tastes and preferences liked in the past, trying to predict the utility of items for a particular user.

> ”More formally, the utility $u(c, s)$ of item s for user c is estimated based on the utilities $u(c_j, s)$ assigned to item s by those users $c_j \in C$ who are ”similar” to user c. For example, in a movie recommendation application, in order to recommend movies to user c, the collaborative recommender system tries to find the ”peers” of user c, i.e., other users that have similar tastes in movies (rate the same movies similarly). Then, only the movies that are most liked by the ”peers” of user c would be recommended.”[8]

According to (Breese et al., 1998), algorithms for collaborative recommendations can be separated into *memory-based* and model-based approaches.

Memory-Based Algorithms Memory-based algorithms aim at predicting ratings of the active user based on previous user ratings from other users and the users preferences. It is assumed that the predicted rating value $p_{\text{u,j}}$ of the active user u for the unseen item j is a weighted sum of the ratings $r_{\text{u,j}}$ of other users $v_1, ..., v_N$. More formally:

$$p_{u,j} = \overline{r_u} + f \sum_{v=1}^{n} w(u, v)(r_{v,j} - \overline{r_v}) \qquad (2.2)$$

where $\overline{r_u}$ is the mean rating of user u, f is a normalisation factor and $w(u, v)$ is a weight decribing the similarity between users u and

[8]cp. Adomavicius and Tuzhilin (2005, 737-738)

v. The most common similarity measure based on user ratings is the *Pearson Correlation*. It computes a weighting from items i both users u and v have already rated and is defined as:

$$w(u,v) = \frac{\sum_{i=1}^{n}(r_{u,i} - \overline{r_u})(r_{v,i} - \overline{r_v})}{\sqrt{\sum_{i=1}^{n}(r_{u,i} - \overline{r_u})^2(r_{v,i} - \overline{r_v})^2}} \tag{2.3}$$

Other similarity measures for the computation of user weights are, among others, the *Vector Similarity*, the *Inverse User Frequency*, and the *Case Amplification*.

Model-Based Algorithms Contrary to memory-based algorithms only a subset of the user ratings are used by model-based algorithms to learn a *model* for the prediction of ratings. For the probabilistic approach an integer based rating scale from 0 to m is assumed. It is defined as:

$$p_{u,j} = E(r_{u,j}) = \sum_{k=0}^{m} Pr(r_{u,i} = k | r_{u,a}, a \in I_u) \tag{2.4}$$

Some examples are *cluster models* and *Bayesian networks*.

Additionally Sarwar et al. (2001) introduced a classification into *User-Based* and *Item-Based* collaborative recommendations, depending on the data correlations and similarities are computed from. Both types of methods are memory-based approaches using all known user ratings.[9]

Item-Based Algorithms Unlike item-based methods this approach computes item similarities based on the rating of users. The algorithms uses all rated items of user u to calculate similarities to the target item j. Hence the *Pearson Correlation* can be redefined (for item-based algorithms) as:

$$w(i,j) = \frac{\sum_{u=1}^{n}(r_{u,i} - \overline{r_i})(r_{u,j} - \overline{r_j})}{\sqrt{\sum_{u=1}^{n}(r_{u,i} - \overline{r_i})^2(r_{u,j} - \overline{r_j})^2}} \tag{2.5}$$

where u is the set of users that have rated both items i and j. This approach can also be applied to the cosine based *Vector Similarity* and adresses some of the common problem described in Section 2.5 (page 14 et seq.).

[9]Since several definitions mentioned above refer to user-based CF, we only describe *Item-Based* CF in the following paragraph.

2.3.3 Content-Based Methods

Content-Based methods basically recommend items to the user that are similar to those items the user preferred in the past. For the computation of item similarities the item's characteristics are represented as item profiles or item feature vectors. Such item profiles are often defined as vectors of weights $(w_{c1}, ..., w_{ck})$ where each weight $w_c k$ denotes the importance of a keyword k_i to the user.

Since item information is often represented as textual information, the methods used to compute these weights are derived from the fields of *Information Filtering* and *Information Retrieval* (Belkin and Croft, 1992). According to Adomavicius and Tuzhilin (2005) one of the best-know measures for specifying keyword weights is the *term frequence/inverse document frequency* (TFxIDF) measure:

$$w_{t,d} = TF_{t,d} \times IDF_t = \frac{f_{t,d}}{max_z(f_{z,d})} \cdot \log \frac{N}{n_t} \tag{2.6}$$

where the *term frequency* $TF_{t,d}$ describes the importance of the term t in the document d, while the *inverse document frequency* IDF_t computes the discriminant strength of a term (over all documents). $TF_{t,d}$ is defined as the frequeny of a single term in a document $f_{t,d}$ normalised by the maximum frequency of all terms in d. IDF_t is defined as the logarithmic ratio of the number of all documents N to the number of all documents containing term t, n_t.

Based on such feature vectors the utility function for content-based systems is usually defined as:

$$u_{c,s} = score(ContentBasedProfile(c), Content(s)) \tag{2.7}$$

where both profiles can be represented as TFxIDF vectors $\vec{w_c}$ (describing the user's profile containing tastes and preferences retrieved from items previously rated) and $\vec{w_s}$ (consisting of the item profile weights).

A simple but effective scoring heuristic is provided by the *cosine similarity measure*:

$$u_{c,s} = \cos(\vec{w}_c, \vec{w}_s) = \frac{\vec{w}_c \cdot \vec{w}_s}{\left|\left|\vec{w}_c\right|\right|_2 \times \left|\left|\vec{w}_c\right|\right|_2} = \frac{\sum_{\text{i=1}}^{K} w_{i,c} w_{i,s}}{\sqrt{\sum_{\text{i=1}}^{K} w_{i,c}^2}\sqrt{\sum_{\text{i=1}}^{K} w_{i,s}^2}} \tag{2.8}$$

where K is the total number of keywords over all documents.

Additional to traditional heuristics mainly based on information retrieval methods, a huge number of other methods have been used. Among others, there are systems using *Bayes classifiers*, *clustering* techniques, *decision trees* and *nearest neighbour* methods.

With a growing interest in recommender systems for multi-media content additional methods from the field of *Music Information Retrieval* have been incorporated into recommender systems. Such methods focusing on the extraction of feature vectors directly out of binary data (e.g. audio files) are classified as *Audio Feature Extraction* methods.

2.3.4 Hybrid Methods

According to Burke (2002) hybrid recommender systems combine various recommendation techniques (like *Collaborative Methods*, *Content-Based Methods*, etc.) to gain better performance with fewer of the drawbacks of the individual one. The type of combination of different methods can be classified into different approaches:

Hybridisation method	**Description**
Weighted	The scores (or votes) of several recommendation techniques are combined together to produce a single recommendation.
Switching	The system switches between recommendation techniques depending on the current situation.
Mixed	Recommendations from several different recommenders are presented at the same time.
Feature combination	Features from different recommendation data sources are thrown together into a single recommendation algorithm.
Cascade	One recommender refines the recommendations given by another.
Feature augmentation	Output from one technique is used as an input feature to another.
Meta-level	The model learned by one recommender is used as input to another.

Table 2.1: *Hybridisation Methods*[10]

[9]cp. Burke (2002, 337)

2.4 Evaluation Measures

As summarised by Herlocker et al. (2004) a large set of statistical measurements can be used for the evaluation of CF based recommender systems. This section summmarises the most popular measures and additionally introduces various public datasets.

2.4.1 Predictive Accuracy Metrics

The following measures are used to evaluate predicted numeric values (e.g. the rating value a user would give for an unknown item):

MAE The *Mean Absolute Error* measures the average absolute deviation between a predicted rating and the user's true rating. The mechanisms of computation are very simple. In Equation 2.9 N is the number of user ratings, $\{p_1, ..., p_N\}$ are the predicted rating values and $\{r_1, ..., r_N\}$ are the real rating values.

$$|\overline{E}| = \frac{\sum_{i=1}^{N} |p_i - r_i|}{N} \tag{2.9}$$

RMSE The *Root Mean Squared Error* emphasises larger errors since the error is squared before being summed up.

$$|\overline{E}| = \sqrt{\frac{\sum_{i=1}^{N} (p_i - r_i)^2}{N}} \tag{2.10}$$

NMAE As stated by Park et al. (2006) the *Normalised Mean Absolute Error* can be computed in two ways: *macro-averaged* and *micro-averaged*. While macro-averaged MAE calculates the mean absolute error of each user separately and averages over all users' averages, the micro-averaged MAE averages errors over all ratings. The MAE is normalised by the MAE of randomly selected predictions (MAE_{random}). Hence a NMAE smaller than 1 means that the algorithm works better than random.

2.4.2 Classification Accuracy Metrics

For the evaluation of classifiers (assigning instances of a dataset to distinct and disjunct classes) these statistical measures are used:

Precision Precision is defined as the ratio of relevant items select to the number of items selected. It is computed using a 2x2 table holding the number of relevant items selected (N_{rs}), the number of relevant not selected items (N_{rn}), the number of irrelevant selected items (N_{is}), and finally the number of irrelevant not selected items (N_{in}). In Equation 2.11 N_s is the total number of selected items. Precision represents the probability that a selected item is relevant.

$$P = \frac{N_{\text{rs}}}{N_s} \tag{2.11}$$

Recall Recall is defined as the ratio of relevant items selected to the total number of relevant items available, thus representing the probability that a relevant item will be selected. In Equation 2.12 N_r denotes the total number of relevant items. Since both measures (*Precision* and *Recall*) only measure binary relevance, they cannot measure the quality of ordering among selected relevant items.

$$R = \frac{N_{\text{rs}}}{N_r} \tag{2.12}$$

F_1 **Measure** For the combination of the measures *Precision* and *Recall* the F measure (or F_1 measure) is often used. It is defined as the ratio of the doubled *Precision-Recall* product to the sum of both measures.

$$F_1 = \frac{2PR}{P + R} \tag{2.13}$$

2.4.3 Prediction-Rating Correlation

In order to measure the correlation between two rating vectors these measures are used:

Pearson Correlation The *Pearson Correlation* (also refered to as *Pearson's Product-Moment Correlation*) measures the extent to which there is linear relationship between two variables (x and y). Besides using it to compute the similarity of two items or two users during the prediction of recommendations, it is widely used to measure the correlation between the estimated ratings and the true ratings (in a test dataset).

$$c = \frac{\sum(x-\overline{x})(y-\overline{y})}{n \cdot stdev(x) \cdot stdev(y)} \tag{2.14}$$

Spearman's ρ This rank correlation measures the extent to which two different rankings agree independent of the actual values of the variables. It is computed similar to the *Pearson Correlation* except that the variables x and y are transformed into the ranks u and v, thus the correlations are computed on the ranks.

$$\rho = \frac{\sum(u-\overline{u})(v-\overline{v})}{n \cdot stdev(u) \cdot stdev(v)} \tag{2.15}$$

2.4.4 Common Datasets

For the evaluation of CF algorithms a handful of public available datasets are provided over the Web and used in various publications. (Sarwar et al., 2000; Goldberg et al., 2001; Yu et al., 2001; Anderson et al., 2003; Ziegler et al., 2005; Park et al., 2006; Brozovsky and Petricek, 2007)

Movie Related

EachMovie[11] *HP/Compaq Research* (formerly DEC Research) ran the EachMovie movie recommender and provided this dataset, containing 2,811,983 ratings (from 1-5) entered by 72,916 users for 1,628 different movies. It has been used in numerous CF publications from the day EachMovie was shutdown (and the dataset was available to the public for use in research) until October, 2004 (when HP retired the EachMovie dataset). It is no longer available for download.

[11]*HP/Compaq Research* officially retired the dataset in October, 2004

MovieLens[12] The *GroupLens* Group currently offers two datasets. The first one consists of 100,000 ratings (from 1-5) for 1,682 movies by 943 users. The second one consists of approximately 1 million ratings (from 1-5) for 3900 movies by 6040 users.

NetFlix[13] *Netflix* introduced this dataset in October, 2006, simultaneously with the launch of the *Netflix Price*[14]. It contains over 100 million of user ratings on 17,770 movies, hence being considered the largest CF dataset available to the public.

Other Domains

Book-Crossing[15] The BookCrossing (BX) dataset was collected from the Book-Crossing community, CTO of Humankind Systems and published by Ziegler et al. (2005). It contains 278,858 users (anonymised but with demographic information) providing 1,149,780 ratings (from 0-10) for about 271,379 books.

Dating Agency[16] This dataset includes 17,359,346 anonymous ratings of 168,791 user profiles made by 135,359 users of the LibimSeTi[17] dating Web site. It was presented by Brozovsky and Petricek (2007).

Jester Joke[18] Goldberg et al. (2001) from the UC Berkeley, CA, introduced this dataset. It contains 4.1 million continuous ratings (from -10.00 to +10.00) of 100 jokes from 73,421 users.

Vote World[19] This dataset includes about 4 million ratings of 696 members of the *European Parliament* on 5745 roll-call votes. It was introduced by Simon Hix (2006).

[12] `http://www.grouplens.org/node/73`, as of 2008/08/30

[13] `http://www.netflixprize.com/download`, as of 2008/08/30

[14] Netflix rewards a $1,000,000 *Grand Prize* to the first team submitting an algorithm with a 10% improvement of the prediction accuracy and a $50,000 *Progress Prize* to the best team of the year, whose system shows the best improvement to the previous years accuracy bar.

[15] `http://www.informatik.uni-freiburg.de/~cziegler/BX`, as of 2008/08/30

[16] `http://www.occamslab.com/petricek/data`, as of 2008/08/30

[17] `http://libimseti.cz`, as of 2008/08/30

[18] `http://goldberg.berkeley.edu/jester-data`, as of 2008/08/30

[19] `http://ucdata.berkeley.edu:7101/new_web/VoteWorld/voteworld/datasets.html`, as of 2008/08/30

2.5 Common Problems

The drawbacks and limitations of recommender systems are described by Adomavicius and Tuzhilin (2005). Additionally Park et al. (2006) mention the *Cold-Start Problem* examined during the evaluation of a non-personalised baseline, a user-based CF system and an item-based CF approach compared to their own hybrid implementation.

The Cold-Start Problem This general problems occurs whenever a new CF based recommender service is launched leading to an extremely sparse user-item rating matrix. Since at that point the system has not collected enough ratings yet to produce meaningful recommendations. This problem can be adressed using content-based recommendations and hybrid systems.

The New-Item Problem Also an issue in CF recommender systems is the *New-Item Problem*. New items are not presented to the user by the RS, so they are unlikely to be found and rated at all. Similar to the *Cold-Start Problem* this problem only occurs in pure rating-based approaches, while content-based approaches and hybrids are more robust.

The New-User Problem The *New-User Problem* occurs in various types of recommender systems. Because the RS has no ratings from the new user and cannot learn the user's preferences, it cannot provide accurate recommendations for that user. Among others, some techniques to overcome this problem are based on the item popularity or the item entropy. Thus the most imformative items can be presented to the user.

Sparsity of Data With a growing size of underlying datasets of a RS data of user ratings becomes very sparse. Since there are often millions of items, users can only manage to rate some of the items leading to very sparse user- and item-vectors. Solutions to his problem are the measuring of user similarity using demographic data (like gender, age, area code, education etc.) or the reduction of dimensionality of sparse ratings matrices.

Limited Content Analysis This problem only occurs in strictly content-based recommender systems, since such systems are restricted to the set of the features associated to the system's items. In content-based systems the set of features (which is extracted automatically in the most cases) describes an item, thus two items with identical features are considered as equal and cannot be distinguished any more. Solutions to this problem are the adjustment of the featureset (to contain more details about items) and once again hybrid approaches.

Overspecialisation In pure content-based recommender systems this problem targets the specialisation on items that are *too similar* to the user's preferences. Although this might not look like a problem, since recommending the best fitting items according to user preferences is the main task of a RS, in some cases two items that are very similar simply represent different interpretations of the same item.[20] To overcome this problem items in the result set that are *too similar* are often filtered out.

2.6 Further Topics of Research

In this section we point out some of the further topics of research in the field of recommender systems.[21]

Attacks Since recommender systems are often attacked by external parties aiming to promote certain items (e.g. a business competitor of a Web shop owner) some publications focus on such attacks. Burke et al. (2006) propose and study different attributes derived from user profiles for their utility in attack detection. Furthermore Mobasher et al. (2007) evaluate how stable several recommender techniques are against different types of attacks. Testing different attacking strategies on a user-based CF algorithm and running a cost-benefit analysis on common recommender systems computing the *return-of-investment* (ROI) was done by Hurley et al. (2007).

Explanations Herlocker et al. (2000) point out the need of explanations in CF recommender systems, and describe the benefits of building an explanation facility into a RS. Moreover the advantages of explaining why a certain recommendation (e.g. list of items) was presented to

[20]An example: Consider a news recommendation service providing news feeds to the user. A content-based recommender approach could easily match the user's preferences to the features of a news report, but it would also recommend different articles from various authors containing the same redundant information.

[21]We want to note that, although there are quite many topics that are not mentioned in this section, we aimed at focusing on the most relevant for our work.

the user are introduced. These benefits are *justification* increasing the user confidence, *user involvement*, *education* and most important of all *acceptance*. Furthermore the authors evaluated the effectivness of different explanation models and techniques as well as the increase of acceptance and filtering performance due to explanations.

Trust In order to overcome some of the common problems of pure CF Massa and Avesani (2004) introduced a trust-aware recommender system that uses a, so called, *Web of Trust* defined by user ratings on other users. This hybrid system uses a trust function over the set of users with the range of [0,1] where 0 means total distrust and 1 total trust. A trust metric module exploits trust propagation in order to predict how much a user can trust another user, and a rating predictor combines CF user similarity with the value of trust. This approach has been further investigated in publications from the same author (Massa and Avesani, 2006, 2007) and by O'Donovan and Smyth (2005) as well.

Music-Related Systems Quite a large amount of recommender systems are used in the domain of *music* (e.g. in various Web portals). These (mainly hybrid) recommenders provide collaborative filtering and often incorporate content-based approaches (Hoashi et al., 2003; Gstrein et al., 2005; Gstrein and Krenn, 2006).

Besides *audio feature extraction* methods for the gathering of item profile information (Li et al., 2004; Cano et al., 2005; Gstrein et al., 2006; Hlavac et al., 2007), further methods from the field of *Music Information Retrieval* (MIR), like content-based music organisation and indexing methods (Rauber et al., 2002; Pampalk et al., 2002) and Web mining methods to compute artist similarity (Knees et al., 2004) or text mining methods over collected lyrics for the computation of genre and track relations (Kleedorfer, 2008; Kleedorfer et al., 2008), can be adapted for the usage in recommender systems.

Generally spoken, since many sources contain music related metadata various methods can be useful to access different information in order to improve the quality of recommendations.

Summary

After we presented a short description of the basic functionality and the historical background of recommender systems, we formulated the recommendation problem and introduced a classification of recommender techniques. We furthermore separated collaborative filtering methods operating on a set of user ratings into the different categories: memory-based algorithms taking all ratings into account, model-based algorithms aiming to improve scalability and robustness predicting recommendations only on a subset of user ratings, and furthermore item-based algorithms computing item list due to item similarity and user preferences. We moreover described that pure content-based techniques rely on the matching of similar aggregated user profiles and extracted or defined item feature vectors. Additionally we introduced hybrid recommender systems explaining the different type of hybridisation methods. Moreover we presented a short survey of various evaluation measures for prediction, classification and correlation as well as a list of public available datasets. Further we discussed common problems of recommender systems like the cold-start problem, sparsity or limited content analysis. Finally we gave a small insight in the further research topics: attacks, explanations, and trust and mentioned some music-related systems and useful methods from the field of MIR that are capable of being adapted for the use in recommender systems.

Chapter 3
Existing Recommender Frameworks

Abstract

In this chapter we describe some state-of-the-art recommender frameworks as well as various recommender systems that are available for application developers. Section 3.1 introduces some recommender archtitectures found in the literature, while Section 3.2 presents a commercial recommender framework. Finally Section 3.3 gives a brief overview of some open-source CF recommender frameworks.

3.1 Research Projects

Most publications about recommender systems either describe a novel approach or some necessary adaptations on well-known recommendation algorithms in order to improve the quality of recommendations. Other papers summarise over existing methods or just focus on a very specific issue. Nevertheless we want to point out some publications that combine different recommendation approaches aiming to design a generic RS.

The hybrid system *RACOFI* (Rule-Applying Collaborative Filtering) presented by Anderson et al. (2003) seperates metadata into two classes: *objective* (i.e. content information) and *subjective* metadata (i.e. user ratings). It collects ratings for different characteristica of an item, rather than for the item itself. The subjective metadata is handled using a CF algorithm, while the objective metadata is processed with a rule inference system that produces recommendation rules in XML form, which are then applied to the predictions to customize the recommendations according to user profiles. The authors state that the system is domain-independent and can easily be applied to any domain, although the reference implementation[1] is restricted to Canadian artists. In a following publication (Lemire et al., 2005) the authors further describe the usage of item-based CF algorithms, as proposed by Sarwar et al. (2001); Linden et al. (2003);

[1] `http://racofi.elg.ca`, as of 2008/10/11

Lemire and Maclachlan (2005). The improved system is used for recommendations in a music Web portal.[2] Finally the authors mention the importance of learning knowledge about objects and that the rule-based approach made it easy to adapt the system to user expectations.

Similar the hybrid system introduced by Hedfi and Trabelsi (2005) uses data mining techniques for clustering the user space (according to the users purchase behaviour) and association rule mining for the detection of affinities between items. Hence recommendation lists for each user cluster and for each user can be computed separately. For the prediction of rating values only similarity values between users within the same cluster are computed using a user-based correlation algorithm.

Rack et al. (2007) present a generic recommender system that provides contextual recommendations based on the combination of previously gathered user feedback data (i.e. ratings and clickstream history), context data, and ontology-based content categorisation schemes. For the access of information, so called, *data adapters* are used that need to register at the *profile manager*, while a *profile broker* matches profiles with the given query. The recommender component is held generically providing two methods: one for the input of user feedback on content items and another to answer whether an item is relveant in a given context or not. All learning algorithms and prediction methods implement the same interface providing these two methods. This generic approach seems to be very flexible concerning new recommender algorithms and different data sources, but its drawback is that all situations available to learners must be defined as explicit profiles.

Li et al. (2007) propose a *Service Oriented Architecture* (SOA) with multiple layers for: clients, presentation purposes, services, service management, resource management, and partners. The main services are: a user analysis service processing access sequences and user feedback with semantic mining algorithms into, so called, *user interest arrays*, and a *recommendation engine* that matches such interests with available products and services registered at the *service registry*. The system uses the TFxIDF measure (explained in Section 2.3.3 on page 8) to retrieve keywords from textual descriptions of available products and services, and provides both products and services as *Web Service Definition Language* (WSDL) structures. Additionally the authors state that the architecture offers assistant functions for: authentication, authorisation, security, etc..

[2]`http://www.indiscover.net`, as of 2008/10/11

3.2 Commercial Frameworks

Finding a generic commercial recommender framework proofed to be quite a hard task, since almost every commercial recommender system is integrated into an application scenario bound to a specific domain (e.g. music or video).

Nevertheless the company *Loomia*[3] provides a peronalized recommendation system claiming that it can be easily integrated. According to the Web site the system works for any kind of site and almost any type of content provided via a Web feed. A simple javascript widget sends user preference data to the *Loomia Similarity Engine*, which matches the provided content with the user's preferences, and generates recommendations within the customer's site.

Additionally a Web service API is offered that is designed to integrate with mobile devices.

Although there is no information on the used recommender technique, we consider this framework as quite interesting since it seems to be very flexible according integration issues as well as content management, and provides recommendations as managed services using Web services.

3.3 Open-Source Frameworks

The following list presents a number of open-source recommender systems and CF frameworks:

Taste: Collaborative Filtering for Java[4] A fully *Java*-based CF engine that supports memory-based CF methods. Both user-based and item-based implementations are provided. It does neither currently support model-based nor content-based techniques. Anyway, *Taste* offers data connectors for file and database access as well as a Web service interface and enterprise *Java* beans support. The author, Sean Owen, has announced plans to merge with the *Apache Mahout*[5] project and released the last version in April 2008.

[3]`http://www.loomia.com`, as of 2008/10/11
[4]`http://taste.sourceforge.net/`, as of 2008/08/15
[5]`http://lucene.apache.org/mahout/`, as of 2008/08/15

CoFE: Collaborative Filtering Engine[6] Another *Java*-based CF engine that just supports a user-based memory-based implementation using *Pearson Correlation* (explained in Equation 2.3 on page 7) on a set of user ratings. User data is stored in a database and recommendations are provided via a client-server architecture. The interface for algorithms is quite small, but allows both recommendations for a user with and without specifying an item type. This engine was developed at the *School of Electrical Engineering and Computer Science*[7] by a group of students supervised by Jonathan L. Herlocker associate professor at the *Oregan State University*[8] in 2004.

Cofi: A Java-Based Collaborative Filtering Library[9] This *Java* framework introduced by a group around Daniel Lemire professor at the *University of Quebec at Montreal*[10] was originally developed for the RACOFI recommender system, and was made public in 2004. It provides a set of different user-based and item-based methods, as well as parsers for the three well-known datasets: EachMovie, Jester, and MovieLens[11].

Vogoo: Recommendation Engine and Collaborative Filtering[12] The *Vogoo* PHP framework includes two item-based and one user-based recommendation engines and allows for the computation of user similarities. It supports multiple item categories and automatic ratings based on purchases and page views. Stéphane Droux developed this engine and released the first version in April 2005 and the current version in March 2008.

[6] `http://eecs.oregonstate.edu/iis/CoFE/`, as of 2008/08/15
[7] `http://eecs.oregonstate.edu/index.html`, as of 2008/10/11
[8] `http://oregonstate.edu`, as of 2008/10/11
[9] `http://www.nongnu.org/cofi/`, as of 2008/08/15
[10] `http://www.uqam.ca/`, as of 2008/10/11
[11] also see Section 2.4.4
[12] `http://www.vogoo-api.com/`, as of 2008/08/15

AURA: Advanced Universal Recommendation Architecture[13] This *Sun Microsystems*[14] project is currently under development by a group around Paul Lamere. The novel approach introduces a so called *textual aura* for an item, that can be gathered from different sources like social tags. Item similarity is computed using the textual similarity of the items' textual auras. Due to this approach explanations can be provided to the user increasing transparency. Futhermore this project aims to combine the advantages of content-based recommendations for sparse user data and collaborative filtering for lots of user data. A set of Web services is planned to enable clients to contribute data from their customers and receive recommendations for their customers.

Summary

In this chapter we introduced some recommenders that seem to incorporate a generic approach or a hybrid system with promising features. Some research projects used data mining methods for clustering and association rule mining. While others define generic interfaces in order to integrate different recommender techniques, or use a WSDL description for items extended with extracted keywords using the TFxIDF measure. Futhermore we presented a commercial recommender framework that claims to be easily integrated using a simple javascript widget communicating with a managed recommendation service via Web services. Finally we listed some open-source CF frameworks offering various memory-based algorithms as well as a novel approach using a *textual aura* for item similarity computation that is currently under development.

[13]`http://research.sun.com/projects/dashboard.php?id=196`, as of 2008/10/11
[14]`http://www.sun.com/`, as of 2008/10/11

Chapter 4
Concept

Abstract

Based on the Chapters 2 and 3 we identify the features in Section 4.1 and design our concept in respect to these features. Section 4.2 briefly describes the basics of the concept while Section 4.3 explains the components of the RS in detail and points out their advantages.

4.1 Identified Features

During our research in the field of recommender systems we identified the following challenges for a generic RS (listed in alphabetic order).

Applicability The RS shall be easily applicable for several scenarios, thus the effort necessary to integrate the RS into existing applications shall be minimised.

Domain-Independence Recommendations should not rely on specific domain knowledge but shall allow the integration of domain-specific metadata as well.

Extensibility The RS shall allow for the integration of different types of recommender algorithms. As described in Section 2.3 (page 5 et seqq.) different recommender systems use various algorithms that range from *Collaborative* to *Content-Based* techniques and many more. A great number of different types of algorithms shall be supported.

Multi-Tenant Capability Recommendations for different tenants shall be provided using one single recommender instance. The effort to integrate a new tenant shall be minised.

Performance and Scalability High quality recommendations shall be delivered to the user in just a few seconds.[1] The system performance

[1]An example: Consider a Web portal as client application where users can browse through the customer's content. The recommendation of best-fitting items for the user

shall not decrease drastically for large data sets (decreasing slightly is expected, since the response time for database queries is increasing with the amount of entries).

Stability The RS shall resist requests from many client applications running as a *24/7 Service*.[2]

Transparency It shall be possible to provide an explanation to the users why items are recommended to them. Furthermore business rules used for generating recommendations shall be transparent to system operators and optionally managed explicitly during a regular maintenance process.

4.2 Concept Overview

In detail the recommender concept developed in this thesis consists of the following functionalities:

- Provide the capability to introduce different types of state-of-the-art recommendation techniques (like *Shopping Cart Analysis*, *Item-Item* recommendations or *Item-Based CF* methods) as well as importing third-party metadata.
- The generation of item-to-item relations is done *offline*[3] (as a background process) and stored explicitly (and persistently) as *Item Association Rules* to ensure stability and performance.
- The core recommendation strategies focus on these pre-calculated item-to-item relations so that the *online* recommendation requests will only query pre-computed information.
- Additionally *online* recommendation algorithms are supported as well.

shall not delay the user's behaviour. The user should not even notice the computational amount of time used to generate recommendations.

[2]24/7 is an abbreviation which stands for ”24 hours a day, 7 days a week”

[3]We use the terms *online* and *offline* as introduced by Linden et al. (2003) to state whether the computations necessary for the generation of recommendations are run live (just when the user requests information) or in advance in a background process (to be retrieved fast and easily whenever a recommendation is presented to a user).

4.3 Components

In order to cover all identified features for the planned RS, the concept was designed to be very flexible but aims to keep the whole system as simple as possible. A description of the main components of this concept is provided in the following.

4.3.1 Items and Item Types

Considering a single customer the main object of recommendations is a set of *Items* from a specific domain (such a set is further called *Content Pool*). Since we aim to service multiple tenants for the RS, the management of such *Content Pools* is essential. In many client applications the *identifiers* (IDs) for each *Item* are not overall unique but only unique over the same type of items. Thus two different items (e.g. a ”track” and an ”artist”) may have the same item ID. As a consequence one can either treat each item type separately from the other types, or disambiguate the duplicate IDs by adding an additional type code (for example an extra field `itemType` in the *Action* database table) which is what we prefer. Such item types are also tenant-specific. Furthermore the RS does not explicitly store *Items* of interest, but rather stores IDs of those items.

4.3.2 Actions and Action Types

Since many recommendation methods rely on different types of user actions one of the major components of this concept are such *Actions* and their persistence to a storage. Such user *Actions* are generally of the form ”User U performs action A upon item I at a given date D” (e.g. ”User U buys item I at 2008/12/23”). To allow a wide range of user *Actions* their structure is designed in a generic way. The types for *Items* and *Actions* are not predefined, which would restrict them to a specific set, rather such types are individually managed for each tenant.

Each *Action* consists of the following attributes:

- *tenant identifier*: An identifier for the tenant (providing sets of items and users). This ID allows the user actions to be separated for each tenant.
- *user identifier*: An identifier for the user (of the given tenant) who acted within the client (e.g. clicked on a specific link in a Web ap-

plication). This ID (together with the tenant ID) distinctly identifies a user.

- *item identifier*: An identifier for the item (of the given tenant) the user acted on.
- *item type identifier*: An identifier for the item type (e.g. ”track”). The item type ID combined with the ID of the item itself and the ID for the tenant uniquely describe an item. (An item ID without corresponding item type ID cannot be assigned to a specific item.)
- *action type identifier*: An idenfier for the action type (e.g. ”view”).
- *action time*: A timestamp of the moment when the action was recorded. Tracking the time when the action occured implies an ordering of user actions, thus allowing recommender methods to use that timestamp for the weighting of user actions.[4]

Concerning recommendations all strategies relying on user *Actions* can be applied - ranging from association rule mining (shopping cart analysis) and various clustering techniques to item-item associations (e.g. ”Users who bought X also bought Y”, ”If you like X you might like Y” etc.).

4.3.3 Generators

Since there is a set of recommender algorithms that take quite a long time to compute a recommendation for the user, the concept provides so called *Item Association Generators* which operate *offline*. The computation of item associations is executed (in advance to user requests) in a periodical interval using background processes to generate and manage such associations.

Additionally *online* recommender methods that are capable of delivering a recommendation in just a few seconds (e.g. rating-based CF methods) for ad-hoc recommendations can optionally persist their results too.

All *Item Association Generators* operate on a minimal data set of user *Actions* and generate explicit *Item Association Rules*.

[4]An example: Recommendations for a user who used to listen to a specific music genre earlier in the past but has switched his/her preferences to another genre can be computed using that information, hence providing more items the user acted on in the near past than items acted on earlier.

4.3.4 Item Association Rules and Association Types

Item Association Rules describe an association between two items and are of the form ”Item I has association A with item J by the value of V” where item I is called *antecedent* and item J is called *consequent*. For the association A an *Association Type* is introduced, which describes different forms of associations ranging from similarity (e.g. ”is similar to”) over composition (e.g. ”is part of”) to usage relations (e.g. ”viewed together”, ”bought together”, etc.). Since item types are generic, a *User* can be considered as *Item* as well allowing associations between users and items (e.g. ”likes”). Similar to item and action types the possible types of associations are also tenant-specific. Since *Item Assocation Rules* (IAR) are stored explicitely in a database they can be requested in a fast and easy way without the need for further calculations, thus allowing recommendations to rely on precalculated item-item associations.

As these rules describe directed relations between items, the following information must be provided:

- *tenant identifier*: Describing the *owner* of the rule.
- *identifier and type of item I*: The antecedent of the rule consisting of item I and its type.
- *identifier and type of item J*: The consequent of the rule consisting of item J and its type.
- *association type and value*: The kind and strength of the association to express the quality of a specific item-item relation.
- *view type*: The view origin of this rule is stored, either an *Adminisitrator* (or system operator) has managed a rule by hand, or the rule was computed from *Community* ratings or the origin was any other (third-party) *System* (accessing item metadata to generate item association rules).
- *source information*: For a more fine-grained distinction of the origin. This data will be used in cases where different information sources (for the same view) are used to generate rules. For example when metadata of two different third-party providers is used to generate rules. Furthermore this information can be used for explanation purposes.
- *timestamp*: When the rule was generated (or updated).

The explicit storage of the rules is one major component of the recommender concept mentioned in this thesis, because:

- performance, scalability, and stability are ensured since time consuming algorithms are operating as *offline* generators.
- third-party information can be imported.
- administrators can manage rules according to their policy, so quality assurance is easier since improving *online* recommendations would mean improving the quality of item metadata.

Based on the fact that the recommender does not store any metadata for users and items (additional to persisting simple IDs), metadata filters cannot be applied. Nevertheless it is possible to introduce content-based item association generators (e.g. audio-based[5] or web-based feature extraction methods) that are in charge of gathering domain knowledge (metadata) from other sources.

4.3.5 Non-Personalised Recommendations

Experiences, we made during the past years of our cooperation with a large music portal operator, show that many users do not want to support explicit profiling (e.g. a user registration combined with the input of user preferences). Furthermore most users (of Web portals) are currently ”walk-in” customers visiting the personalised application two or three times which reduces the value of profile-refinement dramatically. Sparse user profiles as well as poor item metadata lead to recommendations of unsatisfying quality. This is why this approach does not support personalised recommendations by means of explicit user profiles. Since there are no explicit profiles available, users and items are only identified via tenant-specific IDs. The focus of recommendations lies on the gathering of user *Actions* and aggregating them to implicit user profiles and preferences (user-item) as well as item relations (item-item).

[5]*An example*: An *audio feature extraction* technique could work as an item association generator extracting ”sounding similar” associations for audio tracks from the raw audio files and providing newly generated item association rules of the form ”Item A is sounding similar to item B by the association value V”, describing how much two items (A,B) are sounding similar.

4.3.6 Management of Multiple Tenants

To provide the services of the RS to many different applications of various tenants, the management of such tenants is quite important. Each tenant can have specific sets of users, items and actions (performed within the tenants client applications). This ensures that user actions of different recommender customers are not affecting each other. This affects the handling of the data as well as the generation of rules.

The main impacts are that:

- all user *Actions* must be supplemented with a tenant identifier.
- the rule generation must rely on data related to a specific tenant.
- all *Item Association Rules* are extended with tenant information.
- each tenant uses its own set of types for *Actions*, *Items* and *Item Associations*.

Although multi tenant capability is more a technical than a conceptual problem, some side effects can be used to improve the recommendation experience by using appropriate - tenant-specific - rules for other tenants too (if applicable[6]).

4.3.7 Minimising Complexity

Our recommender approach reduces the complexity to a minimum by splitting the recommendation process in several loosely coupled and independent operating components.

Most of the intelligence of the system is incorporated in *offline* performing generators each fulfilling a specific task.

This modularisation:

- minimises complexity at the algorithmic level.
- offers the possibility to combine different strategies easily.
- leads to low maintenance costs.

[6]Tenant comprehensive recommendations can only be applied if the item sets (content pools) of those tenants are overlapping.

- introduces the capability to extend functionality by simply adding new generators.
- introduces stability because generator crashes do not harm the *online* system.
- supports scalability by distributing the generators to different physical machines.

Recommendations are computed based on *offline* generated and explicitly stored rules. This approach addresses performance and scalability topics.

4.3.8 Post-Filtering

In order to improve the basic quality of the recommender our recommendation services use a standard approach for filtering duplicates and filtering based on the users *Action* history (e.g. do not recommend those items a user has already seen or bought).

4.3.9 Direct and Aggregated Ratings

Since gathering as much user information as possible is a common approach in recommender systems, many client applications store user *Actions* rather than direct user *Ratings*. This allows multiple action entries for a *User-Item* tuple[7] while classical rating-based CF algorithms expect only one single rating entry for each user-item pair[8].

Consequently an action-based recommender using rating-based algorithms must provide methods to map (or aggregate) a set of actions into a single rating.

We therefore differentiate between so called *Direct Ratings* and *Aggregated Ratings*. *Direct Ratings* are simply the average rating value over all explicit rating *Actions* of a user for a specific item.[9] For *Aggregated Ratings* nominal user *Actions* on a specific item can be individually mapped to

[7]*An example*: Consider a user who visits a client application two consecutive days, to listen to his favourite song.

[8]*An example*: Consider a user who rates an item twice, but only the last rating is persisted or taken into computations.

[9]*An example*: A specific tenant (e.g. owner of a Web shop) provides actions with the types ”buy”, ”view” and ”rate”. For a *Direct Rating* only actions with type ”rate” are taken into computation, and an average value is computed.

specific rating values (with an underlying ordinal ranking) and then aggregated into a single rating value using a defined aggregation strategy out of a set of different aggregation methods.
The following aggregation strategies are proposed:

- *Average*: Compute an arithmetic mean (over actions of a user for an item). This strategy's advantage is that more actions mapped to a low rating value (e.g. a set of *search* actions) lead to lower ratings, while a single action mapped to a high rating value (e.g. a *buy* action) increases the rating value. But note that this approach requires an implicit ordering of the mapped actions.
- *Maximum*: Use the highest rating. This strategy also requires an implicit ordering of the mapped actions.
- *Most Frequent*: Count the number of occurrences of each user *Action*, and use the most frequent.
- *Newest*: Use the newest action found (according to the action time).
- *Oldest*: Use the oldest action found (according to the action time).

Though we provide a set of strategies for the aggregation of *Actions* to *Ratings* we restrict the usage of the strategy to one for each tenant, since mixing up aggregated rating values that have been computed with a different strategy would result in false recommendations. The best-fitting action to rating mapping and the optimal strategy for a tenant must be evaluated individually over the tenant's set of *Action Types* and the user *Actions* of that tenant.
Since we deal with aggregation here it has to be clear that this mapping is not bijective, in other words one can not compute the original actions (of a user) when given only the final rating value.

Example for the Aggregation of Ratings

Let us consider a music download platform as client application which uses a set of user *Actions* with the mapping (using a rating scale that ranges from 1 to 10) described in Table 4.1 (page 32).
If a user would first search for a specific song, then view the track page, and finally listen to the song (a *preview* action), the aggregated rating value using for the *Average* strategy would be 4. While the purchase of the track (a *buy* action) using for the *Newest* strategy would result in a rating value of 10.

User *Action*	Description	Mapped Rating Value
search	search for a track	2
view	view the track page	4
preview	preview a snippet of the track	6
add to playlist	add the track to the user's playlist	7
rate good	rate the track with a good rating[10]	7
buy	purchase the track online	10

Table 4.1: *Example for the Mapping of Actions to Discrete Rating Values*

Summary

In this chapter we pointed out the necessary features for the RS and the concept that fulfils all identified needs. We explained that user *Actions* are collected, and can be used directly for various *online* recommendations or serve as computational basis for several different *offline* generators which produce *Item Association Rules*. Additionally we mentioned the capability of multiple tenants, and the necessary types of *Actions*, *Associations*, and *Items*. Furthermore we demonstrated the advantages concerning the reduction of complexity and mentioned some basic post-filtering techniques. Finally we stated the difference between *Direct Ratings* and *Aggregated Ratings*.

[10]As described in Section 5.5 (page 45 et seqq.) the decision whether a rating action can be considered as *good* or *bad* is depending on the tenant-specific rating scale.

Chapter 5
Technical Realisation

Abstract

This chapter describes the RS from a technical point of view. Section 5.9 and Section 5.2 point to a preliminary evaluation in Appendix A and the detailed SW infrastructure in Appendix B. An overview of the RS architecture is illustrated in Section 5.1, while Section 5.3 and Section 5.4 reveal the detailed SW design. Furthermore Section 5.5 describes the underlying data model. Finally Section 5.6 provides on overview of the deployment architecture and Section 5.7 explains the supported recommender features of the RS. Additionally Section 5.8 points to the detailed description of the Web service interface in Appendix C.

5.1 Architecture Overview

As pictured in Figure 5.1 (page 34) the RS is a composition of a *Recommender Server*, a *Generator Server* and various *Client Applications*.

The *Recommender Server* is designed as a multi-layered architecture consisting of a *Database Layer* for the access of user *Actions* and item *Association Rules*, an *Application Layer* for *Online* and *Offline Recommender Services*, and an *API Layer* for various *Web Services*. Moreover the *Generator Server* contains different (so called) *Item Association Generators* which create business rules that define a relation between two items. Additionally the architecture allows the import of *Third-Party Metadata*.

Technically, a *Client Application* tracks user *Actions* and sends them to our *Recommender Server* via a *Web Service* call. Those *Actions* are then stored in the database. The design of our RS allows both *online* and *offline* recommendations. *Online* algorithms use *Actions* directly to compute *Recommendations* for the user, while time-consuming methods run as *offline Generators* that produce item associations which are then queried and post-filtered (with filtering methods, e.g. history filtering, duplicate filtering etc., using the present user *Actions*) to present meaningful *Recommendations*. Additionally *Offline Recommender Services* can make use of *Third-Party Metadata* provided as IAR. *Recommendations* are again provided to the *Client Application* via several *Web Service* methods.

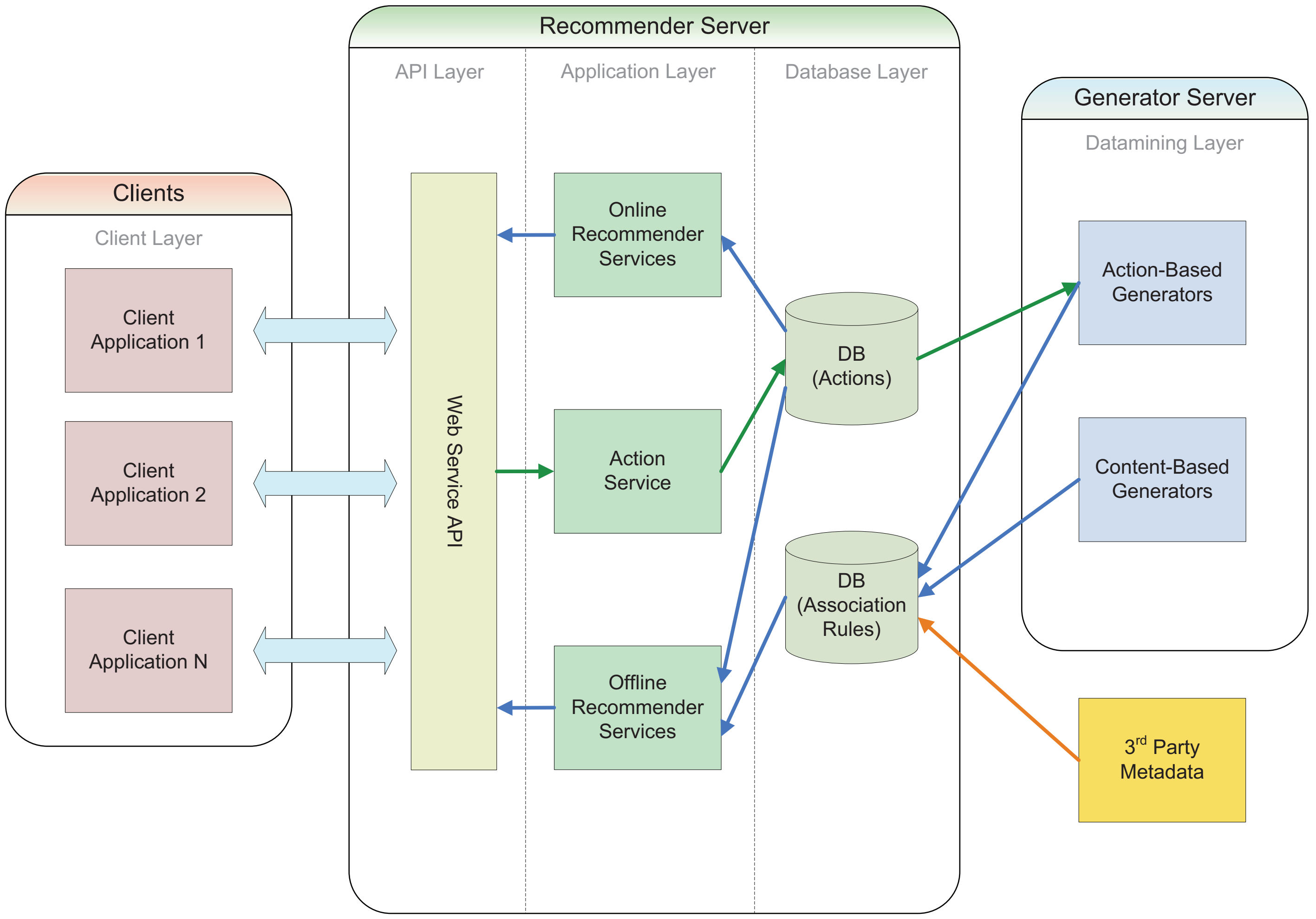


Figure 5.1: *Overview of the Software Architecture*

5.2 Software Infrastructure

This project is fully java-based and uses some open-source *Application Programming Interfaces* (APIs). A full description of the SW infrastructure can be found in Appendix B (page 87 et seqq.).

5.3 Modules and Packages

The complete software consists of 156 *Java* classes and interfaces, a number of test classes, and a set of resource files. In order to provide portions of source code and resource files belonging together, the complete RS is divided into six units which separate the application into organisational modules. Additionally dependencies to third-party artefacts can be managed separately for each module.

These modules are:

- *Core*: This module contains model objects, classes for database access, and basic services like the `ActionService`, the `ItemAssocService`, the `RecommenderService`, and the `RecommendationHistoryService`. All interfaces and classes in this module provide generic methods, thus domain information like action or item types can be parameterised individually. Table 5.1 (page 36 et seq.) describes the subpackages for the *Core* module.

- *Domain*: This package is introduced for the isolation of domain-specific content providing services and database access classes for a generic domain as well as for specific domains like the music domain. Additional interfaces for the access of third-party data and a utility class for the aggregation of actions to ratings are included. The list of subpackages for the *Domain* module is presented in Table 5.2 (page 37 et seq.).

- *Content*: The *Content* module holds online and offline item association generators. An example for an online generator is the `RascalloModellingGenerator` while the `AssocRuleMiner` serves as offline generator. The subpackages for the *Content* module are specified in Table 5.3 (page 38 et seq.).

- *Thirdparty*: This module consists of model objects and classes for the migration of third-party data thus providing classes for the import of third-party information to a SAT recommender database and for the export of such a database to other formats like the `taste_preferences` table. Table 5.4 (page 39) pictures the subpackages for the *Thirdparty* module.

- *Evaluation*: This module is dedicated to the evaluation of recommender algorithms. It contains classes used during the evaluation of the *Taste* `SlopeOneRecommender`. The subpackages for the *Evaluation* module are described in Table 5.5 (page 39).
- *Webapp*: Concerned with the propagation of various Web service methods of the RS the *Webapp* module provides model objects, data access classes, and services for various domains. Additionally an `IDMappingService` allows for the mapping of external `String` IDs to internal `Integer` IDs. Furthermore the `AuthenticationService` manages the access of Web service methods for several tenants. In Table 5.6 (page 40) the subpackages for the *Webapp* module are explained.

The main package for the RS is `at.researchstudio.sat.recommender`. This section furthermore describes the subpackages for each module.[1]

Table 5.1: *The Package Structure of the* Core *Module*

Package	Description
`model.core`	Contains model objects of the core recommender. These objects have generic type parameters that are used to represent IDs and types with different characteristics (e.g. `Integer` on core level and `String` on web service level)
`model.core.transfer`	Contains model objects that are mainly used to transfer constraints like the `TimeConstraintVO`.
`service`	Provides several generic interfaces of recommender core services.
`service.core`	Contains specific core service interfaces bound to the specific type `Integer` for IDs and types.

[1]Generally spoken, for each *Java* interface exists at least one *Java* implementation class. Since interfaces and classes are not located in the same subpackage, but rather the implementation classes are stored in a `.impl` subpackage, we omit these subpackages in the overview description of subpackages.

The package structure of the Core *module (continued)*

Package	Description
`store.dao`	Provides several generic interfaces to access data objects (from the database) that represent the recommender business model.
`store.dao.core`	Consists of several interfaces to access data objects (from the database) that represent the recommender business model. As type for IDs and types `Integer` is used.
`store.dao.core.types`	Holds interfaces and classes to access data objects (from the database) that contain type information.
`util.core`	Provides utility methods for the filtering of duplicates or filtering based on the action history.

Table 5.2: *The Package Structure of the* Domain *Module*

Package	Description
`io.domain`	Contains classes for the output of domain-specific information about an item.
`service.domain`	Provides domain-specific service interfaces and classes. Several types are using a `String` representation, while IDs (of items, tenants and users) are still using the `Integer` type.
`service.domain.music`	Holds service interfaces and classes for the music domain. Several methods are no longer type independent but rather action and item types are encoded in the method name thus the services are not parameterised. Hence such types can be omitted.
`service.domain.rascalli`	Similar to the music subpackage this package contains service interfaces for the RASCALLI domain.

The package structure of the Domain *module (continued)*

Package	Description
`store.dao.domain`	Consists of several interfaces to access data objects (from the database) that represent the recommender business model. As type for IDs `Integer` is used, while a `String` representation is assigned for each type defining a tenant-specific mapping for theses types.
`store.thirdparty`	Contains an interface that is related with the access of a third-party storage, for the output of domain-specific information about an item.
`util.domain`	Provides domain-specific utility classes for the recommender like the `ActionToRatingAggregator`.

Table 5.3: *The Package Structure of the* Content *Module*

Package	Description
`generator`	Provides the interfaces necessary to create item association rules for the various domains. These item association rules hold domain-specific knowledge for tenants, users and sometimes sessions.
`generator.arm`	Contains classes and interfaces to create item association rules using the `AssocRuleMiningService`. This generator operates offline and generates item-item associations similar to a shopping cart analysis approach.
`generator.arm.cli`	Offers a command line interface for the `AssocRuleMiningService` which can be used to run the service in a separated background process.
`generator.arm.model`	Consists of model objects used for the `AssocRuleMiningService`.
`generator.arm.store.dao`	Holds an interface that provides generator specific access methods for `ActionVO` objects (from the database).

The package structure of the Content *module (continued)*

Package	Description
`generator.rascalli`	Provides the classes necessary to create item association rules for the RASCALLI domain. The `RascalloModellingGenerator` as well as two different strategies for it (`PeaCounter` and `WeakeningOverTime`) are introduced, which operate mainly as online generators, but are able to persist generated rules as well.

Package	Description
`model.thirdparty`	Provides model classes to hold data during import or export of third-party data.
`service.thirdparty`	Contains interfaces and service classes that are necessary for the import of third-party data to a SAT recommender database as well as vice versa for the export of SAT data to foreign databases.
`store.dao.thirdparty`	Holds interfaces and data access classes for the import and export of data.

Table 5.4: *The Package Structure of the* Thirdparty *Module*

Package	Description
`evaluation`	Concerned with the evaluation of recommender algorithms containing a class used during the evaluation of the `SlopeOneRecommender`.

Table 5.5: *The Package Structure of the* Evaluation *Module*

Package	Description
`model.webapp`	Contains model objects that are used within the web service layer of the RS. These objects are simpler than the internal objects (which contain type parameters and more fields).
`service.webapp`	Provides several service interfaces and service classes of the Web service layer of the RS such as the `AuthenticationService` and the `IDMappingService`.
`service.webapp.music`	Provides a convenience Web service interface for the music domain.
`service.webapp.music.exception`	Contains an `Exception` for the web service interface for the music domain. [2]
`service.webapp.nodomain`	Provides a domain-independent Web service interface.
`service.webapp.nodomain.exception`	Holds an `Exception` for the domain-independent Web service interface.
`service.webapp.rascalli`	Provides a convenience Web service interface for the RASCALLI domain.
`service.webapp.rascalli.exception`	Contains an `Exception` for the web service interface for the RASCALLI domain.
`service.webapp.rascalli.model`	Provides model objects that are used within the RASCALLI specific Web service interface. These objects are mainly used for caching purposes.
`store.dao.webapp`	Offers interfaces and classes to access data objects (from the database) that are used for web service propagation (e.g. the `AuthenticationDAO`).
`util.webapp`	Contains web service specific utility classes like the `WebAppPathHolder` for the configuration of the profiling aspect `JamonProfilingAspectAdvice`.

Table 5.6: *The Package Structure of the* Webapp *Module*

Additionally some utility classes were introduced and integrated into the

[2] Defining a separate `Exception` for each Web service is the best-practice, since server exceptions should not be visible to a client.

SAT Util project's package `at.researchstudio.sat.util`. These classes provide methods for the automatic import of data via CSV [3] files and various aspects for caching, exception mapping, logging, and profiling issues.

5.4 Software Design

Figure 5.2 (page 44) gives a simplified overview of the software design of the RS decribing the main interfaces and their associations.[4] The following sections provide a detailed description of these interfaces.

5.4.1 DAO Interfaces

For the access of several database tables (introduced in Section 5.5 on page 45 et seqq.) we use the *Data Access Objects* (DAO) pattern[5] encapsulating data access methods in separate interfaces for each table. The DAOs of the *Core* module consist of the `ActionDAO`, the `ItemAssocDAO`, and the `TenantDAO` used for tasks of recommendation, the `RecommendationDAO` and `RecommendedItemDAO` for the logging of computed recommendations as well as several DAOs for tenant-specific types (like the `ActionTypeDAO` or the `ItemTypeDAO`). Additionaly the `RuleMiningDAO` supports specific methods used by the `AssocRuleMiningService`. For the mapping of external `String` IDs to internal `Integer` IDs the `IDMappingDAO` from the *SAT Util* project is used. Finally the `AuthenticationDAO` is dedicated to the management of tenant authentication. In Figure 5.2 (page 44) all boxes for DAO interfaces are kept in yellow and orange.

5.4.2 Core Services

The services of the *Core* module mainly use the corresponding DAOs and additionally *SAT Util* aspects for caching (mentioned above). The `ActionService`, the `ItemAssocService`, the `RecommenderService`, and the `TenantService` build the main services for recommendation, while the

[3]`http://en.wikipedia.org/wiki/Comma-separated_values`, as of 2008/09/29

[4]Note: Since the RS consists of too many interfaces and classes to be pictured in one single diagramm, we have omitted several abstract base classes, generic interfaces, model classes and implementation classes for each interface. We rather point out the major interfaces and some important utility classes.

[5]`http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html`, as of 2008/09/29

`RecommenderHistoryService` manages the persistence of several recommendations provided to a user storing additional information about the context of the query as well as the results. The assigned colour for the boxes of these services pictured in Figure 5.2 (page 44) is blue.

5.4.3 Domain Services

The domain services `DomainActionService`, `DomainItemAssocService`, and `DomainRecommenderService`. Several types (for actions, items, associations, etc.) are using a tenant-specific `String` representation retrieved via the `TypeMappingService`, while IDs (of items, tenants and users) are using the `Integer` type. domain-specific interfaces like the `MusicActionService`, the `MusicRecommenderService`, and the `RascalliActionService` provide convenience methods using only those type specific methods available for that domain. Domain Services are kept in green in Figure 5.2 (page 44).

5.4.4 Webapp Services

For the propagation of Web services the `ShopRecommenderWS`, the `MusicShopRecommenderWS`, the `RascalliDFKIWS`, and the `RascalloModellingWS` are available, a description of these services is presented in Section 5.8 (page 52 et seq.). All of these services use the `IDMappingService` and the `AuthenticationService`[6] to allow for `String` IDs and restrict access of these services as well as *SAT Util* aspects for logging, profiling, and exception mapping (mentioned above). In Figure 5.2 (page 44) boxes for Web services are painted in purple.

5.4.5 Generators

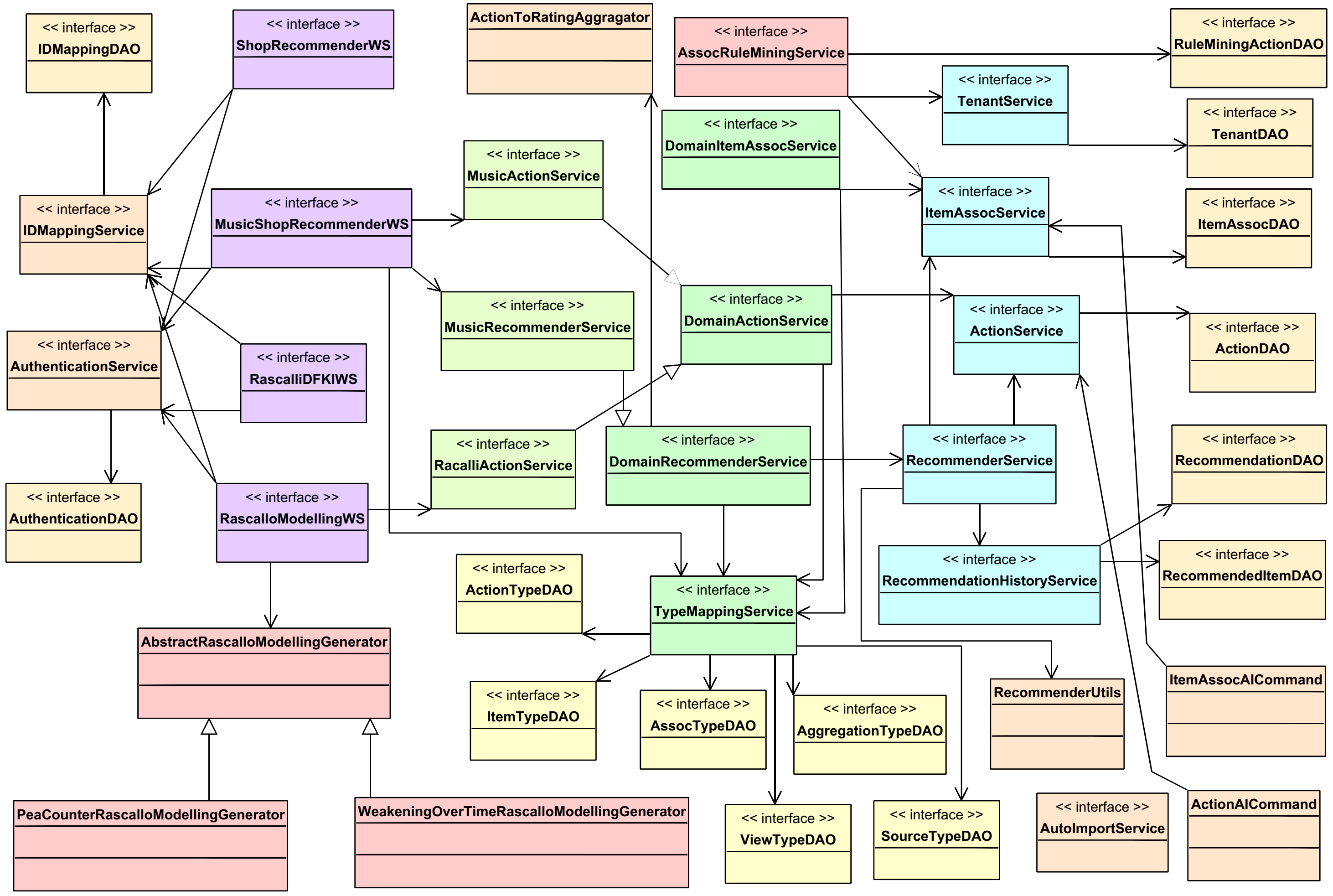
The red boxes in Figure 5.2 (page 44) are assigned to item association generators. An online generator is presented in `AbstractRascalloModellingGenerator`, while the `AssocRuleMiningService` represents an offline generator. The functionality of both generators is explained in Section 5.7 (page 50 et seqq.).

[6]The `AuthenticationService` uses API-Keying to restrict access on IP or domain basis, hence a key (assigned to a tenant) is added to the signature of all Web service methods.

5.4.6 Utility Classes

Finally all pink boxes in Figure 5.2 (page 44) picture several utility classes that are used within the RS. The `RecommenderUtils` class provides methods for the filtering of duplicates or filtering based on the action history of a user, while the `AutoImportService` is used for the import of CSV files. The `AutoImportService` uses the command[7] pattern with the need of a specific command class for each database table (for e.g. the `ActionAICommand` or the `ItemAssocAICommand`).

[7]`http://en.wikipedia.org/wiki/Command_pattern`, as of 2008/09/29



<< interface >> IDMappingDAO
<< interface >> ShopRecommenderWS
ActionToRatingAggragator
<< interface >> AssocRuleMiningService
<< interface >> RuleMiningActionDAO
<< interface >> TenantService
<< interface >> TenantDAO
<< interface >> DomainItemAssocService
<< interface >> MusicActionService
<< interface >> IDMappingService
<< interface >> MusicShopRecommenderWS
<< interface >> ItemAssocService
<< interface >> ItemAssocDAO
<< interface >> MusicRecommenderService
<< interface >> DomainActionService
<< interface >> ActionService
<< interface >> ActionDAO
<< interface >> AuthenticationService
<< interface >> RascalliDFKIWS
<< interface >> RacalliActionService
<< interface >> DomainRecommenderService
<< interface >> RecommenderService
<< interface >> RecommendationDAO
<< interface >> AuthenticationDAO
<< interface >> RascalloModellingWS
<< interface >> ActionTypeDAO
<< interface >> TypeMappingService
<< interface >> RecommendationHistoryService
<< interface >> RecommendedItemDAO
AbstractRascalloModellingGenerator
<< interface >> ItemTypeDAO
<< interface >> AssocTypeDAO
<< interface >> AggregationTypeDAO
RecommenderUtils
ItemAssocAICommand
PeaCounterRascalloModellingGenerator
WeakeningOverTimeRascalloModellingGenerator
<< interface >> ViewTypeDAO
<< interface >> SourceTypeDAO
<< interface >> AutoImportService
ActionAICommand

Figure 5.2: *Simplified Overview of the Software Design*

5.5 Data Model

The comprehensive data model pictured in Figure 5.3 (page 47) shows the database tables and their associations. A brief description for those database tables is provided in the remainder of this section.

Tenant The `Tenant` table contains tenant-specific information like an `id`, a `stringId`, and a `description`. Furthermore the range for the rating scale can be defined using `ratingRangeMin` and `ratingRangeMax`, while `ratingRangeNeutral` is used for the decision whether a rating is considered as *good* or *bad*. In Figure 5.3 (page 47) the `Tenant` table is coloured pink.

Actions and Item Associations Concerning recommendations the database tables `Action` and `ItemAssociation` hold the most important information. A description of the conceptual fields can be found in Section 4.3.2 (page 25 et seq.) and Section 4.3.4 (page 27 et seq.). Additionally the `Action` table provides the following attributes: a `sessionId`, an `IP`, and a textual `description` for all action entries as well as a `ratingValue` for rating actions and a flag `searchSucceeded` plus the `numberOfFoundItems` for search actions. The `ItemAssociation` table is extended with an `active` flag for the management of rules. The assigned colour for `Actions` and `ItemAssociations` in Figure 5.3 (page 47) is blue.

History of Recommendations In order to persist all recommendations ever computed the tables `Recommendation` and `RecommendedItem` are used. General user information and the list of results as well as optional query parameters (like the `queriedItemId` or the `relatedActionTypeId`) can be stored. Furthermore these tables contain an attribute for the `explanation` provided to the user describing why this recommendation was presented and why each of the recommended items is contained. Optionally the `itemAssocId` of the underlying busines rules can be saved. In Figure 5.3 (page 47) the tables are painted in green.

Types Caused by the generic concept of the RS several types are defined separately for each tenant. Such types are: the `ActionType`, the `AssociationType`, the `AggregateType`, the `ItemType`, the `SourceType`, and the `ViewType`. In Figure 5.3 (page 47) the tables are kept in yellow.

Authentication The `Authentication` table stores tenant-specific IPs and domain URLs. Its colour in Figure 5.3 (page 47) is orange.

Mapping of IDs In order to provide the possibility to propagate Web service methods using `String` IDs but use `Integer` IDs for the faster execution of item association generators the `IDMapping` table is used which is coloured purple in Figure 5.3 (page 47).

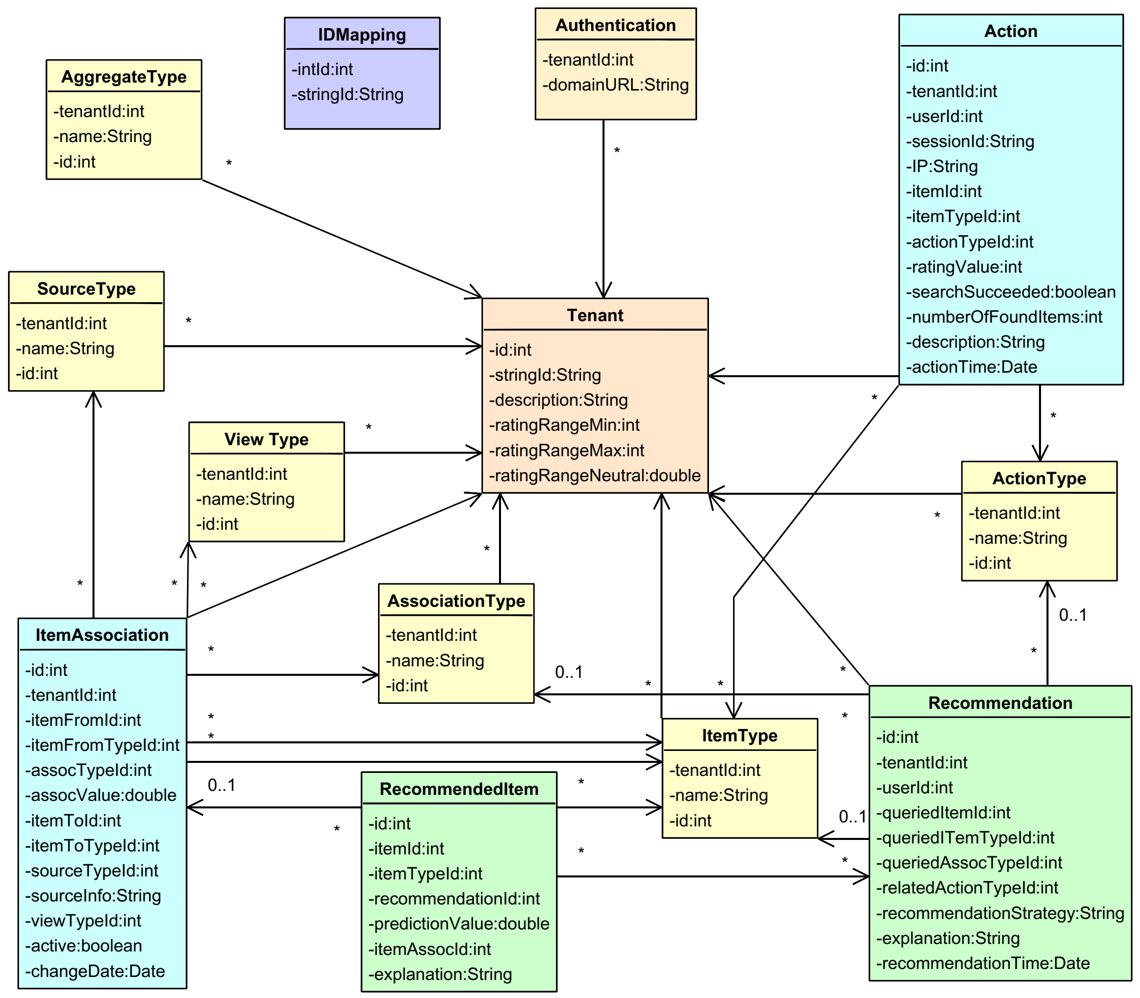


Figure 5.3: *Data Model describing database tables and their associations*

5.6 Deployment Architecture

Figure 5.4 (page 49) demonstrates the deployment architecture of the RS. The following servers are used for the running system:

Recommender Server On this server the Web application of the RS operates providing action logging as well as recommendations using various Web service interfaces. In Figure 5.4 (page 49) the *RecommenderServer* is coloured green.

Database Server The *Database Server* is located in the same internal network as the *Recommender Server* and contains several versions of recommender databases. The assigned colour in Figure 5.4 (page 49) is blue.

Metadata Server Also assigned to the same internal network each offline generator can run on a separate machine distributing computation efforts on multiple machines. The generators directly access the *Database Server* for the retrieval of user actions and the persistence of IAR. The *Metadata Server* is painted pink in Figure 5.4 (page 49).

Furthermore we plan the implementation of an administrative tool for the RS which then would be deployed to a different server.

Adminstration Server It is planned to launch a future administration tool for the RS on a separate server. This tool will be able to manage business rules as well as change settings of the RS and several running generators (separately for each tenant) and is kept in yellow in Figure 5.4 (page 49).

In addition to the running live system we use the same deployment architecture for another set of servers running in our test environment. For the actual task of deploying the Web application of the RS we use different *Maven2* profiles for each environment (live and test), while for the migration of databases we use simple SQL scripts.

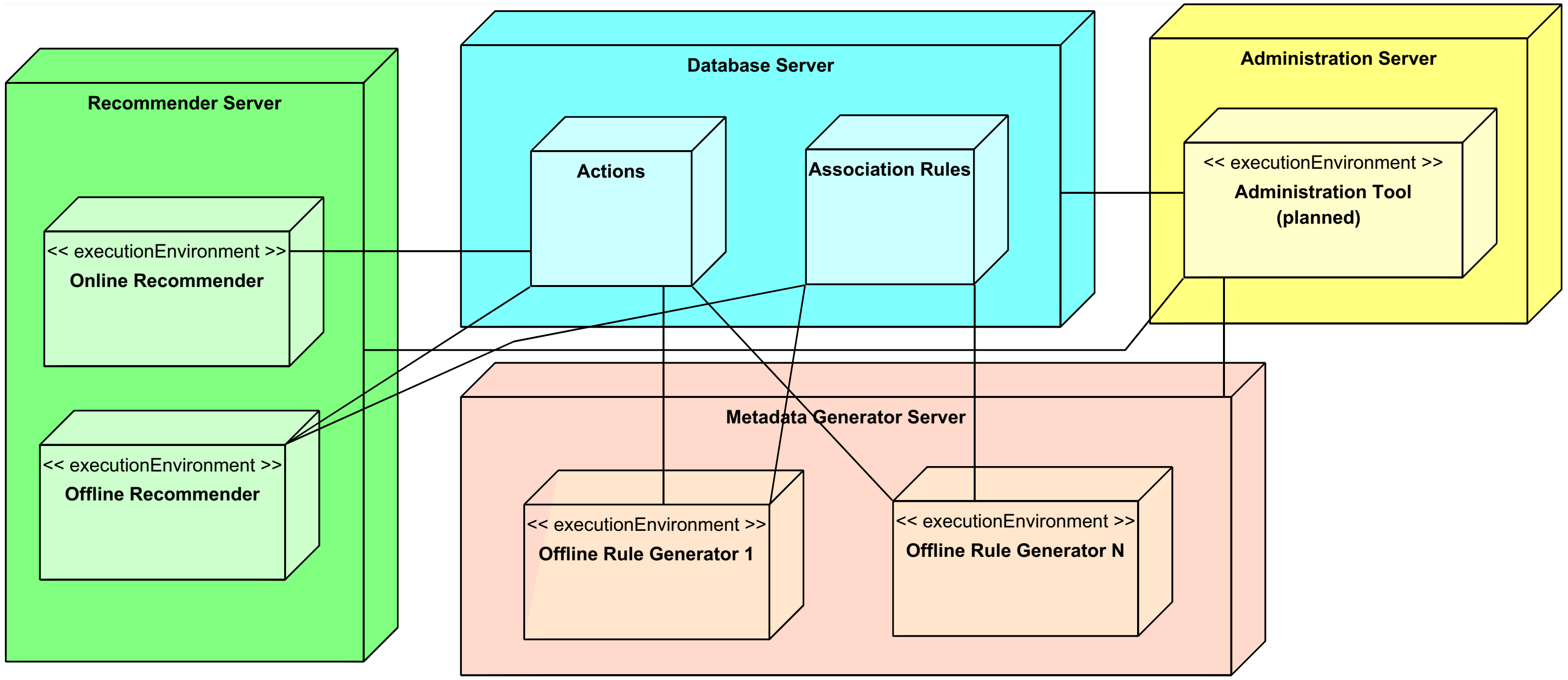


Figure 5.4: *Deployment Architecture*

5.7 Supported Recommender Features

5.7.1 An Offline Generator

For the integration of an offline generator we have adapted the SAT *Association Rule Miner* (ARM) algorithm invented, adapted, and provided by Erich Gstrein. The ARM functions similar to standard market basket analysis approaches using the set of user actions as shopping cart and generating IAR of the form e.g. ”Item I viewed together with item J”, or ”Item I bought together with item J”. These rules allow recommendations like ”Users who viewed item I also viewed item J”.

The main parameters for this algorithm are:

Support defines the number of market baskets fulfilling a potentially introduced item association rule. Defining a certain threshold for the support allows to manage the generation of rules. Thus a rule is only generated when both items of the rule (antecedent and consequent) occur together in a high enough number of baskets. Most often the confidence is defined in percent of the overall number of baskets, allowing the usage of the algorithm (with the same settings) on content pools of different sizes.

Confidence is the proportion of the number of baskets containing both items (antecedent and consequent) to the number of baskets containing only the antecedent. Hence a probability for the success of the generated rule is defined.

Lift is defined as the proportion of *Confidence* to *Expected Confidence* which describes the proportion of the number of baskets only containing the consequent to the overall number of baskets. Hence *Lift* describes the probability gain for the consequent for a given antecedent.

Further improvements and implementation details of the algorithm underly an non-disclosure agreement and will be revealed in the near future in a publication of Erich Gstrein.

5.7.2 An Online Generator

As online generator we have designed and implemented the *Rascalli Modelling Generator* (RMG) which aggregates user actions (for the RASCALLI music domain) into user preferences requesting additional content metadata (e.g. the assigned genres of an artist). These preferences are stored as IAR of the basic form ”User U likes item I” (for e.g. ”Peter likes the genre pop”).

Technically, for the RASCALLI music domain we decide between three different item types (artists, genres, and tracks) and five action types (choose topic, preview, rate, search, and view).

The RMG can be configured using the following parameters:

Action Mapping A set of weights for the set of actions defining an ordering between the given action types.

Propagation Factors Two propagation factors for the track-artist and artist-genre propagation. These parameters are used to propagate user actions over artists to the assigned artist’s genres, as well as for the propagation of track actions to the appropriate artist and genres of that track.

Lowest Action Multiply Factor This factor is used to compute the overall minimum preference value (for the given *Action Mapping*). It defines the number of actions (with the lowest weight) necessary for a preference rule.

Max Artist Genre Count Defines the allowed maximum number of genres assigned to an artist for the propagation of genre preferences. This parameter was introduced to overcome the problem that some artists (of the underlying content pool) are dedicated to too many genres, such that propagating genre information for that artist would only result in noise instead of valuable information.

We have implemented two different strategies of the RMG:

Pea Counting Strategy This implementation simply uses the parameters mentioned above aggregating (weighted) user actions to preferences for certain artists, genres, and tracks.

Weakening Over Time Strategy This strategy is an extension of the *Pea Counting Strategy* using the same aggregation process, but for each new action all previous actionweights are alleviated. We assume that this strategy fits better to the needs of a recommender since user actions of the past are weighted lower than recent user actions which allows for the change of user preferences over time.

The complete showcase for this generator is provided in Chapter 6 (page 55 et seqq.) introducing several client applications.

5.7.3 Statistics over Rankings and Ratings

Additionally to online and offline generators we provide simple statistics over the rankings of items and user ratings. The rankings of items are classified over the action type, hence statistics like ”The list of the ten most viewed items” are provided. Concerning ratings we have implemented a set of methods compiling a list of items (ordered by their rating value over all users).

5.8 Web Service Interfaces

As general interface for various clients from the SAT or partner institutes and customers a set of different Web services is propagated.

The RS provides four different Web services[8]:

- `ShopRecommenderWS`[9] A domain-independent Web service, where all item types and association types are completely parameterisable, while action types for a classical Web shop[10] are assumed. It supports item-based recommendations (like `alsoViewedItems` or `itemsBasedOnBuyingHistory`) as well as rating-based recommendations (like `alsoGoodRatedItems`) and additionally offers simple statistics over rankings (like `mostViewedItems`) and ratings (like `goodItemRatings`).

- `MusicShopRecommenderWS`[11] A domain-specific Web service representing a convenience interface of the `ShopRecommenderWS` using the same service methods but restricting item types to the music domain.[12] Nevertheless not all combinations of suggested item and action types

[8]Note: The URLs of these Web services are currently existing, but may not be available for long, since the version of the RS changes with every release and thus Web service URLs change as well.

[9]`http://intralife.researchstudio.at/sat-recommender-1.2/nodomain?wsdl`, as of 2008/09/26

[10]Such actions of a Web shop are typically ”buy”, ”view”, ”search” and ”rate”.

[11]`http://intralife.researchstudio.at/sat-recommender-1.2/music?wsdl`, as of 2008/09/26

[12]For now we support the item types ”artist”, ”track” and ”genre”, but other item types like ”album” or ”group” can be added easily.

are implemented as convenience methods, since their underlying semantic would not be appropriate for a music web shop.[13]

- `RascalliDFKIWS`[14] A domain-specific Web service for the *Deutsches Forschungszentrum für Künstliche Intelligenz* (DFKI)[15] providing *Visual Browser* (VB)[16] specific methods for the transmission of user actions.
- `RascalloModellingWS`[17] A domain-specific Web service for the *RASCALLI Environment*[18] supplying methods to retrieve artist, genre, and track preferences as well as viewed *Visual Browser* topics by count. (Both RASCALLI client applications are introduced in Section 6.3 on page 58 et seqq..) Additionally it allows the switching between two modelling strategies.

A full listing of Web service methods for each of these four Web services is given in Appendix C (page 92 et seqq.).

5.9 Preliminary Evaluation of a Rating-Based Algorithm

In the very early stage of this project, when the decision whether to implement several recommender algorithms by ourselves or to use existing implementations within open-source frameworks integrated in our RS was not made yet, we evaluated a rating-based item-to-item CF algorithm. The full description of this task can be found in Appendix A (page 75 et seqq.). The evaluation showed that the specific implementation we tested did not scale very well with a growing size of user ratings. Thus we decided not to

[13]An example: The combination of item type ”artist” with the action type ”buy” would be quite absurd.

[14]`http://intralife.researchstudio.at/sat-recommender-1.2/rascalli/dfki?wsdl`, as of 2008/09/26

[15]`http://www.dfki.de`, as of 2008/10/03

[16]Since the VB does not provide a Web site for the main entry but rather only offers specific artist pages, the following URL, pointing to the artist page of ”Madonna”, is provided as an example.

`http://rascalli.dfki.de/live/ontology.page?ctag=dfki&id=Artist.14913`, as of 2008/10/03

[17]`http://intralife.researchstudio.at/sat-recommender-1.2/rascalli/website?wsdl`, as of 2008/09/26

[18]`http://intralife.researchstudio.at/rascalli`, as of 2008/09/30

use this implementation but rather to implement several algorithms and to integrate them into our RS.[19]

Summary

We demonstrated an overview of the RS architecture and provided a description of the module and package structure in this chapter. Furthermore we explained the detailed SW design (consisting of the major interfaces and classes). Moreover the data model and the deployment architecture were presented. Finally we introduced supported recommender features and Web services for their propagation. Additionally we pointed to appendix chapters describing the general SW infrastructure, the *Java* interfaces of the Web services, and the evaluation of a rating-based algorithm.

[19]At this point of the project the goals were slightly adapted, but to demonstrate the full entireness of the work at hand we kept this chapter and moved it to the appendix.

Chapter 6
Applications

Abstract

In order to describe an application scenario for the RS this chapter introduces the RASCALLI project in Section 6.1 and describes the use case for the RS in Section 6.2. Furthermore the client applications that have integrated Web services of the RS are introduced in Section 6.3.

6.1 The RASCALLI Project

Since the RS is used in the *Responsive Artificial Situated Cognitive Agents Living and Learning on the Internet* (RASCALLI) project this section gives an overview of the project.

> ”The project RASCALLI aims at the development of responsive artificial situated cognitive agents that live and learn on the Internet (Rascalli). Rascalli represent a growing class of cooperative agents that do not have a physical presence, but nevertheless are equipped with major ingredients of cognition including situated correlates of physical embodiment to become adaptive, cooperative, and self improving in a certain environment (Internet) given certain tasks. Their task-based processing of Web content requires an action-based model of interpretative perception. Because of the size and importance of their memory, special attention is paid to the associative structuring of the acquired information based on interests and experience, and to models of an active, permanently structure-creating and restructuring memory. With Rascalli we aim at artificial agents that are able to combine human and computer skills in such a way that both kinds of abilities can be optimally employed for the benefit of the human user.
>
> We develop and implement a system that integrates a cognitive model and architecture with modules such as digital memory, knowledge representation and special purpose components as

> realized in question-answering systems and music search engines. The system allows a user to create a digital presence functioning like an avatar or agent that lives in the Internet and can be instructed by the user to collect information on certain user defined topics, to engage on behalf of the user in Web communities, to establish contact with other digital presences, avatars and users, etc. In the human-computer interface, Rascalli appear as embodied conversational characters. The feasibility of the approach is demonstrated via two application scenarios: a scenario where users train their Rascalli to successfully compete in a quiz game, and a scenario where Rascalli assist the human user in gathering and organizing information related to music."[1]

6.2 Use Case for the Recommender System

The use case described in Figure 6.1 (page 57) is used for the application scenario (mentioned above) where a user agent (rascallo) is trained to provide it's user with music related information. While the user navigates through client applications user actions are collected at the RS. These actions are then processed by item assocation generators producing association rules for that user forming an implicit user profile of liking and disliking. Furthermore these rules (e.g. "User I likes item J") are used to provide user preferences to the rascallo, hence allowing the rascallo to improve recommendations of items presented to the user.

[1]cp. `http://www.ofai.at/rascalli/project/project.html`, as of 2008/09/29

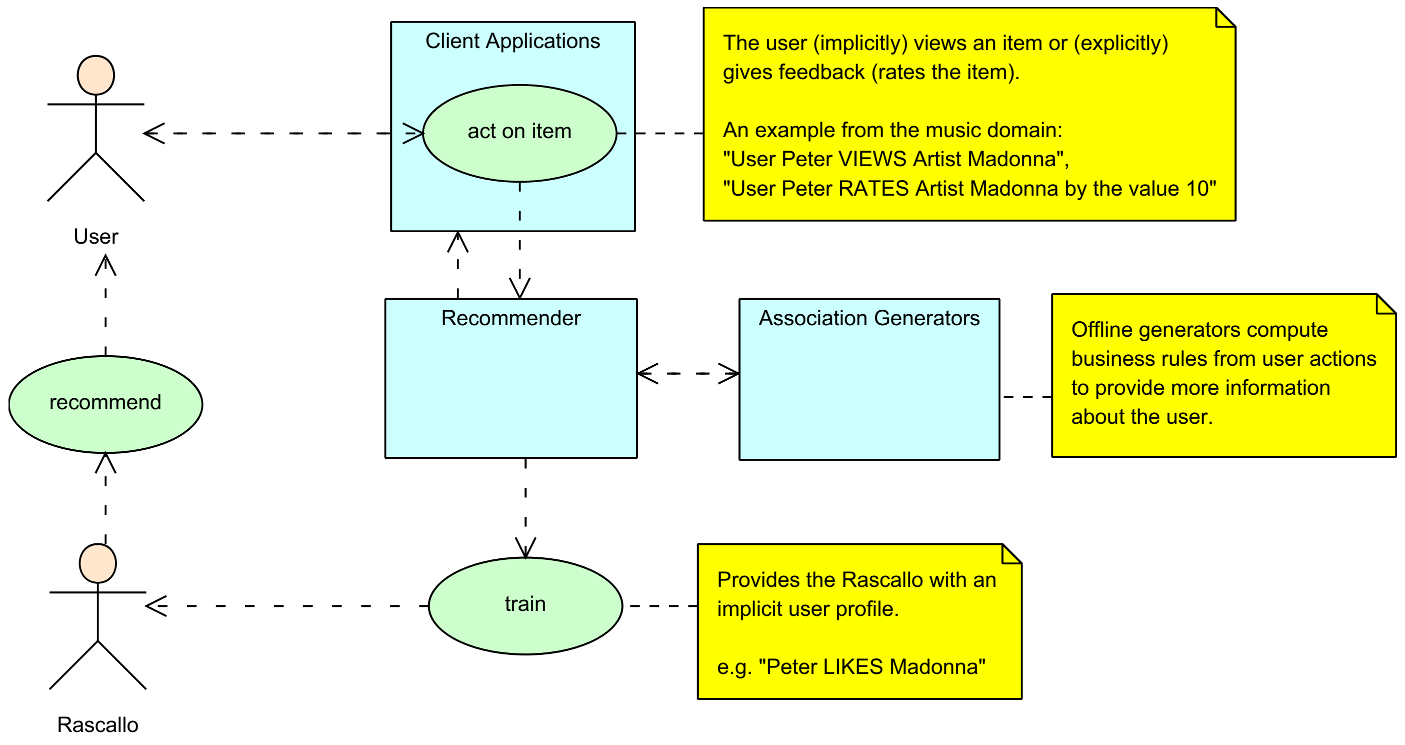

Client Applications
act on item
The user (implicitly) views an item or (explicitly) gives feedback (rates the item).
An example from the music domain:
"User Peter VIEWS Artist Madonna",
"User Peter RATES Artist Madonna by the value 10"
User
recommend
Recommender
Association Generators
Offline generators compute business rules from user actions to provide more information about the user.
train
Provides the Rascallo with an implicit user profile.
e.g. "Peter LIKES Madonna"
Rascallo

Figure 6.1: *RASCALLI Use Case: Training a Rascallo with the Recommender*

6.3 Introduction of Involved Applications

The *RASCALLI Platform* supports multi-agent architectures as well as multiple versions of the same component and combines several applications developed at different research institutes (Krenn and Schollum, 2008; Eis et al., 2008). Nevertheless this chapter describes only those applications involved in the use case described in Section 6.2 (page 56 et seq.).

6.3.1 The SAT *Music Explorer*

The SAT *Music Explorer* (MEX)[2] enables fast and target-oriented navigation in huge music archives.

Besides providing artist and genre information (as pictured in Figure 6.2 on page 59) one of the MEX core features is a playlist generator (Figure 6.3 on page 59) based on *sound similarity* computed from the sound signal.

The *Music Explorer* uses the `MusicShopRecommenderWS` for the actual task of tracking user actions and additionally has incorporated recommender methods and statistics. So each time an artist page is displayed to the user, an action ”User U has viewed artist A” is stored to the recommender database. Since the MEX is strictly dedicated to the music domain it uses four different item types (artists, genres, playlists, and tracks) and four action types (preview, rate, search, and view).

6.3.2 The DFKI *Visual Browser*

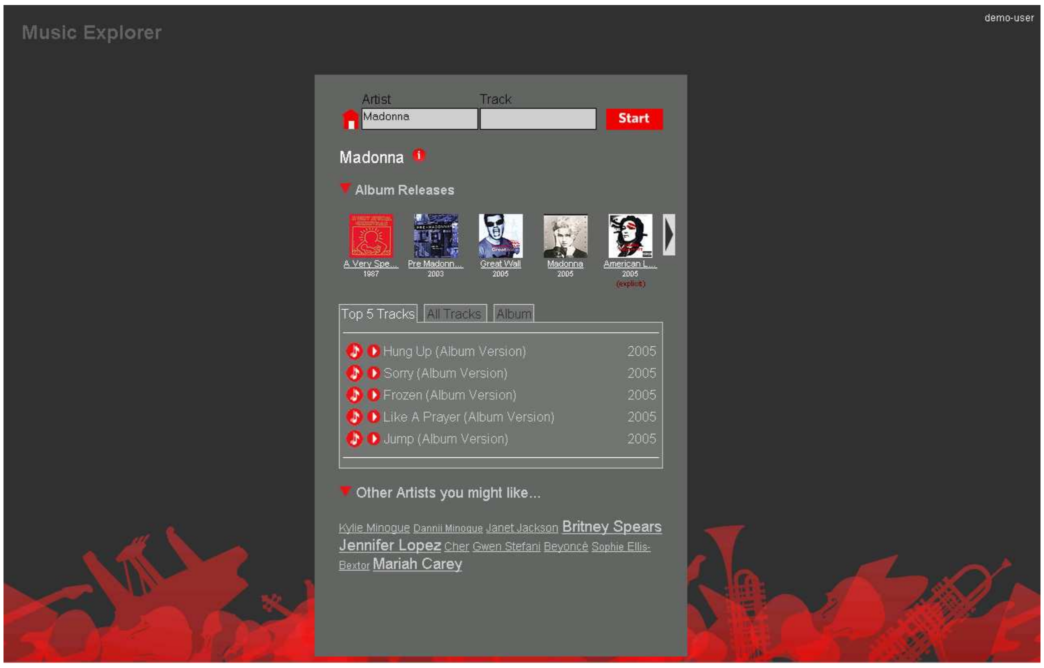
The *Visual Browser* developed at the DFKI provides extensive artist metadata to the user.

Besides background information like full name, gender, and related genres the VB demonstrates various artist connections (parents, children, and partners) as well as records and prizes (concerning the artist’s professional career).

The *Visual Browser* uses the `RascalliDFKIWS` for the logging of user actions of the type ”choose topic” (plus additional topic information) to the RS. A screenshot for the VB is presented in Figure 6.4 (page 60).

Furthermore all user actions from the MEX and the VB are processed by the online item association generator RMG (as described in Section 5.7.2 on page 50 et seqq.), hence an implicit user model consisting of preferences for artists, genres, and tracks is computed and stored as IAR.

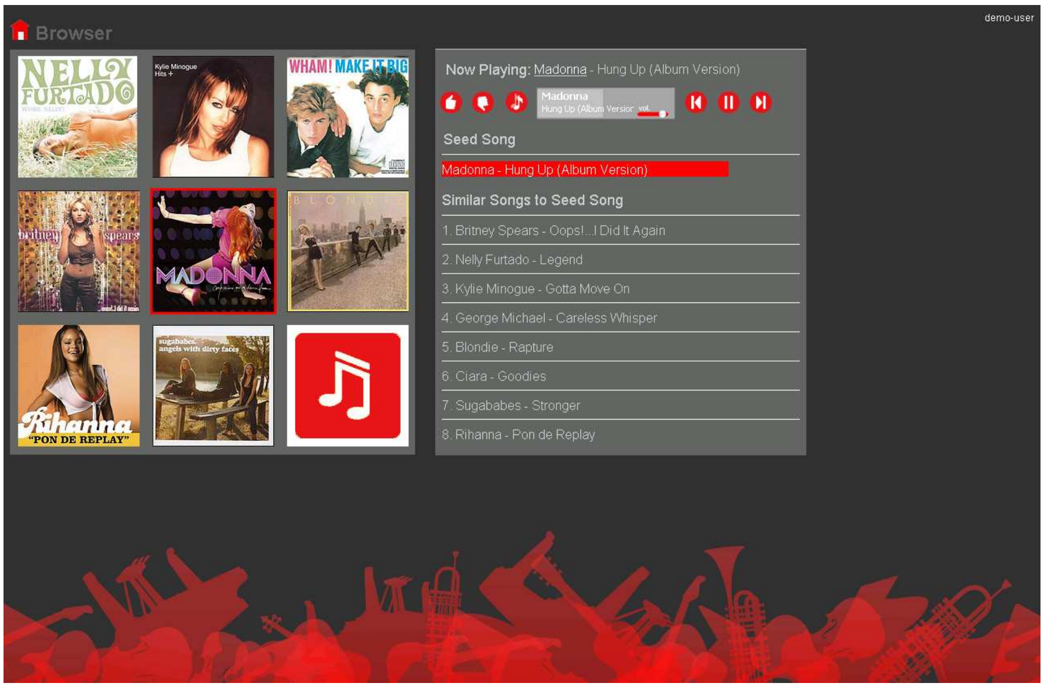
[2]`http://rascalli.researchstudio.at`, as of 2008/10/03



Hung Up (Album Version) 2005
Sorry (Album Version) 2005
Frozen (Album Version) 2005
Like A Prayer (Album Version) 2005
Jump (Album Version) 2005

Kylie Minogue Dannii Minogue Janet Jackson Britney Spears Jennifer Lopez Cher Gwen Stefani Beyoncé Sophie Ellis-Bextor Mariah Carey

Figure 6.2: Music Explorer: *Artist Page*

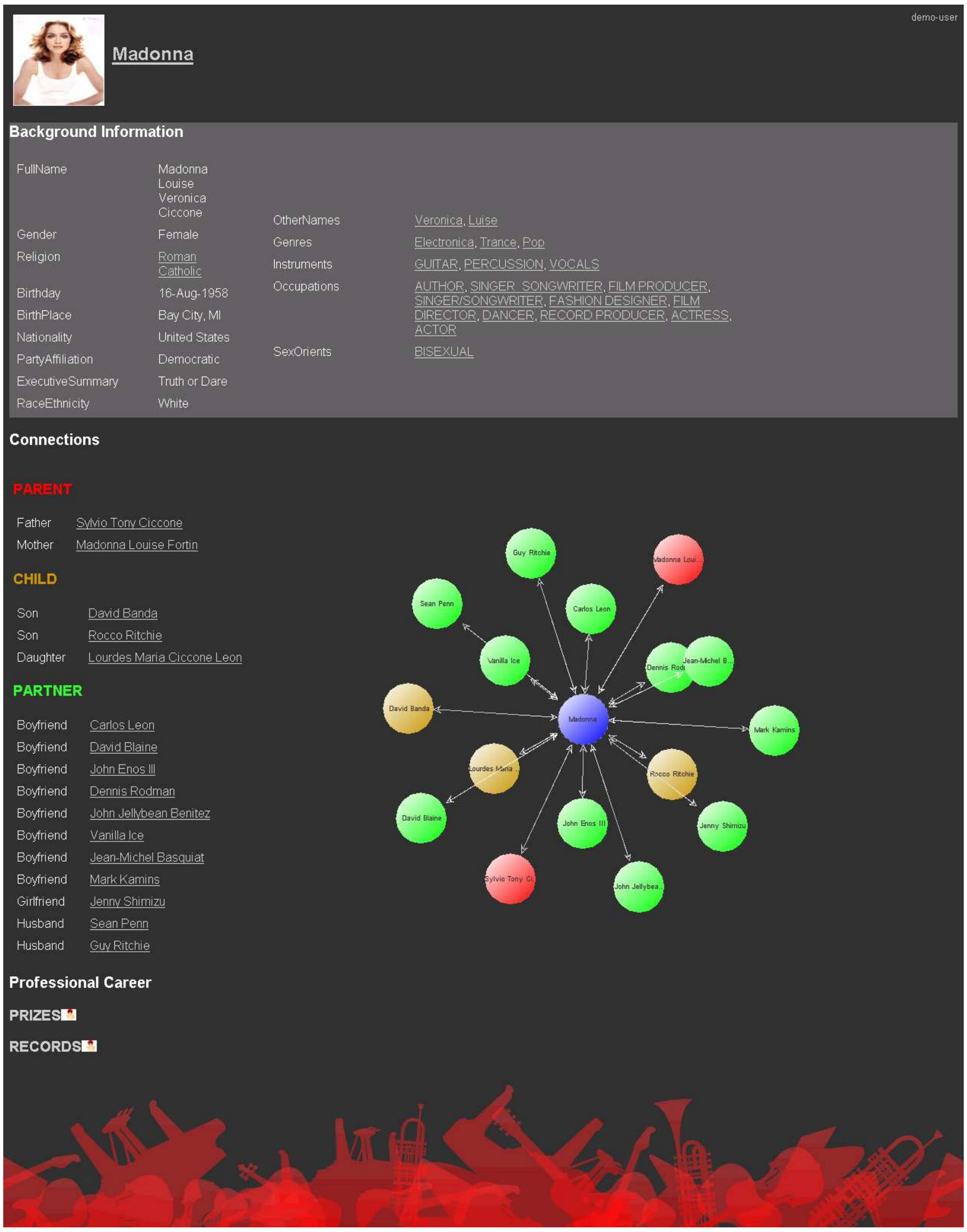


Now Playing: Madonna - Hung Up (Album Version)

Madonna - Hung Up (Album Version)

1. Britney Spears - Oops!...I Did It Again
2. Nelly Furtado - Legend
3. Kylie Minogue - Gotta Move On
4. George Michael - Careless Whisper
5. Blondie - Rapture
6. Ciara - Goodies
7. Sugababes - Stronger
8. Rihanna - Pon de Replay

Figure 6.3: Music Explorer: *Explorer Page*



demo-user

Madonna

Background Information

FullName	Madonna Louise Veronica Ciccone
Gender	Female
Religion	Roman Catholic
Birthday	16-Aug-1958
BirthPlace	Bay City, MI
Nationality	United States
PartyAffiliation	Democratic
ExecutiveSummary	Truth or Dare
RaceEthnicity	White

OtherNames	Veronica, Luise
Genres	Electronica, Trance, Pop
Instruments	GUITAR, PERCUSSION, VOCALS
Occupations	AUTHOR, SINGER_SONGWRITER, FILM PRODUCER, SINGER/SONGWRITER, FASHION DESIGNER, FILM DIRECTOR, DANCER, RECORD PRODUCER, ACTRESS, ACTOR
SexOrients	BISEXUAL

Connections

PARENT

Father	Sylvio Tony Ciccone
Mother	Madonna Louise Fortin

CHILD

Son	David Banda
Son	Rocco Ritchie
Daughter	Lourdes Maria Ciccone Leon

PARTNER

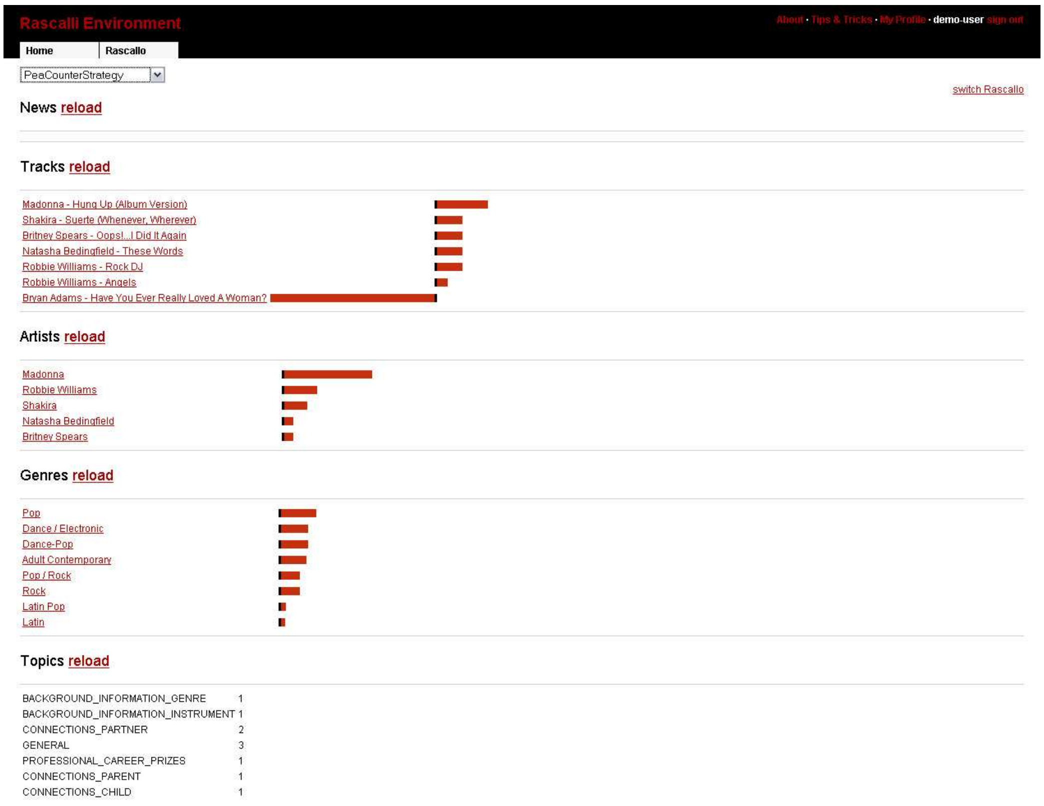
Boyfriend	Carlos Leon
Boyfriend	David Blaine
Boyfriend	John Enos III
Boyfriend	Dennis Rodman
Boyfriend	John Jellybean Benitez
Boyfriend	Vanilla Ice
Boyfriend	Jean-Michel Basquiat
Boyfriend	Mark Kamins
Girlfriend	Jenny Shimizu
Husband	Sean Penn
Husband	Guy Ritchie

Professional Career

PRIZES

RECORDS

Figure 6.4: Visual Browser: *Artist Page*



News reload

Tracks reload

Madonna - Hung Up (Album Version)
Shakira - Suerte (Whenever, Wherever)
Britney Spears - Oops!...I Did It Again
Natasha Bedingfield - These Words
Robbie Williams - Rock DJ
Robbie Williams - Angels
Bryan Adams - Have You Ever Really Loved A Woman?

Artists reload

Madonna
Robbie Williams
Shakira
Natasha Bedingfield
Britney Spears

Genres reload

Pop
Dance / Electronic
Dance-Pop
Adult Contemporary
Pop / Rock
Rock
Latin Pop
Latin

Topics reload

BACKGROUND_INFORMATION_GENRE	1
BACKGROUND_INFORMATION_INSTRUMENT	1
CONNECTIONS_PARTNER	2
GENERAL	3
PROFESSIONAL_CAREER_PRIZES	1
CONNECTIONS_PARENT	1
CONNECTIONS_CHILD	1

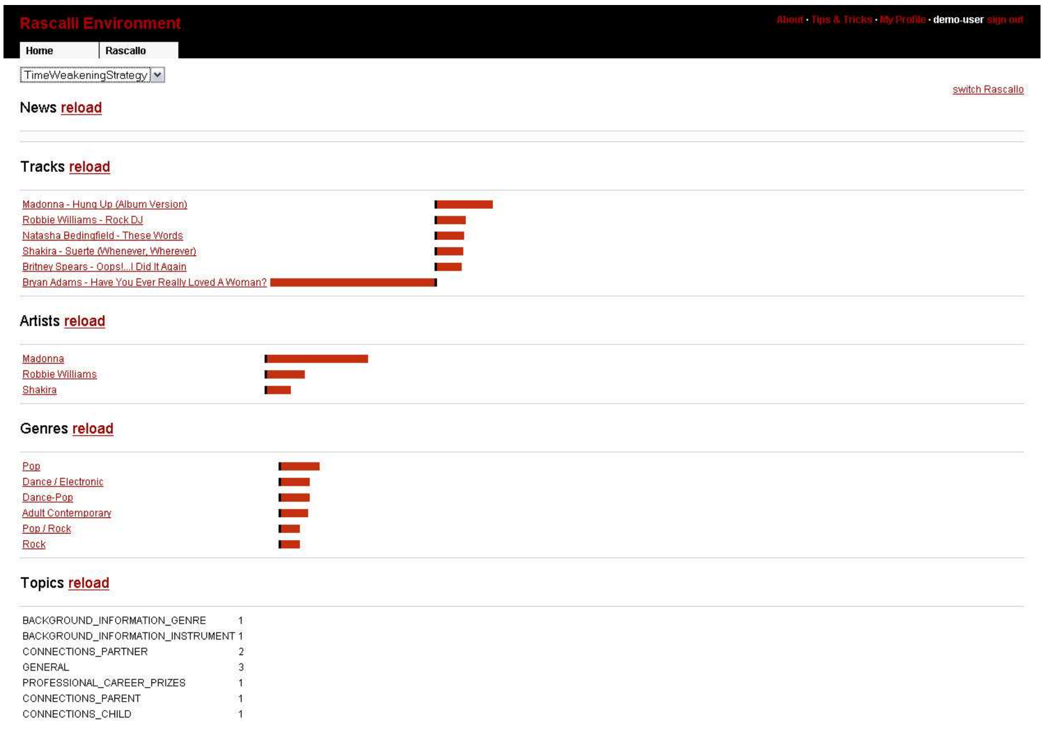
Figure 6.5: RASCALLI Environment*:* Pea Counting *Strategy*

6.3.3 The SAT *RASCALLI Environment*

The main entry point for the *RASCALLI Platform* is provided by the SAT *RASCALLI Environment*[3] allowing for the registration of a user and the configuration of the user's rascallos.

Additionally users are guided to the *Music Explorer* and the *Visual Browser*. Furthermore the *RASCALLI Environment* displays the user's profile using both strategies of the RMG (mentioned in Section 5.7.2 on page 50 et seqq.). The user profile for the *Pea Counting* strategy is depicted in Figure 6.5, while the profile using the *Weakening Over Time* strategy is presented in Figure 6.6 (page 62).

[3]`http://intralife.researchstudio.at/rascalli`, as of 2008/10/03



Rascalli Environment

About · Tips & Tricks · My Profile · demo-user sign out

Home | Rascallo

TimeWeakeningStrategy

switch Rascallo

News reload

Tracks reload

Madonna - Hung Up (Album Version)
Robbie Williams - Rock DJ
Natasha Bedingfield - These Words
Shakira - Suerte (Whenever, Wherever)
Britney Spears - Oops!...I Did It Again
Bryan Adams - Have You Ever Really Loved A Woman?

Artists reload

Madonna
Robbie Williams
Shakira

Genres reload

Pop
Dance / Electronic
Dance-Pop
Adult Contemporary
Pop / Rock
Rock

Topics reload

BACKGROUND_INFORMATION_GENRE	1
BACKGROUND_INFORMATION_INSTRUMENT	1
CONNECTIONS_PARTNER	2
GENERAL	3
PROFESSIONAL_CAREER_PRIZES	1
CONNECTIONS_PARENT	1
CONNECTIONS_CHILD	1

Figure 6.6: RASCALLI Environment*:* TimeWeakening *Strategy*

One of the RASCALLI project partners, the *New Bulgarian University* (NBU)[4], is concerned with the evaluation of the RASCALLI system and it's use cases. Results will be released in a publication of the NBU at the end of the RASCALLI project.

Summary

In this chapter we demonstrated an application scenario for the RS and briefly described the RASCALLI project. Additionally we introduced the client applications involved: the SAT *Music Explorer*, the SAT *RASCALLI Environment* and the DFKI *Visual Browser*. Finally we mentioned the external evaluation of the scenario through the RASCALLI partner institute NBU.

[4]`http://www.nbu.bg/cogs/center/index.html`, as of 2008/09/30

Chapter 7
Conclusion

In the work at hand we presented a generic recommender that can be applied to various domains and different recommender techniques.

After an overview of existing recommender approaches and evaluation measures, we pointed out some common problems. Furthermore we explained some interesting recommender approaches including some of their features. We then identified the challenges for our RS which are, among others: domain-independence, extensibility, multi-tenant capability, and scalability. We designed and presented a concept that mainly relies on the gathering of user actions, and the processing of these actions either in *offline* operating item association generators, or *online* as direct recommendation services. In order to serve as managed service for multiple tenants and additionally reduce complexity of computations, we introduced types for items, actions, and associations that allow for the generation of item-to-item business rules limited to a bounded set of actions. We proposed that the usage of such rules additionally: supports different recommender approaches since third-party metadata can be imported as rules (e.g. a content-based approach that defines item-similarity), enables complex algorithms to work offline (e.g. collaborative filtering methods on large data sets), and allows for the individual management of the recommendation process as well as providing explanations to the user, why an item was recommended. On a technical level we furthermore described the SW architecture, the SW infrastructure, the existing modules and packages, the SW design, the data model, the deployment architecture, and the used recommender methods. We mentioned various Web service interfaces for domain-independent recommendations, as well as convenience services for the music domain and the RASCALLI domain, where we applied the system. These Web services proofed to be easily integrated into multiple client applications within the RASCALLI project, as we finally demonstrated a use case in a real-life scenario as well as the functional efficiency of the concept.

Concluding we believe that the proposed approach adresses the major problems of recommender systems: the cold-start problem, sparsity of data, and scalability. Nevertheless evidence can be found once content-based item association generators are introduced and the number of collected user actions exceeds the usual capacity.

Chapter 8
Future Work

We plan the implementation of the rating-based item-to-item *Slope One* recommender algorithm as well as the incorporation of content-based generators. Additionally the appropriation of baseline evaluation measures as the *Pearson Correlation* (Section 2.4.3 on page 12) is under discussion.

In addition we target to implement an administrative tool for the RS to manage stored user *Actions* and *Item Association Rules* for each tenant separately. Thus a system operator could manually override or disable certain rules and add new ones as well.

Furthermore we aim to enhance the RS to incorporate methods of the field of *User Modelling* allowing for the aggregation of user actions into (implicit and explicit) user profiles as well as incremental refinement of such profiles. Current forms of representing a user model like *RDF/OWL* or *XML/XML-Schema*, and new standards like *OpenTaste*[1] shall be investigated.

Another interesting field of research are *Self Adapting Systems* that incorporate methods for the self-management of the systems configuration. For the RS such an approach offers the possibility to refine the configuration of generators and the system itself automatically.[2]

Moreover we consider to broaden our research in the field of *Custom User Tags* providing methods to compute a similarity distance between items based on the categories assigned by the users.

Also a *Representational State Transfer (REST)*[3] architecture will be introduced for all Web services soon, allowing well-defined operations over HTTP and representing queried data (respectively resources) in a uniform format (typically HTML, XML or JSON).

Finally we plan to add a simplified Web service interface (we call it *EasyRec*) that uses just a single item type. Additionally a Web application for the registration of new tenants, the transmission of actions, and the retrieval of recommendations via the *EasyRec* interface will be launched and used to demonstrate the basic functionality of the RS.

[1]`http://www.opentaste.net`, as of 2008/30/08

[2]An example: An item association generator would then decide whether to operate *online* or *offline* depending on the tenants content and the time needed for the generation of business rules.

[3]`http://en.wikipedia.org/wiki/REST`, as of 2008/09/19

List of Acronyms

API	Application Programming Interface
ARM	Association Rule Miner
CF	Collaborative Filtering
DAO	Data Access Object
DFKI	Deutsches Forschungszentrum für Künstliche Intelligenz
IAR	Item Assocation Rules
ID	Identifier
IDE	Integrated Development Environment
MEX	Music Explorer
MIR	Music Information Retrieval
NBU	New Bulgarian University
RASCALLI	Responsive Artificial Situated Cognitive Agents Living and Learning on the Internet
RMG	Rascalli Modelling Generator
ROI	return-of-investment
RS	Recommender System
RSA	Research Studios Austria Forschungsgesellschaft mbH
SAT	Smart Agent Technologies Studio
SO	Slope One
SOA	Service Oriented Architecture
SW	Software
TFxIDF	term frequency/inverse document frequency
VB	Visual Browser
WSDL	Web Service Definition Language

List of Figures

List of Listings

List of Tables

Bibliography

Adomavicius, G. and Tuzhilin, A. (2005). Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749. Member-Gediminas Adomavicius and Member-Alexander Tuzhilin.

Anderson, M., Ball, M., Boley, H., Greene, S., Howse, N., Lemire, D., and McGrath, S. (2003). Racofi: A rule-applying collaborative filtering system. In *COLA ’03: Proceedings of the Workshop on Collaboration Agents: Autonomous Agents for Collaborative Environments*. IEEE/WIC.

Balabanović, M. and Shoham, Y. (1997). Fab: Content-based, collaborative recommendation. *Communications of the ACM*, 40(3):66–72.

Belkin, N. J. and Croft, W. B. (1992). Information filtering and information retrieval: Two sides of the same coin? *Communications of the ACM*, 35(12):29–38.

Breese, J. S., Heckerman, D., and Kadie, C. (1998). Empirical analysis of predictive algorithms for collaborative filtering. In *UAI ’98: Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 43–52. Morgan Kaufmann.

Brozovsky, L. and Petricek, V. (2007). Recommender system for online dating service. In *ZNALOSTI ’07: Proceedings of the 6th annual conference Znalosti*, pages 17–18, Ostrava, Czech Republic. VSB.

Burke, R. (2002). Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370.

Burke, R., Mobasher, B., Williams, C., and Bhaumik, R. (2006). Classification features for attack detection in collaborative recommender systems. In *KDD ’06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 542–547, New York, NY, USA. ACM.

Cano, P., Koppenberger, M., and Wack, N. (2005). Content-based music audio recommendation. In *MULTIMEDIA ’05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 211–212, New York, NY, USA. ACM.

Eis, C., Skowron, M., and Krenn, B. (2008). Virtual agent modeling in the rascalli platform. In *PerMIS'08: Proceedings of the Performance Metrics for Intelligent Systems Workshop*, NIST, Gaithersburg, MD, USA. ACM.

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.

Goldberg, K., Roeder, T., Gupta, D., and Perkins, C. (2001). Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, 4(2):133–151.

Gstrein, E., Kleedorfer, F., and Krenn, B. (2006). Automated meta data generation for personalized music portals. Technical Report TR-2006-01, Austrian Research Centers GmbH, Studio Smart Agent Technologies, Hasnerstrasse 123, A-1160 Vienna, Austria.

Gstrein, E., Kleedorfer, F., Mayer, R., Schmotzer, C., Widmer, G., Holle, O., and Miksch, S. (2005). Adaptive personalization: A multi-dimensional approach to boosting a large scale mobile music portal. In *Fifth Open Workshop on MUSICNETWORK: Integration of Music in Multimedia Applications*, pages 1–8, Vienna, Austria.

Gstrein, E. and Krenn, B. (2006). Mobile music personalization at work. In *Proceedings of the ECAI 2006 Workshop on Recommender Systems*, pages 122–124, Riva del Garda, Italy.

Hedfi, S. and Trabelsi, A. (2005). Design of an hybrid recommender system personalization, evaluation and prediction. In *IEBC '05: Proceedings of the First International E-Business Conference*. BESTMOD, ISG de Tunis.

Herlocker, J. L., Konstan, J. A., and Riedl, J. (2000). Explaining collaborative filtering recommendations. In *CSCW '00: Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 241–250, New York, NY, USA. ACM.

Herlocker, J. L., Konstan, J. A., Terveen, L. G., and Riedl, J. T. (2004). Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems*, 22(1):5–53.

Hlavac, P., Krenn, B., and Gstrein, E. (2007). Soundscout: A song recommender based on sound similarity for huge commercial music archives.

Technical Report TR-2007-01, Austrian Research Centers GmbH, Studio Smart Agent Technologies, Hasnerstrasse 123, A-1160 Vienna, Austria.

Hoashi, K., Matsumoto, K., and Inoue, N. (2003). Personalization of user profiles for content-based music retrieval based on relevance feedback. In *MULTIMEDIA '03: Proceedings of the eleventh ACM international conference on Multimedia*, pages 110–119, New York, NY, USA. ACM.

Hurley, N. J., O'Mahony, M. P., and Silvestre, G. C. M. (2007). Attacking recommender systems: A cost-benefit analysis. *IEEE Intelligent Systems*, 22(3):64–68.

Kleedorfer, F. (2008). Automatic topic detection in song lyrics. Master's thesis, Vienna University of Technology.

Kleedorfer, F., Knees, P., and Pohle, T. (2008). Oh oh oh whoah! towards automatic topic detection in song lyrics. In *ISMIR '08: Proceedings of the 9th International Conference on Music Information Retrieval*, pages 287–292, Philadelphia, PA, USA.

Knees, P., Pampalk, E., and Widmer, G. (2004). Artist classification with web-based data. In *ISMIR '04: Proceedings of the 5th International Conference on Music Information Retrieval*, pages 517–524, Barcelona, Spain.

Krenn, B. and Schollum, C. (2008). The rascalli platform - for a flexible and distributed development of virtual systems augmented with cognition. In *CogSys '08: Proceedings of the International Conference on Cognitive Systems*, University of Karlsruhe, Karlsruhe, Germany. IEEE.

Lemire, D., Boley, H., McGrath, S., and Ball, M. (2005). Collaborative filtering and inference rules for context-aware learning object recommendation. *International Journal of Interactive Technology and Smart Education*, 2(3).

Lemire, D. and Maclachlan, A. (2005). Slope one predictors for online rating-based collaborative filtering. In *SDM'05: Proceedings of SIAM Data Mining*, pages 471–475, Newport Beach, California, USA.

Lemire, D. and McGrath, S. (2005). Implementing a rating-based item-to-item recommender system in php/sql. Technical Report D-01, Ondelette.com.

Li, C., Qi, J., Wei, W., and Shu, H. (2007). A service-oriented architecture for semantic recommendation and integration of products/application services (soa-ripas) in globalization. In *APSCC '07: Proceedings of the 2nd IEEE Asia-Pacific Service Computing Conference*, pages 382–389, Washington, DC, USA. IEEE Computer Society.

Li, Q., Kim, B. M., Guan, D. H., and whan Oh, D. (2004). A music recommender based on audio features. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 532–533, New York, NY, USA. ACM.

Linden, G., Smith, B., and York, J. (2003). Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80.

Massa, P. and Avesani, P. (2004). Trust-aware collaborative filtering for recommender systems. In *Proceedings of Federated International Conference On The Move to Meaningful Internet: CoopIS, DOA, ODBASE*, volume 3290, pages 492–508.

Massa, P. and Avesani, P. (2006). Trust-aware bootstrapping of recommender systems. In *Proceedings of the ECAI 2006 Workshop on Recommender Systems*, pages 29–33, Riva del Garda, Italy.

Massa, P. and Avesani, P. (2007). Trust-aware recommender systems. In *RecSys '07: Proceedings of the 2007 ACM conference on Recommender systems*, pages 17–24, New York, NY, USA. ACM.

Mobasher, B., Burke, R., Bhaumik, R., and Sandvig, J. J. (2007). Attacks and remedies in collaborative recommendation. *IEEE Intelligent Systems*, 22(3):56–63.

O'Donovan, J. and Smyth, B. (2005). Trust in recommender systems. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 167–174, New York, NY, USA. ACM.

Pampalk, E., Rauber, A., and Merkl, D. (2002). Content-based organization and visualization of music archives. In *MULTIMEDIA '02: Proceedings of the tenth ACM international conference on Multimedia*, pages 570–579, New York, NY, USA. ACM.

Park, S.-T., Pennock, D., Madani, O., Good, N., and DeCoste, D. (2006). Naïve filterbots for robust cold-start recommendations. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 699–705, New York, NY, USA. ACM.

Rack, C., Arbanowski, S., and Steglich, S. (2007). A generic multipurpose recommender system for contextual recommendations. In *ISADS '07: Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems*, pages 445–450, Washington, DC, USA. IEEE Computer Society.

Rauber, A., Pampalk, E., and Merkl, D. (2002). Content-based music indexing and organization. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 409–410, New York, NY, USA. ACM.

Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., and Riedl, J. (1994). Grouplens: An open architecture for collaborative filtering of netnews. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186, New York, NY, USA. ACM.

Resnick, P. and Varian, H. R. (1997). Recommender systems. *Communications of the ACM*, 40(3):56–58.

Sarwar, B., Karypis, G., Konstan, J., and Reidl, J. (2001). Item-based collaborative filtering recommendation algorithms. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 285–295, New York, NY, USA. ACM.

Sarwar, B., Karypis, G., Konstan, J., and Riedl, J. (2000). Analysis of recommendation algorithms for e-commerce. In *EC '00: Proceedings of the 2nd ACM conference on Electronic commerce*, pages 158–167, New York, NY, USA. ACM.

Schafer, J. B., Konstan, J., and Riedi, J. (1999). Recommender systems in e-commerce. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 158–166, New York, NY, USA. ACM.

Simon Hix, Abdul Noury, G. R. (2006). Dimensions of politics in the european parliament. *American Journal of Political Science*, 50(2):494–520.

Upendra, S. (1994). Social information filtering for music recommendation. Technical Report TR-94-04, MIT EECS, Learning and Common Sense Group, MIT Media Laboratory. also Masters Thesis.

Upendra, S. and Maes, P. (1995). Social information filtering: Algorithms for automating “word of mouth”. In *CHI ’95: Proceedings of the SIGCHI Conference on Human factors in Computing Systems*, pages 210–217, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

Yu, K., Xu, X., Ester, M., and Kriegel, H.-P. (2001). Selecting relevant instances for efficient and accurate collaborative filtering. In *CIKM ’01: Proceedings of the tenth international conference on Information and knowledge management*, pages 239–246, New York, NY, USA. ACM.

Ziegler, C.-N., McNee, S. M., Konstan, J. A., and Lausen, G. (2005). Improving recommendation lists through topic diversification. In *WWW ’05: Proceedings of the 14th international conference on World Wide Web*, pages 22–32, New York, NY, USA. ACM.

Appendix A
Evaluation of a Slope One Recommender Implementation on a Large Music Database

Abstract

This chapter gives a detailed report of some performance tests with a large music recommender database and the *Slope One* (SO) algorithm implemented in the common open-source CF framework *Taste*[1]. It also shows that the algorithms within the *Taste* framework can be integrated into our RS (with a small amount of effort), but also that the tested *Slope One* implementations do not scale very well for large datasets.

A.1 Goal

Since our RS is designed to manage business rules generated from different sources using various algorithms, but only item-based recommendations are implemented currently, we aim to broaden the range of supported recommender algorithms and invest some effort in the evaluation of a rating-based CF algorithm.

The goal is to measure the performance of the SO algorithm on a large music database (that was provided for research in the field of *Collaborative Filtering* by the *Verisign Communications GmbH*[2]) and the applicability of the *Taste* framework to our RS.

A.2 The Slope One Recommender Algorithm

The *Slope One* recommender algorithm (which was introduced by Lemire and Maclachlan (2005) and implemented by Lemire and McGrath (2005)) is a rating-based collaborative filtering algorithm predicting a user-rating

[1] `http://taste.sourceforge.net/`, as of 2008/06/20
[2] `http://www.3united.com`, as of 2008/06/20

for a given item by computing the average difference between ratings of other users who have already rated for that specific item (and for some of the other items the given user has already rated before).[3]

A.2.1 Basic Functionality

Slope One is a memory-based item-to-item CF algorithm trying to predict how a user would rate a given item from other user ratings.

SO works on the intuitive principle of a *popularity differential* between items for users. In a pair-wise fashion it is determined how much better one item is liked than another. One way to measure this differential is simply to subtract the average rating of the two items. In turn, this difference can be used to predict another user's rating of one of those items, given the rating of the other.

The authors have introduced three different prediction schemes: the SO scheme, the weighted SO scheme and the bi-polar SO scheme. In contrary to the simple SO scheme, the weighted scheme takes the number of observed ratings into consideration, while the bi-polar scheme separates the user ratings into two classes of good and bad ratings, hence reducing the number of predicted ratings.

Example for the *Slope One* Algorithm

Consider two users A and B and two items I and J . User A gave item I a rating of 1, whereas user B gave it a rating of 2, while user A gave item J a rating of 1.5. We observe that item J is rated better than item I by 1.5-1 = 0.5 points, thus we could predict that user B will give item J a rating of 2+0.5 = 2.5.

A.2.2 Open-Source Frameworks Providing a Slope One Implementation

To avoid re-implementation of commonly known algorithms we broadened our research field and examined some open-source tools as well. The focus of the search was to find open-source frameworks for collaborative filtering that provide an implementation of the SO algorithm. (Frameworks

[3]also see: `http://en.wikipedia.org/wiki/Slope_One`, as of 2007/03/18

for other recommendation methods like clustering or search-based approaches were omitted during this task.) The list of surveyed open-source CF frameworks can be found in Section 3.3 (page 20 et seqq.).

After taking a short look at all three *Java* frameworks it turned out that the *Taste* framework is the most active and most current framework, since it is updated about once a month, while the other *Java* projects (*Cofi* and *CoFE*) turned out to be outdated student projects that have not been updated for a while (and therefore do not support that many of the currently well known algorithms). The *Vogoo* framework was left out for now, since our RS is fully implemented in *Java*, and therefore a PHP framework would always lead to a performance loss when integrated into a *Java* framework. (But maybe the implementation of some of the algorithms could vary in (timing and quality) performance, hence a future look into these implementations could result in a possible improvement of the quality of our RS.)

A.3 The Open-Source Collaborative Filtering Framework *Taste*

> ”*Taste* is a flexible, fast collaborative filtering engine for *Java*. The engine takes users’ preferences for items (”tastes”) and returns estimated preferences for other items. (...)
>
> *Taste* provides a rich set of components from which you can construct a customized recommender system from a selection of algorithms. Taste is designed to be enterprise-ready; it’s designed for performance, scalability and flexibility. It supports a standard EJB interface for J2EE-based applications, but Taste is not just for *Java*; it can be run as an external server which exposes recommendation logic to your application via Web services and HTTP. (...)
>
> Taste supports both *memory-based* and *item-based* recommender systems, *slope one* recommenders, and a couple other [sic!] experimental implementations. It does not currently support *model-based* recommenders.”[4]

[4]cp. *Taste* documentation for version 1.5.2 (from 2007/04/12), index.html

A.4 Evaluation

A.4.1 Integration

Before we could start our evaluation process, we had to integrate the *Taste* framework into our technical infrastructure adding support for *Maven2* builds. We also contributed the *Maven2* build file to the *Taste* framework to help enabling *Maven2* builds additional to existing *Ant* builds.[5]

After the integration we tested the *Slope One* recommender on a *Taste* test database to obtain a feeling for the framework and create a vertical slice (providing a simple showcase).

The basic structure of a database table used for the *Slope One* algorithm is described best with the MySQL database DDL script in Listing A.1.

```
CREATE TABLE taste_preferences (
  userId VARCHAR(10) NOT NULL,
  itemId VARCHAR(10) NOT NULL,
  preference FLOAT NOT NULL
)
```

Listing A.1: *MySQL CREATE TABLE Statement for a* `taste_preferences` *Table*

A.4.2 Database Connectivity

The first task was to implement a database connection between our SAT recommender database and the *Taste* framework. Basically there are two main `DataModel` implementations provided by the framework. One for a file access (`FileDataModel`) and various JDBC[6] implementations for a DB access (e.g. `MySQLJDBCDataModel`). Since the `MySQLJDBCDataModel` provides an additional constructor that allows the parameterised definition of the database table containing the user ratings and the necessary columns for user, item, and rating, implementing another `DataModel` was omitted.

After the first tests it turned out that the database table containing the ratings needed additional database indices listed in Listing A.2 (for a faster read access).

```
CREATE INDEX user_item_ids
  ON taste_preferences (userId, itemId);

```

[5]At this point we'd like to thank *Sean Owen* (the author of the *Taste* framework) for his support during the integration of *Maven2* builds and for publishing our work within the *Taste* framework.

[6]The Java *Database Connectivity* (JDBC) is a *Java* API for the access of relational databases.

```sql
CREATE INDEX user_id
  ON taste_preferences (userId);

CREATE INDEX item_id
  ON taste_preferences (itemId);
```

Listing A.2: *MySQL CREATE INDEX Statements for a `taste_preferences` Table*

A.4.3 Evaluation Datasets

After the database connection (to the recommender database) was established, an initial test was run. But since the used snapshot contains more than 500,000 ratings this first test took too long to be feasible for repeated tests and we decided to test the algorithms on several different sample sizes of the database.

We randomly re-sampled the ratings table, to obtain different number of ratings:

- about 10,000 ratings (further called *10k* sample)
- about 50,000 ratings (*50k* sample)
- about 100,000 ratings (*100k* sample)
- about 519,000 ratings (the full database snapshot; further called *FULL* sample)

A.4.4 General Test Setup

The *Taste* framework provides two slightly different implementations of the *Slope One* algorithm.

Both were tested with all four dataset samples (mentioned above). For all tests the `weighted`[7] parameter was set to `true`, and a `CachingRecommender` was used around the `SlopeOneRecommender` and `SlopeOneRecommender2` implementations in order to improve the performance. Additionally the maximum heap size of the *Java* virtual machine was enlarged to `265 MB`.

Since we were basically interested in the applicability to large datasets and the performance of the algorithm, we decided to measure the average response times for the recommendations given for an item the user has already rated and a new item not rated by the user yet. (Which is the basic

[7]The weighted parameter indicates that the implementation acts as a *weighted Slope One Recommender*, also see Section A.2.1 (page 76)

CPU	Pentium 4
CPU Speed	3.00 GHz
Memory Size	1 GB RAM
Harddrive	160 GB, IDE (Western Digital)

Table A.1: *Hardware Specifications of Test Machine*

Software	Version
Windows	XP Service Pack 1
Sun Java	J2SE 1.5.0_07
MySQL	5.0.18 CE
Taste	1.5.2

Table A.2: *Software Components and its Versions used for Testing*

use case as mentioned above in Section A.2.1 on page 76.) For a growing size of the dataset we expect almost linear response times for known items and slightly increasing response times for unknown items.

A.4.5 Test Environment

Hardware Setup

All tests were performed on a Dell Optiplex GX 280 SMT computer. Table A.1 specifies the most important hardware.

Software Setup

All important software components were running on the same machine described in the preceding section. The specific software versions listed in Table A.2 were used for several tests.

A.4.6 Test Results

This section describes the setup and the results for all tests that have been executed.

Note: The response times within the tables belong to both implementations of the `SlopeOneRecommender`. In the rare case that the response times vary, different times are separated by a slash '/'.

Test 1

For the first test we used the basic settings without database indices. Table A.3 (page 81) shows the results. In both figures picturing the response time for known (Figure A.1 on page 83) and unknown (Figure A.2 on page 84) ratings the plot number *1* is assigned to this test.

Test 2

The second test was set up almost identically to the first test, except that more free disk space (about 2 GB) was provided. The average response times for this test are shown in Table A.4. For the figure Figure A.1 (page 83) the corresponding plot number for this test is *2* and in Figure A.2 (page 84) plot numbers *2a* and *2b* are correlated to the curves of this test's results.

Test 3

For the third test the database indices mentioned in Section A.4.2 (page 78) were created additionally to the prior setting from *Test 2* in Section A.4.6. The results can be found in Table A.5 (page 82). Adding indices to the ratings table slightly improved the response times. Similar to the previous test the assigned plot number for Figure A.1 (page 83) is *3* while the plot numbers allotted to the results of this test in Figure A.2 (page 84) are *3a* and *3b*.

[8] The test of the second implementation on the FULL dataset broke when the free disk space was below 1 GB on the hard drive. So we had to re-execute the first test as well, after obtaining more disk space (about 2GB free space).

	10k	50k	100k	FULL
known ratings	78 ms	125 ms	172 ms	859 ms
unknown ratings	656 ms	14 s	117 s	1443 s [8]

Table A.3: *Response Time for Ratings on Different Sizes of Datasets (Test 1)*

	10k	50k	100k	FULL
known ratings	1 ms	15 ms	62 ms	429 ms
unknown ratings	625 ms	10 s	15 / 41 s	1089 / 1158 s

Table A.4: *Response Time for Ratings on Different Sizes of Datasets providing about 2 GB Free Disk Space (Test 2)*

Test 4

The final test on this dataset is based on the settings from *Test 3* in Section A.4.6 (page 81) but uses an additional argument to cap the number of entries to a maximum of one million (which adjusts memory usage at a possible cost of accuracy). Average response times of this test can be found in Table A.6. Capping the maximum number of entries kept in the memory to a million only improved the performance of one of the two algorithm implementations. Plot numbers *4* in Figure A.1 (page 83) as well as *4a* and *4b* in Figure A.2 (page 84) belong to the results of this test.

Test 5

Finally we tested the `SlopeOneRecommender` algorithm on a much larger dataset that was also provided by the *Verisign Communications GmbH*. The database snapshot (from the *Verisign Communications GmbH* asian server from Oct. 15th 2006) contains 7,485,292 user ratings (on items).

It showed that the first test (as well as all succeeding tests) with the large dataset immediately failed due to a lack of memory (and raised an *OutOfMemoryError*) during the initialization phase of both implementations of the `SlopeOneRecommender` algorithm.

	10k	50k	100k	FULL
known ratings	1 ms	15 ms	15 ms	328 ms
unknown ratings	531 ms	8 s	14 / 40 s	929 / 913 s

Table A.5: *Response Time for Ratings on Different Sizes of Datasets providing about 2 GB Free Disk Space using Indices (Test 3)*

	10k	50k	100k	FULL
known ratings	31 ms	16 ms	16 ms	46 ms
unknown ratings	593 ms	5 / 9 s	13 / 14 s	990 / 1616 s

Table A.6: *Response Time for Ratings on Different Sizes of Datasets providing about 2 GB Free Disk Space using Indices and a Cap for Number of Entries (Test 4)*

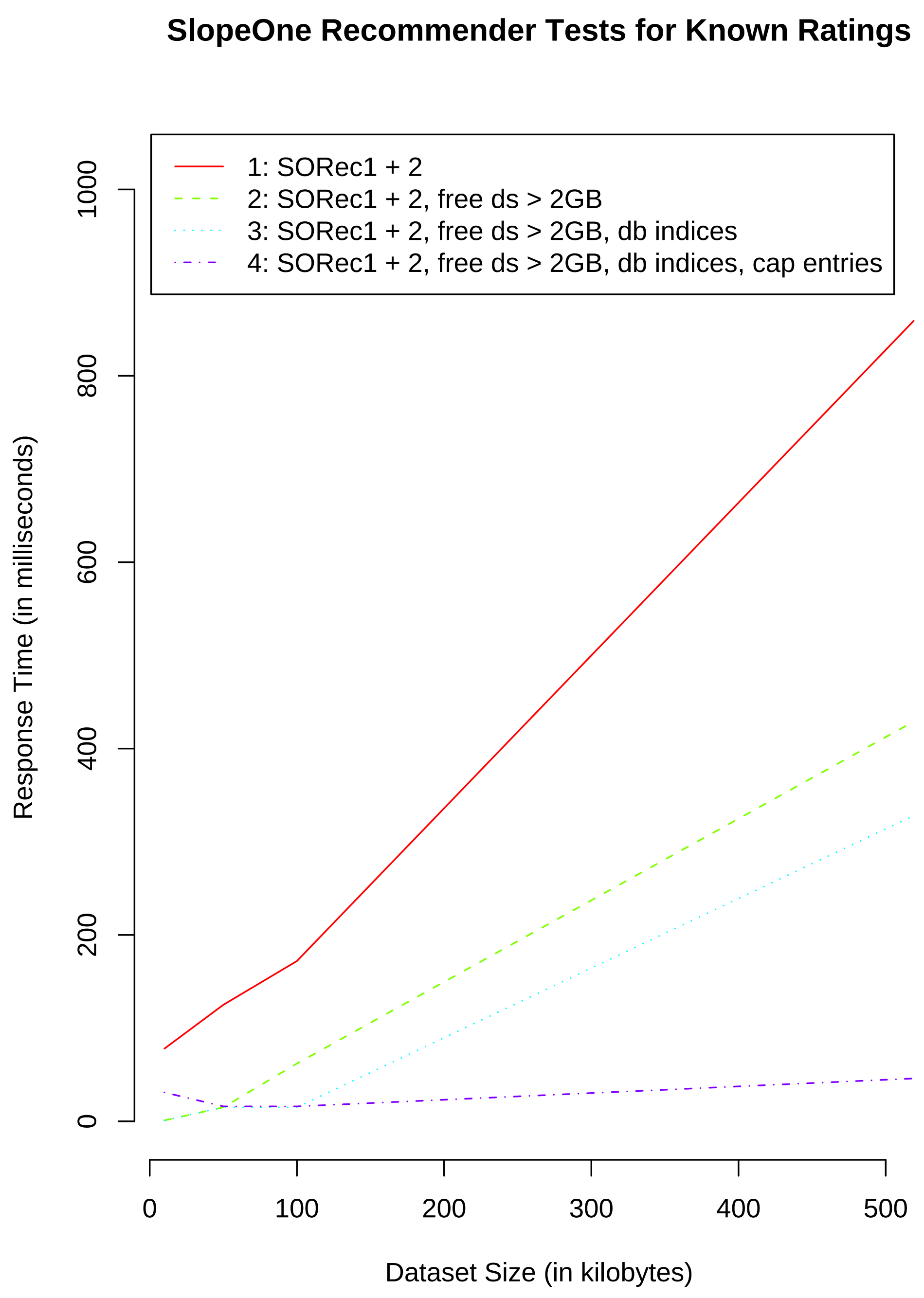


Figure A.1: *Slope One Recommender Tests for Known Ratings*

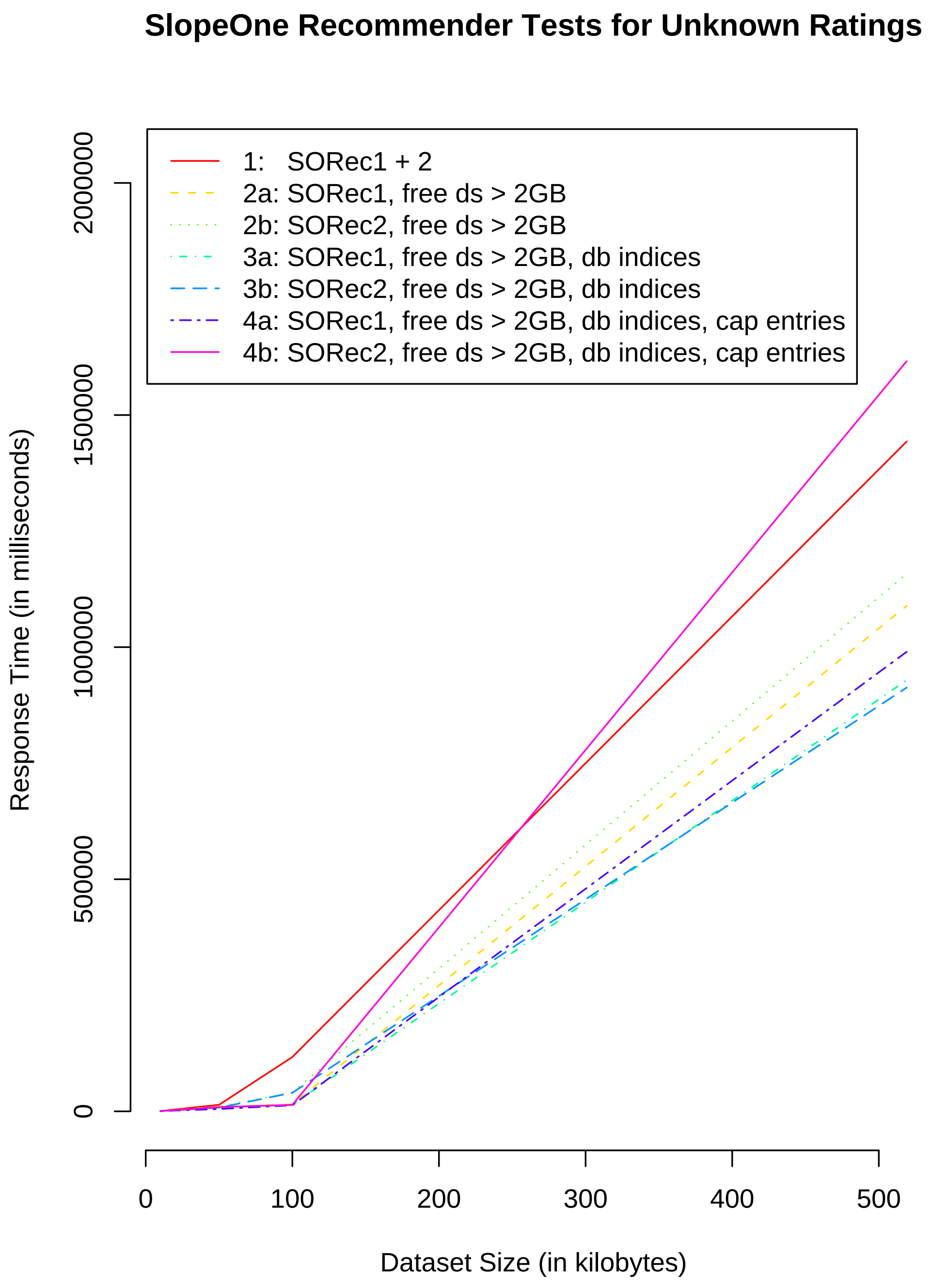


Figure A.2: *Slope One Recommender Tests for Unknown Ratings*

A.5 Conclusion

Our tests indicate that both `SlopeOneRecommender` implementations within the *Taste* framework do not scale well with an increasing number of ratings. Plot number *4* in Figure A.1 (page 83) shows that the curve of the best test-run representing the average response time for *known ratings* is almost linear with the number of ratings after tweaking the settings, while the plots *4a* and *4b* in Figure A.2 (page 84) point out that modifying the configuration improved the computation time slightly, but the performance is still decreasing drastically with a higher number of ratings.

The tested algorithms do not persist already computed data nor do they pre-compute unknown ratings in advance, thus incremental refinement of rating values is not supported. Also the `DataModel` is kept completely in the main memory, which leads to a coherence between the response time and the free disk space on the hard drive caused by page swapping mechanisms of the operating system due to the usage of virtual memory.[9]

Additionally the *Taste* `DataModel` lacks the capability to decide between different item types, and it does not support tenant-specific preferences either. These functionalities must be provided by the application (using the framework) which leads to the need of implementing some additional *glue code*.

We established that the *Taste* framework seems to be well planned, has simple database and file accessors, and provides some state-of-the-art recommendation algorithms that can be easily integrated into our RS, but finally we had to determine that the tested `SlopeOneRecommender` implementation does not perform well on large datasets. We also considered to develop a new scalable database-based `SlopeOneRecommender` implementation that pre-computes some of the necessary data (i.e. item tuples) in order to avoid long response times for the recommendation.

Finally we made the decision to stop investigating the *Taste* framework further, since it would require more effort than implementing the algorithms of our interest (supporting item types as well as different tenants, and most important of all, performing well on large datasets) by ourselves.

[9]Using a significantly larger memory may lead to a better performance (i.e. improved response times for unknown ratings).

A.6 Latest *Taste* Enhancements

The present evaluation was made in the early stage of this project (from March 2007 to April 2007). We reported our findings about the problems with the SO implementations to Sean Owen (the author of the *Taste* framework).[10] Our feedback seemed to inspire Sean Owen to implement an additional database-based `SlopeOneRecommender` promising a better scalability for larger datasets. Since the *Taste* framework release version 1.6 RC1 (from 2007/08/09) this new implementation is contained.

It might be an option to evaluate the improved version of the algorithm, but since the author of the *Taste* framework has announced plans to merge with the *Apache Mahout*[11] project, we decided to wait for the fusion of the two projects.

[10]We'd like to thank Sean Owen once again for his patience and help (via a long e-mail correspondence) concerning several tweaks and improvements.

[11]`http://lucene.apache.org/mahout/`, as of 2008/07/22

Appendix B
Software Infrastructure

Abstract

After an extensive evaluation of available alternatives the technical infrastructure for this project has been designed and implemented as detailed in this chapter. Section B.1 describes the SW components (and APIs) that are used during the runtime, while Section B.2 points out some development tools. The last section (Section B.3) describes additional output.

B.1 Software Components

This section describes the used software runtime components. Table B.1 (page 89) specifies detailed version information.

B.1.1 Programming Language

Since *Java*[1] is one of the most widely used programming languages, it offers integration with almost any available software technology. Since all our other software projects are implemented in *Java*, an easy integration of legacy code is also given.

To make the newest *Java* features available for this project the *Java* version used project wide is J2SE5.0[2]. The most-favoured features are generics that provide compile-time type information for collections and thus lead to fewer runtime bugs and better readability of the source code. Additional features of J2SE5.0 are among others: faster String operations, annotations, language supported enums instead of constants, and enhanced language features, like the new improved `for` loop.

[1]`http://java.sun.com`, as of 2008/08/30

[2]`http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html`, as of 2008/08/30

B.1.2 Dependency Injection and Aspects

The *Spring Framework*[3] is a well-known standard framework for *Java* projects. It supports modern development concepts such as *Dependency Injection* (also known as *Inversion of Control*) and aspect-oriented programming (AOP). Additionally it provides various database connectors.

B.1.3 Database

As it is the commonly most preferred open-source database software, we use *MySQL*[4] as database. It was originally founded and developed by a Swedish company and acquired in 2007 (and since then developed and supported) by the *Sun Microsystems*[5] company. Among other features it supports stored procedures, triggers, sub-queries, and updatable views.

B.1.4 Logging

For a standardised and unified logging process the logging framework *apache.commons.logging*[6] has been chosen which uses *Log4J*[7] per default but is very flexible allowing to plug in other logging frameworks as the *Java util.logging*[8] framework introduced with the J2SE1.4 or any proprietary logging framework.

B.1.5 Caching

For various caching purposes (e.g. for the access of static database content) we use the open-source API *Ehcache*[9].

B.1.6 Web Services and Web Application Server

In order to provide our services to project partners (and customers) we integrated several Web service interfaces. The used Web service stack is

[3]`http://www.springframework.org`, as of 2008/08/30

[4]`http://www.mysql.com`, as of 2008/09/02

[5]`http://www.sun.com`, as of 2008/09/24

[6]`http://jakarta.apache.org/commons/logging`, as of 2008/08/30

[7]`http://logging.apache.org`, as of 2008/08/30

[8]`http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html`, as of 2008/08/30

[9]`http://ehcache.sourceforge.net`, as of 2008/08/30

Category	Software	Version
Programming Language	Sun Java	J2SE 1.5.0_07
Framework	Spring Framework	2.5
Database	MySQL	5.0.18 CE
Logging	Commons Logging	1.2
Logging	Log4j	1.2.12
Caching	Ehcache	1.4.1
Web Services	Metro	1.0
Servlet Container	Apache Tomcat	5.5.17
Profiling	JAMon	2.4

Table B.1: *Versions of Infrastructure Software Components*

Metro[10] which is part if the *Sun Glassfish*[11] project. As servlet container we use the *Apache Tomcat*[12] which is the designated reference implementation for *Java Servlets*[13].

B.1.7 Profiling

For the profiling on method level we applied the *JAMon* API[14] to the most importent interface methods.

B.2 Development Tools

This section describes some of the used development tools. Table B.2 (page 91) specifies detailed version information.

B.2.1 IDE

For the actual process of programming the software we use is the *Eclipse*[15] *Integrated Development Environment* (IDE). It offers a lot of extensions for various tasks, e.g. automated tests, *Maven2*[16] builds, issue tracking systems, version control, and many more.

[10]`https://metro.dev.java.net`, as of 2008/08/30
[11]`https://glassfish.dev.java.net/`, as of 2008/08/30
[12]`http://tomcat.apache.org`, as of 2008/09/02
[13]`http://java.sun.com/products/servlet`, as of 2008/09/02
[14]`http://jamonapi.sourceforge.net`, as of 2008/09/02
[15]`http://www.eclipse.org`, as of 2008/09/02
[16]`http://maven.apache.org`, as of 2008/09/02

B.2.2 Version Control

For the maintenance of source code, configuration files, scripts, and documentation we use the version control system *Subversion*[17]. It provides many advantages over other version control systems such as versioned directories and operations (copy, delete, and rename), efficient binary diffing, quick branching as well as tagging operations, and many more.

B.2.3 Build Process

The build process of *Java* software is managed using *Maven2*, thus introducing a standardised project layout, build lifecycle and produced software artefacts. These features help to make *Java* projects comprehensible across development team bounds and to develop modular software quite easily. Dependency management is another important point in favour of *Maven2* as it provides a solution for all project teams to integrate each other's software artifacts into their build environments. A generic *Maven2* project descriptor has been developed, which contains many useful plugins for generating the *Maven2* project Web site, including test code coverage, heuristic bug detection, changelogs, code style checking, developer activity, dependency analysis and many more.

B.2.4 Continuous Integration

Continuous integration software provides a central point of control. The tool *Continuum*[18] has been selected and evaluated for this purpose as it integrates very well with maven2-managed software projects and the SVN version control system. It allows nightly builds that compile and test specified SW projects at a fixed time, reporting build warnings and errors, as well as test failures via e-mail and through a Web interface.

B.2.5 Testing

In order to provide a stable set of SW components the major components of our RS are supported by a comprehensive set of unit tests. The implementation of these tests is based on the *JUnit*[19] test framework and

[17] `http://subversion.tigris.org`, as of 2008/09/02
[18] `http://maven.apache.org/continuum`, as of 2008/09/02
[19] `http://www.junit.org`, as of 2008/09/02

Category	Software	Version
IDE	Eclipse	SDK 3.2.1., later JEE Europa
Version Control	Subversion	Server 1.2.3 (r15833)
Version Control	Subversion	Client 1.4.3
Build Process	Maven2	2.0.7, later 2.0.8 and 2.0.9
Continuous Integration	Continuum	1.0.3
Testing	JUnit	4.4
Testing	DBUnit	2.2.2
Issue Management	Mantis	1.1.1

Table B.2: *Versions of Infrastructure Development Tools*

integrated with the *Maven2* build environment as well as the continuous integration framework. For unit testing of DAO we use the *DBUnit*[20] API.

B.2.6 Issue Management

For the management of issues (bugs, feature requests and open tasks) we use *Mantis*[21]. It supports complex filtering, multiple projects (with subprojects), e-mail notifications and is fully Web based.

B.3 Additional Output

- A common file structure and a standardised package structure (derived from *Maven2* conventions)
- *JavaDoc* for several classes
- A *Maven2* project site

Summary

In this appendix we pointed out the software components used during runtime and the software tools used in the development process describing their main features and stating their versions.

[20]`http://www.dbunit.org`, as of 2008/09/02
[21]`http://www.mantisbt.org`, as of 2008/09/02

Appendix C
Web Service Interfaces

Abstract

In this Appendix the *Java* interfaces of several existing Web services of the RS are listed. Section C.1 and Section C.2 describe the domain-independent and the music domain recommender interfaces for a Web shop, while Section C.3 and Section C.4 disclose the Web service interfaces for the RASCALLI domain.

C.1 ShopRecommenderWS

Listing C.1 lists the *Java* interface of the domain-independent shop recommender.

```
package at.researchstudio.sat.recommender.service.webapp.
    nodomain;

import at.researchstudio.sat.recommender.model.core.transfer.
    TimeConstraintVO;
import at.researchstudio.sat.recommender.model.webapp.
    RankedItem;
import at.researchstudio.sat.recommender.model.webapp.Rating;
import at.researchstudio.sat.recommender.model.webapp.
    RecommendedItem;
import at.researchstudio.sat.recommender.service.webapp.
    nodomain.exception.ShopRecommenderException;

/**
 * Recommender Webservice wrapper interface
 *
 * @author Roman Cerny
 */
public interface ShopRecommenderWS
{
    // Actions
    /**
     * storing a 'purchase item' action in the recommender
         storage
     */
    public void purchaseItem(Integer tenant, String userId,
        String sessionId, String ip, String itemId, String
        itemType, String description) throws
        ShopRecommenderException;

    /**
     * storing a 'view item' action in the recommender
         storage
     */
    public void viewItem(Integer tenant, String userId,
        String sessionId, String ip, String itemId, String
        itemType, String description) throws
        ShopRecommenderException;

    /**
     * storing a 'rate item' action in the recommender
         storage
     */
    public void rateItem(Integer tenant, String userId,
        String sessionId, String ip, String itemId, String
        itemType, Integer ratingValue, String description)
        throws ShopRecommenderException;

    /**
     * storing a 'search item' action in the recommender
         storage
     */
    public void searchItem(Integer tenant, String userId,
        String sessionId, String ip, String itemId, String
        itemType, Boolean searchSucceeded, Integer
        numberOfFoundItems, String description) throws
        ShopRecommenderException;


    // Rankings
    /**
     * returns 'bought items' (of a given item type for a
         given tenant) ranked by the frequency of 'buy'
         actions
     */
    public RankedItem[] mostBoughtItems(Integer tenant,
        String itemType, Integer numberOfResults,
        TimeConstraintVO timeRange, Boolean sortDescending)
        throws ShopRecommenderException;

    /**
     * returns 'viewed items' (of a given item type for a
         given tenant) ranked by the frequency of 'view'
         actions
     */
    public RankedItem[] mostViewedItems(Integer tenant,
        String itemType, Integer numberOfResults,
        TimeConstraintVO timeRange, Boolean sortDescending)
        throws ShopRecommenderException;

    /**
     * returns 'rated items' (of a given item type for a
         given tenant) ranked by the frequency of 'rate'
         actions
     * rating values are NOT taken into consideration, thus
         simply showing how often an item was rated at all
     */
    public RankedItem[] mostRatedItems(Integer tenant, String
         itemType, Integer numberOfResults, TimeConstraintVO
        timeRange, Boolean sortDescending) throws
        ShopRecommenderException;

    /**
     * returns 'searched items' (of a given item type for a
         given tenant) ranked by the frequency of 'search'
         actions
     */
    public RankedItem[] mostSearchedItems(Integer tenant,
        String itemType, Integer numberOfResults,
        TimeConstraintVO timeRange, Boolean sortDescending)
        throws ShopRecommenderException;


    // Ratings
    /**
     * returns 'ratings' (of a given item type for a given
         tenant, optionally user and session) ordered by the
         'ratingValue'
     */
    public Rating[] itemRatings(Integer tenant, String userId
        , String sessionId, String itemType, Integer
        numberOfResults, TimeConstraintVO timeRange) throws
        ShopRecommenderException;

    /**
     * returns 'bad ratings' (of a given item type for a
         given tenant, optionally user and session) ordered
         by the 'ratingValue'
     * only returns ratings with a value LOWER or EQUAL than
         the tenant specific threshold 'ratingRangeNeutral'
     */
    public Rating[] badItemRatings(Integer tenant, String
        userId, String sessionId, String itemType, Integer
        numberOfResults, TimeConstraintVO timeRange) throws
        ShopRecommenderException;

    /**
     * returns 'good ratings' (of a given item type for a
         given tenant, optionally user and session) ordered
         by the 'ratingValue'
     * only returns ratings with a value HIGHER than the
         tenant specific threshold 'ratingRangeNeutral'
     */
    public Rating[] goodItemRatings(Integer tenant, String
        userId, String sessionId, String itemType, Integer
        numberOfResults, TimeConstraintVO timeRange) throws
        ShopRecommenderException;

    /**
     * returns the 'last good ratings' (of a given item type
         for a given tenant, optionally user and session)
         ordered by the 'actionTime'
     * only returns ratings with a value HIGHER than the
         tenant specific threshold 'ratingRangeNeutral'
     */
    public Rating[] lastGoodItemRatings(Integer tenant,
        String userId, String sessionId, String itemType,
        Integer numberOfResults) throws
        ShopRecommenderException;


    // Recommendations
    /**
     * returns {@link RecommendedItem}s, based on the '
         purchase' history of items with the given '
         consideredItemType',
     * taking 'numberOfLastActionsConsidered' actions into
         consideration,
     * looking for business rules (item assocs) with the
         given 'assocType' and 'requestedItemType' for items
         in the result
     */
    public RecommendedItem[] itemsBasedOnPurchaseHistory(
        Integer tenant, String userId, String sessionId,
        String consideredItemType, Integer
        numberOfLastActionsConsidered, String assocType,
        String requestedItemType) throws
        ShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on the '
         viewing' history of items with the given '
         consideredItemType',
     * taking 'numberOfLastActionsConsidered' actions into
         consideration,
     * looking for business rules (item assocs) with the
         given 'assocType' and 'requestedItemType' for items
         in the result
     */
    public RecommendedItem[] itemsBasedOnViewingHistory(
        Integer tenant, String userId, String sessionId,
        String consideredItemType, Integer
        numberOfLastActionsConsidered, String assocType,
        String requestedItemType) throws
        ShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on the '
         searching' history of items with the given '
         consideredItemType',
     * taking 'numberOfLastActionsConsidered' actions into
         consideration,
     * looking for business rules (item assocs) with the
         given 'assocType' and 'requestedItemType' for items
         in the result
     */
    public RecommendedItem[] itemsBasedOnSearchingHistory(
        Integer tenant, String userId, String sessionId,
        String consideredItemType, Integer
        numberOfLastActionsConsidered, String assocType,
        String requestedItemType) throws
        ShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on business
         rules that identify items as 'bought together'
     */
    public RecommendedItem[] alsoBoughtItems(Integer tenant,
        String userId, String sessionId, String itemId,
        String itemType, String requestedItemType) throws
        ShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on business
         rules that identify items as 'viewed together'
     */
    public RecommendedItem[] alsoViewedItems(Integer tenant,
        String userId, String sessionId, String itemId,
        String itemType, String requestedItemType) throws
        ShopRecommenderException;
    /**
     * returns {@link RecommendedItem}s, based on business
         rules that identify items as 'searched together'
     */
    public RecommendedItem[] alsoSearchedItems(Integer tenant
        , String userId, String sessionId, String itemId,
        String itemType, String requestedItemType) throws
        ShopRecommenderException;


    // Utility methods
    /**
     * returns the possible item types of the given tenant
     */
    public String[] getItemTypes(Integer tenantId) throws
        ShopRecommenderException;

    /**
     * returns the possible association types of the given
         tenant
     */
    public String[] getAssocTypes(Integer tenantId) throws
        ShopRecommenderException;
}
```

Listing C.1: *Domain-Independent Web Service Interface for a Shop Recommender*

C.2 MusicShopRecommenderWS

In Listing C.2 the *Java* interface for the music domain-specific shop recommender is printed.

```
package at.researchstudio.sat.recommender.service.webapp.
    music;

import at.researchstudio.sat.recommender.model.core.transfer.
    TimeConstraintVO;
import at.researchstudio.sat.recommender.model.webapp.
    RankedItem;
import at.researchstudio.sat.recommender.model.webapp.Rating;
import at.researchstudio.sat.recommender.model.webapp.
    RecommendedItem;
import at.researchstudio.sat.recommender.service.webapp.music
    .exception.MusicShopRecommenderException;

/**
 * Music Recommender Webservice wrapper interface (for the
     music domain)
 *
 * @author Roman Cerny
 */
public interface MusicShopRecommenderWS
{
    // Actions
    /**
     * storing a 'purchase track' action in the recommender
         storage
     */
    public void purchaseTrack(Integer tenantId, String userId
        , String sessionId, String ip, String trackId, String
         description) throws MusicShopRecommenderException;

    /**
     * storing a 'view artist' action in the recommender
         storage
     */
    public void viewArtist(Integer tenantId, String userId,
        String sessionId, String ip, String artistId, String
        description) throws MusicShopRecommenderException;

    /**
     * storing a 'view genre' action in the recommender
         storage
     */
    public void viewGenre(Integer tenantId, String userId,
        String sessionId, String ip, String genreId, String
        description) throws MusicShopRecommenderException;

    /**
     * storing a 'view track' action in the recommender
         storage
     */
    public void viewTrack(Integer tenantId, String userId,
        String sessionId, String ip, String trackId, String
        description) throws MusicShopRecommenderException;

    /**
     * storing a 'rate artist' action in the recommender
         storage
     */
    public void rateArtist(Integer tenantId, String userId,
        String sessionId, String ip, String artistId, Integer
         ratingValue, String description) throws
        MusicShopRecommenderException;

    /**
```

```
     * storing a 'rate track' action in the recommender
         storage
     */
    public void rateTrack(Integer tenantId, String userId,
        String sessionId, String ip, String trackId, Integer
        ratingValue, String description) throws
        MusicShopRecommenderException;

    /**
     * storing a 'search artist' action in the recommender
         storage
     */
    public void searchArtist(Integer tenantId, String userId,
         String sessionId, String ip, String artistId,
        Boolean searchSucceeded, Integer numberOfFoundArtists
        , String description) throws
        MusicShopRecommenderException;

    /**
     * storing a 'search track' action in the recommender
         storage
     */
    public void searchTrack(Integer tenantId, String userId,
        String sessionId, String ip, String trackId, Boolean
        searchSucceeded, Integer numberOfFoundTracks, String
        description) throws MusicShopRecommenderException;

    /**
     * storing a 'preview track' action in the recommender
         storage
     */
    public void previewTrack(Integer tenantId, String userId,
         String sessionId, String ip, String trackId, String
        description) throws MusicShopRecommenderException;

    /**
     * storing a 'add track to playlist' action in the
         recommender storage
     */
    public void addTrackToPlaylist(Integer tenantId, String
        userId, String sessionId, String ip, String trackId,
        String description) throws
        MusicShopRecommenderException;

    // Rankings
    /**
     * returns 'bought tracks' (for a given tenant) ranked by
          the frequency of 'buy' actions
     */
    public RankedItem[] mostBoughtTracks(Integer tenantId,
        Integer numberOfResults, TimeConstraintVO timeRange,
        Boolean sortDescending) throws
        MusicShopRecommenderException;

    /**
     * returns 'viewed artists' (for a given tenant) ranked
         by the frequency of 'view' actions
     */
    public RankedItem[] mostViewedArtists(Integer tenantId,
        Integer numberOfResults, TimeConstraintVO timeRange,
        Boolean sortDescending) throws
        MusicShopRecommenderException;

    /**
     * returns 'viewed genres' (for a given tenant) ranked by
          the frequency of 'view' actions
     */
    public RankedItem[] mostViewedGenres(Integer tenantId,
        Integer numberOfResults, TimeConstraintVO timeRange,
        Boolean sortDescending) throws
        MusicShopRecommenderException;

    /**
     * returns 'viewed tracks' (for a given tenant) ranked by
          the frequency of 'view' actions
     */
    public RankedItem[] mostViewedTracks(Integer tenantId,
        Integer numberOfResults, TimeConstraintVO timeRange,
        Boolean sortDescending) throws
        MusicShopRecommenderException;

    /**
     * returns 'rated artists' (for a given tenant) ranked by
          the frequency of 'rate' actions
     */
    public RankedItem[] mostRatedArtists(Integer tenantId,
        Integer numberOfResults, TimeConstraintVO timeRange,
        Boolean sortDescending) throws
        MusicShopRecommenderException;

    /**
     * returns 'rated tracks' (for a given tenant) ranked by
         the frequency of 'rate' actions
     */
    public RankedItem[] mostRatedTracks(Integer tenantId,
        Integer numberOfResults, TimeConstraintVO timeRange,
        Boolean sortDescending) throws
        MusicShopRecommenderException;
    /**
     * returns 'searched artists' (for a given tenant) ranked
         by the frequency of 'search' actions
     */
    public RankedItem[] mostSearchedArtists(Integer tenantId,
         Integer numberOfResults, TimeConstraintVO timeRange,
         Boolean sortDescending) throws
        MusicShopRecommenderException;

    /**
     * returns 'searched tracks' (for a given tenant) ranked
        by the frequency of 'search' actions
     */
    public RankedItem[] mostSearchedTracks(Integer tenantId,
        Integer numberOfResults, TimeConstraintVO timeRange,
        Boolean sortDescending) throws
        MusicShopRecommenderException;

    /**
     * returns 'previewed Tracks' (for a given tenant) ranked
         by the frequency of 'preview' actions
     */
    public RankedItem[] mostPreviewedTracks(Integer tenantId,
         Integer numberOfResults, TimeConstraintVO timeRange,
         Boolean sortDescending) throws
        MusicShopRecommenderException;

    /**
     * returns 'tracks that have been added to a playlist' (
        for a given tenant) ranked by the frequency of 'add
        to playlist' actions
     */
    public RankedItem[] mostAddedToPlaylistTracks(Integer
        tenantId, Integer numberOfResults, TimeConstraintVO
        timeRange, Boolean sortDescending) throws
        MusicShopRecommenderException;

    // Ratings
    /**
     * returns 'ratings' over artists (for a given tenant,
         optionally user and session) ordered by the '
         ratingValue'
     */
    public Rating[] artistRatings(Integer tenantId, String
        userId, String sessionId, Integer numberOfResults,
        TimeConstraintVO timeRange) throws
        MusicShopRecommenderException;
    /**
     * returns 'bad ratings' over artists (for a given tenant
         , optionally user and session) ordered by the '
         ratingValue'
     * only returns ratings with a value LOWER or EQUAL than
         the tenant specific threshold 'ratingRangeNeutral'
     */
    public Rating[] badArtistRatings(Integer tenantId, String
         userId, String sessionId, Integer numberOfResults,
        TimeConstraintVO timeRange) throws
        MusicShopRecommenderException;

    /**
     * returns 'good ratings' over artists (for a given
         tenant, optionally user and session) ordered by the
         'ratingValue'
     * only returns ratings with a value HIGHER than the
         tenant specific threshold 'ratingRangeNeutral'
     */
    public Rating[] goodArtistRatings(Integer tenantId,
        String userId, String sessionId, Integer
        numberOfResults, TimeConstraintVO timeRange) throws
        MusicShopRecommenderException;

    /**
     * returns the 'last good ratings' over artists (for a
         given tenant, optionally user and session) ordered
         by the 'actionTime'
     * only returns ratings with a value HIGHER than the
         tenant specific threshold 'ratingRangeNeutral'
     */
    public Rating[] lastGoodArtistRatings(Integer tenantId,
        String userId, String sessionId, Integer
        numberOfResults) throws MusicShopRecommenderException
        ;

    /**
     * returns 'ratings' over tracks (for a given tenant,
         optionally user and session) ordered by the '
         ratingValue'
     */
    public Rating[] trackRatings(Integer tenantId, String
        userId, String sessionId, Integer numberOfResults,
        TimeConstraintVO timeRange) throws
        MusicShopRecommenderException;

    /**
     * returns 'bad ratings' over tracks (for a given tenant,
          optionally user and session) ordered by the '
        ratingValue'
     * only returns ratings with a value LOWER or EQUAL than
        the tenant specific threshold 'ratingRangeNeutral'
     */
    public Rating[] badTrackRatings(Integer tenantId, String
        userId, String sessionId, Integer numberOfResults,
        TimeConstraintVO timeRange) throws
        MusicShopRecommenderException;

    /**
     * returns 'good ratings' over tracks (for a given tenant
        , optionally user and session) ordered by the '
        ratingValue'
     * only returns ratings with a value HIGHER than the
        tenant specific threshold 'ratingRangeNeutral'
     */
    public Rating[] goodTrackRatings(Integer tenantId, String
         userId, String sessionId, Integer numberOfResults,
        TimeConstraintVO timeRange) throws
        MusicShopRecommenderException;

    /**
     * returns the 'last good ratings' over tracks (for a
        given tenant, optionally user and session) ordered
        by the 'actionTime'
     * only returns ratings with a value HIGHER than the
        tenant specific threshold 'ratingRangeNeutral'
     */
    public Rating[] lastGoodTrackRatings(Integer tenantId,
        String userId, String sessionId, Integer
        numberOfResults) throws MusicShopRecommenderException
        ;

    // Recommendations
    /**
     * returns {@link RecommendedItem}s, based on the '
        purchase' history of track items,
     * taking 'numberOfLastActionsConsidered' actions into
        consideration,
     * looking for business rules (item assocs) with the
        given 'assocType' and track items in the result
     */
    public RecommendedItem[] tracksBasedOnPurchaseHistory(
        Integer tenantId, String userId, String sessionId,
        Integer numberOfLastActionsConsidered, String
        assocType) throws MusicShopRecommenderException;

    /**
```

```
     * returns {@link RecommendedItem}s, based on the 'view'
         history of items with the given 'consideredItemType
         ',
     * taking 'numberOfLastActionsConsidered' actions into
         consideration,
     * looking for business rules (item assocs) with the
         given 'assocType' and artist items in the result
     */
    public RecommendedItem[] artistsBasedOnViewingHistory(
        Integer tenantId, String userId, String sessionId,
        String consideredItemType, Integer
        numberOfLastActionsConsidered, String assocType)
        throws MusicShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on the 'view'
         history of items with the given 'consideredItemType
         ',
     * taking 'numberOfLastActionsConsidered' actions into
         consideration,
     * looking for business rules (item assocs) with the
         given 'assocType' and genre items in the result
     */
    public RecommendedItem[] genresBasedOnViewingHistory(
        Integer tenantId, String userId, String sessionId,
        String consideredItemType, Integer
        numberOfLastActionsConsidered, String assocType)
        throws MusicShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on the 'view'
         history of items with the given 'consideredItemType
         ',
     * taking 'numberOfLastActionsConsidered' actions into
         consideration,
     * looking for business rules (item assocs) with the
         given 'assocType' and track items in the result
     */
    public RecommendedItem[] tracksBasedOnViewingHistory(
        Integer tenantId, String userId, String sessionId,
        String consideredItemType, Integer
        numberOfLastActionsConsidered, String assocType)
        throws MusicShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on the 'search
         ' history of items with the given '
         consideredItemType',
     * taking 'numberOfLastActionsConsidered' actions into
         consideration,
     * looking for business rules (item assocs) with the
         given 'assocType' and artist items in the result
     */
    public RecommendedItem[] artistsBasedOnSearchingHistory(
        Integer tenantId, String userId, String sessionId,
        String consideredItemType, Integer
        numberOfLastActionsConsidered, String assocType)
        throws MusicShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on the 'search
         ' history of items with the given '
         consideredItemType',
     * taking 'numberOfLastActionsConsidered' actions into
         consideration,
     * looking for business rules (item assocs) with the
         given 'assocType' and tracks items in the result
     */
    public RecommendedItem[] tracksBasedOnSearchingHistory(
        Integer tenantId, String userId, String sessionId,
        String consideredItemType, Integer
        numberOfLastActionsConsidered, String assocType)
        throws MusicShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on business
         rules that identify tracks as 'bought together'
     */
    public RecommendedItem[] alsoBoughtTracks(Integer
        tenantId, String userId, String sessionId, String
        trackId) throws MusicShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on business
         rules that identify artists as 'viewed together'
         with items of the given item type
     */
    public RecommendedItem[] alsoViewedArtists(Integer
        tenantId, String userId, String sessionId, String
        itemId, String itemType) throws
        MusicShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on business
         rules that identify genres as 'viewed together' with
          items of the given item type
     */
    public RecommendedItem[] alsoViewedGenres(Integer
        tenantId, String userId, String sessionId, String
        itemId, String itemType) throws
        MusicShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on business
         rules that identify tracks as 'viewed together' with
          items of the given item type
     */
    public RecommendedItem[] alsoViewedTracks(Integer
        tenantId, String userId, String sessionId, String
        itemId, String itemType) throws
        MusicShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on business
         rules that identify artists as 'searched together'
         with items of the given item type
     */
    public RecommendedItem[] alsoSearchedArtists(Integer
        tenantId, String userId, String sessionId, String
        itemId, String itemType) throws
        MusicShopRecommenderException;

    /**
     * returns {@link RecommendedItem}s, based on business
         rules that identify tracks as 'searched together'
         with items of the given item type
     */
    public RecommendedItem[] alsoSearchedTracks(Integer
        tenantId, String userId, String sessionId, String
        itemId, String itemType) throws
        MusicShopRecommenderException;

    // Utility methods
    /**
     * returns the possible item types of the given tenant
     */
    public String[] getItemTypes(Integer tenant) throws
        MusicShopRecommenderException;

    /**
     * returns the possible association types of the given
         tenant
     */
    public String[] getAssocTypes(Integer tenant) throws
        MusicShopRecommenderException;
```

```
}
```

Listing C.2: *Music Domain Web Service Interface for a Shop Recommender*

C.3 RascalliDFKIWS

The Listing C.3 reveals the *Java* interface for the RASCALLI domain-specific web service, used within the DFKI *Visual Browser* for the gathering of user actions.

```
package at.researchstudio.sat.recommender.service.webapp.
    rascalli;

import at.researchstudio.sat.recommender.service.webapp.
    rascalli.exception.RascalliDFKIProfilerException;

/**
 * Rascalli DFKI Webservice wrapper interface
 *
 * @author Roman Cerny
 */
public interface RascalliDFKIWS
{
    // main topics
    /**
     * storing a 'choose_topic' action (with the topic "
         BACKGROUND_INFORMATION") for the given user of a
         tenant for the given artist in the recommender
         storage
     */
    public void chooseArtistTopicBackgroundInformation(
        Integer tenantId, String userId, String sessionId,
        String ip, String artistId) throws
        RascalliDFKIProfilerException;

    /**
     * storing a 'choose_topic' action (with the topic "
         CONNECTION") for the given user of a tenant for the
         given artist in the recommender storage
     */
    public void chooseArtistTopicConnection(Integer tenantId,
         String userId, String sessionId, String ip, String
        artistId) throws RascalliDFKIProfilerException;

    /**
     * storing a 'choose_topic' action (with the topic "
         PROFESSIONAL_CAREER") for the given user of a tenant
         for the given artist in the recommender storage
     */
    public void chooseArtistTopicProfessionalCareer(Integer
        tenantId, String userId, String sessionId, String ip,
         String artistId) throws
        RascalliDFKIProfilerException;

    // sub topics of 'connections'
    /**
     * storing a 'choose_topic' action (with the topic "
        CONNECTION_PARENT") for the given user of a tenant
        for the given artist in the recommender storage
     */
    public void chooseArtistTopicConnectionParent(Integer
        tenantId, String userId, String sessionId, String ip,
         String artistId) throws
        RascalliDFKIProfilerException;

    /**
     * storing a 'choose_topic' action (with the topic "
        CONNECTION_SIBLING") for the given user of a tenant
        for the given artist in the recommender storage
     */
    public void chooseArtistTopicConnectionSibling(Integer
        tenantId, String userId, String sessionId, String ip,
         String artistId) throws
        RascalliDFKIProfilerException;

    /**
     * storing a 'choose_topic' action (with the topic "
        CONNECTION_CHILD") for the given user of a tenant
        for the given artist in the recommender storage
     */
    public void chooseArtistTopicConnectionChild(Integer
        tenantId, String userId, String sessionId, String ip,
         String artistId) throws
        RascalliDFKIProfilerException;

    /**
     * storing a 'choose_topic' action (with the topic "
        CONNECTION_PARTNER") for the given user of a tenant
        for the given artist in the recommender storage
     */
    public void chooseArtistTopicConnectionPartner(Integer
        tenantId, String userId, String sessionId, String ip,
         String artistId) throws
        RascalliDFKIProfilerException;

    // sub-topics of 'professional career'
    /**
     * storing a 'choose_topic' action (with the topic "
         PROFESSIONAL_CAREER_PRICES") for the given user of a
          tenant for the given artist in the recommender
         storage
     */
    public void chooseArtistTopicProfessionalCareerPrices(
        Integer tenantId, String userId, String sessionId,
        String ip, String artistId) throws
        RascalliDFKIProfilerException;

    /**
     * storing a 'choose_topic' action (with the topic "
         PROFESSIONAL_CAREER_RECORDS") for the given user of
         a tenant for the given artist in the recommender
         storage
     */
    public void chooseArtistTopicProfessionalCareerRecords(
        Integer tenantId, String userId, String sessionId,
        String ip, String artistId) throws
        RascalliDFKIProfilerException;

    // general method for other artist topics
    /**
     * storing a 'choose_topic' action (with the passed topic
         ) for the given user of a tenant for the given
         artist in the recommender storage
     */
    public void chooseArtistTopicByString(Integer tenantId,
        String userId, String sessionId, String ip, String
        artistId, String topic) throws
        RascalliDFKIProfilerException;
}
```

Listing C.3: *RASCALLI Domain Web Service Interface for the DFKI* Visual Browser

C.4 RascalloModellingWS

Finally Listing C.4 describes the *Java* interface for the RASCALLI domain-specific web service, used within the SAT *RASCALLI Environment* for the presentation of user preferences.

```
package at.researchstudio.sat.recommender.service.webapp.
    rascalli;

import at.researchstudio.sat.recommender.service.webapp.
    rascalli.exception.RascalloModellingProfilerException;
```

```java
import at.researchstudio.sat.recommender.service.webapp.
    rascalli.model.Preference;
import at.researchstudio.sat.recommender.service.webapp.
    rascalli.model.TopicCounter;

/**
 * Rascalli Webservice wrapper interface
 *
 * @author Roman Cerny
 */
public interface RascalloModellingWS
{
    /**
     * returns the artist preferences of a given user of a
         tenant using only actions with a specific sessionId
     */
    public Preference[] artistPreferencesForSession(Integer
        tenantId, String userId, String sessionId) throws
        RascalloModellingProfilerException;

    /**
     * returns the genre preferences of a given user of a
         tenant using only actions with a specific sessionId
     */
    public Preference[] genrePreferencesForSession(Integer
        tenantId, String userId, String sessionId) throws
        RascalloModellingProfilerException;

    /**
     * returns the track preferences of a given user of a
         tenant using only actions with a specific sessionId
     */
    public Preference[] trackPreferencesForSession(Integer
        tenantId, String userId, String sessionId) throws
        RascalloModellingProfilerException;

    /**
     * returns the topic counters of a given user of a tenant
          using only actions with a specific sessionId
     */
    public TopicCounter[] topicCountersForSession(Integer
        tenantId, String userId, String sessionId) throws
        RascalloModellingProfilerException;

    /**
     * returns the artist preferences of a given user of a
         tenant
     */
    public Preference[] artistPreferencesForUser(Integer
        tenantId, String userId) throws
        RascalloModellingProfilerException;

    /**
     * returns the genre preferences of a given user of a
         tenant
     */
    public Preference[] genrePreferencesForUser(Integer
        tenantId, String userId) throws
        RascalloModellingProfilerException;

    /**
     * returns the track preferences of a given user of a
         tenant
     */
    public Preference[] trackPreferencesForUser(Integer
        tenantId, String userId) throws
        RascalloModellingProfilerException;

    /**
     * returns the topic Counters of a given user of a tenant
     */
    public TopicCounter[] topicCountersForUser(Integer
        tenantId, String userId) throws
        RascalloModellingProfilerException;

    /**
     * returns the artist preferences of a given a tenant (
         using all actions from all users within all sessions
         )
     */
    public Preference[] artistPreferences(Integer tenantId)
        throws RascalloModellingProfilerException;

    /**
     * returns the genre preferences of a given a tenant (
         using all actions from all users within all sessions
         )
     */
    public Preference[] genrePreferences(Integer tenantId)
        throws RascalloModellingProfilerException;

    /**
     * returns the track preferences of a given a tenant (
         using all actions from all users within all sessions
         )
     */
    public Preference[] trackPreferences(Integer tenantId)
        throws RascalloModellingProfilerException;

    /**
     * returns the topic counters of a given user of a tenant
     */
    public TopicCounter[] topicCounters(Integer tenantId)
        throws RascalloModellingProfilerException;

    /**
     * activates the 'pea counter' strategy, using frequency
         counts of actions (for all action types)
     */
    public void activatePeaCounterStrategy() throws
        RascalloModellingProfilerException;

    /**
     * activates the 'weakening over time' strategy, using
         preference weakening over time
     */
    public void activateWeakeningOverTimeStrategy() throws
        RascalloModellingProfilerException;

    /**
     * checks if the 'pea counter' strategy is currently the
         active strategy
     */
    public boolean isPeaCounterStrategyActivated() throws
        RascalloModellingProfilerException;

    /**
     * checks if the 'weakening over time' strategy is
         currently the active strategy
     */
    public boolean isWeakeningOverTimeStrategyActivated()
        throws RascalloModellingProfilerException;
}
```

Listing C.4: *RASCALLI Domain Web Service Interface for the SAT* RASCALLI Environment

Summary

In this appendix we listed several methods of *Java* interfaces for various existing Web services of the RS.