A framework for the rapid prototyping of knowledge-based recommender systems in the learning domain

Almudena Ruiz-Iniesta, Guillermo Jiménez-Díaz and Mercedes Gómez-Albarrán
Dep. of Software Engineering and Artificial Intelligence
Complutense University of Madrid, Spain
almudenari@fdi.ucm.es, gjimenez@fdi.ucm.es, albarran@sip.ucm.es

Abstract. In this paper we present a framework for the rapid prototyping of knowledge-based recommender systems applied to learning object recommendation. With a recommendation scheme of five stages as starting point, the framework can be configured and adapted to build different recommenders. The framework not only provides default implementations of alternative strategies for each stage, but can easily be extended with new implementations. Finally, we exemplify the use of the framework by implementing two different recommenders.

Keywords: recommender systems, frameworks, learning objects

1 Introduction

The availability of educational resources in electronic repositories facilitates student self-learning as a complementary activity to lectures in the classroom. An example of these repositories is Learning Object Repositories (LOR), which provide the mechanism to share, manage and use educational resources. Some examples of LOR are ARIADNE (ARIADNE 2006), MERLOT (Schell & Burns 2002) or MACE (Stefaner et al. 2007). Usually, the large number of resources that exist in these repositories makes access difficult. Most LORs have a simple search engine that suggests resources without taking into account the particular characteristics and needs of the target student. Therefore, it is necessary to provide support for personalized access to the resources that suit the needs, goals and preferences of the students.

Recommender systems support users in pre-selecting information they may be interested in. There are three main classes of recommender systems based on the kind of knowledge source employed (Jannach et al. 2010): collaborative, content-based and knowledge-based recommenders. Collaborative recommenders exploit preferences, past behaviour or opinions of an existing user community and apply collaborative filtering techniques to predict which items the current user will most probably like or be interested in. In order to apply collaborative filtering techniques nothing has to be known about the items to be recommended, but user ratings are needed. Content-based recommenders employ item descriptions in order to make a recommenda-
tion. Usually item descriptions are compared with the current user profile, which also describes her interests in terms of descriptions of her preferred items. Unlike collaborative recommenders, content-based recommenders do not require the existence of a large user community or a rating history, and recommendation lists can be generated even if there is only one single user. Finally, knowledge-based recommenders typically make use of additional information about both the current user and the available items (for instance: a similarity function to compare the items, domain ontologies, explicit recommendation rules, etc.). This additional information is employed in order to generate the recommendations. In many situations recommender systems do not apply a single recommendation technique, leading to hybrid recommenders that explore how to combine some basic recommendation techniques (for example, content-based and collaborative) in order to find some synergies and improve the recommendation experience. Recommender systems have traditionally been applied in the field of e-commerce (Wei et al. 2007). However, our work is included in a leading area of research that transfers recommendation techniques to the academic field (Manouselis et al. 2011). More specifically, our work focuses on a poorly-exploited application of recommender systems that entails providing personalized recommendation support to access LORs.

We are conscious that the e-learning field imposes specific requirements on the recommendation process, such as taking into account the cognitive state of the learners, their goals and/or preferences, or using pedagogic strategies as guiding principles for recommendation. Thus, we have explored different recommendation alternatives in the last years.

Our first approach to Learning Objects (LO) recommendation proposed a case-based recommender system (a subtype of content-based recommenders that employs case-based reasoning techniques in order to generate the recommendation) that suggested those LOs that were most similar to a student query (Gómez-Albarrán & Jiménez-Díaz 2009). This case-based strategy provided a weak personalization experience because it only took into account the query proposed by the student. Therefore, this case-based strategy suffered from the overspecialization problem that commonly affects pure-similarity approaches: only those LOs that are highly correlated with the student query are candidates to be recommended.

Since then, our efforts (Ruiz-Iniesta et al. 2009, Ruiz-Iniesta et al. 2011) have focused on analysing strategies that could address these two handicaps: weak personalization and overspecialization. The alternatives that we have proposed belong to the field of knowledge-based recommenders and rely on the existence of a knowledge base with information about the sources involved in the recommendation process: LOs and student profiles. In particular:

- The LOs have been developed according to the Learning Object Metadata (LOM) standard. Each LO has information about the concepts in the field of study that it covers.
- The student profile stores information about the goals she achieved in the learning process. The learning goals attained by a student are represented by the concepts that she should know and the mastery level achieved in each of them.
The knowledge base also includes a domain ontology populated with concepts in the concrete field of study. Ontology concepts are organized in a taxonomy using the typical is_a relation and are used for indexing the LOs and for representing the student profile.

The analysis and comparison of our alternative recommendation strategies required the development of multiple prototypes that implement them. In this context, we have sought solutions for a rapid prototyping of our recommendation strategies. Nowadays, there exist some libraries that help in the development of recommender systems, for example: Mahout\(^1\), Duine\(^2\), MyMediaLite (Gantner et al. 2011) or Lenskit (Ekstrand et al. 2011). All of them are focused on the implementation of collaborative filtering algorithms, so they do not provide support for knowledge-based approaches. JColibri (Recio-García et al. 2008) is an alternative that supports the development of case-based recommender systems and also has the basic tools needed for the development of knowledge-based and collaborative recommenders. However, JColibri focuses on the generation of recommendations based on measures of similarity with the query, while ignoring aspects of personalization which are essential in LO recommendation. Consequently, none of these existing libraries were adapted to the research area that we were exploring. Therefore, we decided to design a framework that allows rapid prototyping of knowledge-based recommenders for the learning domain.

A framework is a reusable semi-complete architecture for developing applications in a specific domain, which allows code and design reuse (Pree 1994). Rapid prototyping is an approach which allows us to take crucial design decisions as early as possible. A rapid prototyping system should support maximal re-use and innovative combinations of existing methods, as well as the simple and quick integration of new ones. Our framework was designed with these principles in mind. In this sense, our work represents a significant contribution in the field of frameworks for knowledge-based recommender systems.

This paper describes the framework proposed and is organized as follows. Section 2 describes the recommendation stages of the recommendation process. Section 3 describes the framework for the LO recommendation that we have developed, detailing the main classes (section 3.1), the hooks (section 3.2) and the default implementations provided for several strategies (section 3.3). Section 4 exemplifies the use of the framework for the development of two knowledge-based recommenders. Last section concludes the paper and outlines some lines of future work.

2 Stages of the recommendation process

The recommendation process is structured in five stages (Figure 1): the query elicitation stage, retrieval stage, filtering stage, rating stage and selection stage. Each

\(^1\) http://mahout.apache.org/
\(^2\) http://www.duineframework.org/
stage can be completed by following different approaches, which adds variability and flexibility to the recommendation process. Let us briefly explain each stage:

- **Query elicitation stage**: Student preferences are commonly acquired by using a query. This is a reactive strategy because the system reacts to a user query. Additionally, the recommendation process can start in a proactive way. In this case, the system takes the initiative and suggests a recommendation, for example, based on the information stored in the user profile.

- **Retrieval stage**: This stage is responsible for retrieving the resources that are candidates to generate a recommendation. The retrieval method to be employed depends on how the LOs are indexed in the recommender. Although our interests focus on exploiting the use of an ontology for indexing the LOs, we should not ignore the existence of other indexing alternatives. Regardless of how the LOs are indexed, we consider that it is crucial to provide a flexible retrieval process according to the terms included in the query or the information obtained by the recommender in a proactive way. For instance, if there are no LOs that strictly satisfy the student query, this flexible retrieval could suggest LOs indexed by a subset of the (same or similar) concepts in the query.

- **Filtering stage**: As stated before, the learning domain imposes new restrictions to the recommendation process related to personalization. For example, although two students could pose the same query to the system, the LOs retrieved could be completely useful for one of them but not for the other one because their mastery levels differ or they have different learning styles. Therefore, the filtering stage may be interesting or necessary for filtering the LOs that are useless according to the student profile, by taking into account dimensions like her mastery level, her previous interests or her learning style, among others. While this filtering process could be part of the retrieval stage, we prefer to consider it separately to add more flexibility to the resulting recommendation approaches.

- **Rating stage**: A recommendation is generated by using the most useful LOs for the target student and the query posed. This usefulness is estimated by defining a quality metric, which rates the LOs according to the attributes related to the LO, the query and the target student. Although in our first approach we proposed a quality metric that took into account the similarity between the query and the
A framework for the rapid prototyping of knowledge-based recommender systems in the learning domain

concepts covered by the LO, we consider that the quality of an LO can be assessed according to different aspects, like the pedagogical utility of the LO to the student, its correlation to the student’s interests or learning styles, etc. Moreover, the quality metric should not be limited to a single relevancy, but can be measured as an aggregation of different relevancies. The rating stage lets us explore the use of different quality metrics and the use of diverse aggregation functions to combine several quality metrics.

• **Selection stage:** Although the retrieval and filtering processes reduce the number of candidates to generate the final recommendation, this number is usually large enough to need a selection of the most interesting candidates to be recommended. Besides, we believe that the learning domain imposes a strong restriction on the size of the recommendation. Providing a long list of LOs to practice can produce an overwhelming effect on the student. Commonly, the way to shorten the recommendation is by limiting the list to the $k$ most useful candidates according to the quality metric. However, we cannot obviate that a short recommendation can be overspecialized, so that the LOs included in the recommendations are highly correlated to each other. This way, if the first LOs are not interesting for the student, probably none of the LOs selected will have any interest. For this reason, we consider that the inclusion of approaches that add diversity to the LOs recommended is mandatory when the final candidates are selected.

3 A framework for learning object recommendation

The analysis of variability aspects in a knowledge-based recommendation process has evolved into the definition of a framework for the rapid prototyping of these recommender systems. First, the variability aspects of the recommendation were identified as the stages of the recommendation process described in Section 2. Subsequently, we defined the abstract classes of the framework in charge of the control flow and responsible for executing the stages in an orderly fashion. Next, we defined the *hooks* in the framework (Pree 1994), those predefined points that need to be configured and where subclasses and specific methods in the recommender must be implemented. Finally, we created a set of classes that inherit from the abstract classes defined for this framework and implemented specific strategies used in our prototypes. Thus, the development of new recommender systems becomes easier, as we will show in Section 3.3. The end result is the class diagram depicted in Figures 2 and 3. Figure 2 depicts the top levels of the class hierarchy. Figure 3 depicts the rest of the classes contained in the framework, organized according to the stages in the recommendation process. Next, we describe the whole class design.

3.1 The main classes in the framework

The main class in the framework is **CBRecommender**. This class contains the infrastructure for implementing a knowledge-based recommender of LOs. This class is also responsible for controlling the execution of the recommendation process. The
configuration parameters of the recommendation process are stored following a blackboard architecture (Stegemann et al. 2007). Any recommender implemented using this framework needs to create at least a subclass of `CBRecommender`. It has to implement the hooks that will be presented in section 3.2, or it can use some of the default implementations provided by the framework.

![Class diagram in UML of the proposed architecture](image)

Fig. 2. Class diagram in UML of the proposed architecture (framework hooks appear in italics): main class and direct subclasses.

Each stage of the recommendation process corresponds to a framework class:

- The `QueryElicitationStage` class is responsible for creating the query that will be employed to start the recommendation process.
- The `RetrievalStage` class corresponds to the stage responsible for generating the set of LO candidates from the query.
- The `FilteringStage` class is responsible for making a first filtering of the set of candidates from the previous stage. Its behaviour is common to all recommenders: iterating over the set of LOs and deciding which LOs should pass to the next stage. Therefore, this stage relies on a `Filter` object (according to the `Strategy` design pattern (Gamma et al. 1995)). This object will decide whether an LO will be removed from the set of candidates.
- The `RatingStage` class is responsible for assigning quality values to each LO candidate after the filtering stage. Like the `FilteringStage` class, the behaviour of this class is the same in all recommenders: iterating over the set of LOs and assigning a quality value to each one. As in the previous case, we follow the `Strategy` pattern: this class delegates quality assessment to an object that extends the `QualityMetric` class. This class will be detailed below.
- The `SelectionStage` class is responsible for building the final set of LOs recommended to the student.
A framework for the rapid prototyping of knowledge-based recommender systems in the learning domain

Fig. 3.a. Class hierarchy of the *Query elicitation* and the *Retrieval* stages.

Fig. 3.b. Class hierarchy of the *Filtering* and the *Rating* stages.

Fig. 3.c. Class hierarchy of the *Selection* stage.

Fig. 3. UML design of the proposed architecture (framework hooks appear in *italics*): class hierarchies corresponding to each stage in the recommendation process.
The **QualityMetric** class was designed while keeping in mind that the assessment of a usefulness LO should be a flexible algorithm. LO quality can be measured with different utility metrics and it is also necessary to combine several metrics in a flexible way to compose LO quality. For this reason, we adopted the **Composite** design pattern (Gamma et al. 1995) to implement this class. This pattern establishes that a **QualityMetric** can be an individual utility metric (represented by the **Utility** class), or a composition of different quality metrics (represented by the **AggregationFunction** class). The latter was improved by including weights for each component that takes part in the aggregation function.

Although the quality value of an LO often takes a normalized value in the interval \([0, 1]\) we decided to use a transfer object (**Transfer Object** or **Value Object** (Alur et al. 2003)) called **QualityTO** as the result of the assessment stage. This transfer object is generated by the **QualityMetric** class and it can contain, in addition to the quality value, all those attributes and values that may be relevant to the next stage and, therefore, to the final recommendation. The same solution was used in the **SelectionStage** class, which generates **RecommendationTO** transfer objects to store the LOs recommended and any additional data associated with each LO that will provide information about the recommendations. In this way, we combine two different approaches in the assessment process: the common numerical approach and symbolic criteria. The latter can be useful, for example, when providing justifications or explanations about the behaviour of the recommender, which could increase user confidence and trust in the recommendation process. Finally, we also employed the transfer object **QueryTO** in the **QueryElicitationStage** class to store the query.

### 3.2 The framework hooks

The classes described above have several hooks that let us modify the behaviour of the recommender. These hooks are instantiated by the subclasses that implement concrete strategies in order to extend the framework or use it to build a custom recommender.

The classes that implement the recommendation process stages have been developed using a two-phase initialization pattern. This pattern guarantees that the objects can be efficiently used in consecutive executions of the recommender. This pattern requires that any subclass that extends a class by modeling a stage of the process must implement the following hook methods: `boolean init()` and `void end()`. The former will be used to validate and initialize the stage, either by using the configuration parameters obtained from the blackboard, or by using the configuration parameters provided in the object creation. The latter will mainly be used to release resources and leave it ready for a new recommendation process. This two-phase initialization pattern is also used in other framework classes.

Some classes that represent the recommendation stages have other hook methods:

- The **QueryElicitationStage** class defines the **QueryTO generateQuery()** hook method. It is responsible for building the query employed in the recommendation process.
A framework for the rapid prototyping of knowledge-based recommender systems in the learning domain

- The RetrievalStage class defines the Collection<LO> retrieve(QueryTO) hook method. This method returns a set of LOs using the query generated by the previous class.
- The SelectionStage class defines the List<LO, RecommendationTO> select(Collection<LO, QualityTO>) hook method. This method generates the resulting list of recommended LOs. This method builds the RecommendationTO objects for each LO, by using a list of LOs and their respective QualityTOs objects, generated by the RatingStage class.

The FilteringStage class has a Collection<LO> filter(Collection<LO>) method to filter the LOs retrieved. However, this method does not represent a hook method. As mentioned before, the flexibility of this stage is provided by the delegation of the filtering strategy to the Filter class. This class does have a hook, the boolean filter(LO) method, responsible for deciding when an LO is removed from the list of candidates passed to the assessment stage. The Filter class also has the methods init and end, which are required by the two-phase initialization pattern.

Like the previous class, the RatingStage class has a Collection<LO, QualityTO> rate(Collection<LO>) method, which is responsible for assessing the LOs, but it does not represent a hook. This method is responsible for iterating over the previously retrieved and filtered LOs and it delegates the computation of the quality value of an LO to the QualityMetric class. Then it composes the set formed by pairs of LOs and their corresponding QualityTOs that will be transferred to the next stage of the recommendation process. The QualityMetric class has the method QualityTO computeQuality(LO) as a hook. Both the individual utility metrics (Utility) and aggregation functions (AggregationFunction) define their behaviour by implementing this method, which is responsible for calculating the quality of the LO provided as parameter and generating the associated QualityTO object. As in the Filter class, the use of the two-phase initialization pattern imposes two additional hook methods on the QualityMetric class: init and end.

The CBRecommender class has a set of hooks that need to be implemented by any subclass in order to generate a recommender, namely:

- createQueryElicitationStage, createRetrievalStage, createSelectionStage: These are the Factory Methods (Gamma et al. 1995) responsible for creating the concrete instances at each stage of the recommendation process. It is worth noting that there is no factory method for the assessing (RatingStage) and filtering stages (FilteringStage) because, as we detailed above, other classes are responsible for providing flexibility to these stages.
- createFilter: This factory method is responsible for creating the object in the Filter class used by the FilteringStage class to discard useless LOs.
- createQualityMetric: This factory method is responsible for creating the QualityMetric used by the RatingStage class to assess the LOs. This method creates the objects that make up the QualityMetric. When using a
quality metric as a combination of multiple utilities, these are composed by using a subclass of AggregationFunction. The individual utilities are included in the aggregation by using the add(QualityMetric, double) method, which supports the use of weights for each individual metric in the global quality metric.

- configureRecommendation(): It sets the parameters required for a recommendation, and keeps them on the blackboard. A user identifier is the only mandatory parameter needed for a recommendation, and can be used to access the user profile in those stages that need it. However, this method could be implemented so that it adds to the blackboard all those additional parameters that are considered necessary for the specific recommender.

- finishRecommendation(List<LO, RecommendationTO>): This method is in charge of processing the result of the recommendation. This may include, among other issues, displaying the recommendation, the data storage for further evaluations of the recommender or the update of the user profile according to the recommendation provided.

In order to coordinate the execution of the hooks described along this section, the CBRecommender class implements the following Template Methods (Gamma et al. 1995):

- init(): This method is responsible for initializing the recommender. It creates the instances that implement the recommender stages, the filtering strategies and the quality metrics by using the factory methods described above.

- initRecommendation(): This method is responsible for preparing the execution of a recommendation. First, it invokes the method configureRecommendation(). Then, it initializes the QueryElicitationStage and generates the query by invoking the generateQuery() method in this class, storing the query on the blackboard. Finally, it initializes the recommendation stages by invoking the respective init method of each class.

- recommend(): This method is responsible for generating the recommendation. It executes each of the recommender’s stages in an orderly way, invoking the hook methods in each class that serves as a recommender stage, as shown in Figure 4. The final result is an ordered list of LOs along with the additional information associated to each of them (encapsulated in a RecommendationTO object).

- endRecommendation(List<LO, RecommendationTO>): This method is responsible for the final processing of the recommendation and the release of the recommender stages. The former is implemented by calling the finishRecommendation(List <LO, RecommendationTO >) method, while the latter is implemented by sending the end message to each object representing a recommender's stage.
A framework for the rapid prototyping of knowledge-based recommender systems in the learning domain 11

Fig. 4. Passing messages produced by executing the method recommend.

3.3 The implementation of some concrete classes

In order to tend to a black box model, we have included the implementation of some subclasses of the abstract classes. This way, framework users can build, in an easy way, basic recommender systems that rely on the existence of a domain ontology to index the LOs and represent the student profile. Next subsections detail these classes.

3.3.1 Implementations for the Query elicitation, Retrieval and Selection stages

For the Query elicitation stage the framework includes the implementation of the UserQueryElicitationStage class, which inherits from the QueryElicitationStage class. The method generateQuery of this subclass uses a graphical interface to interact with the user and to request the query. This interface shows the domain concepts available in the query ontology and the user selects the concepts that make up the query.

For the Retrieval stage we have included the implementation of two different strategies. On the one hand, we have implemented an accurate retrieval strategy – AccurateRetrievalStage – that is responsible for retrieving those LOs indexed, at least, for all the query concepts. On the other hand, we have implemented an approximate retrieval strategy – ApproximateRetrievalStage – that selects those LOs indexed with at least one query concept or by siblings of one or several query concepts according to the concept organization within the ontology. Both classes inherit from the RetrievalStage class.

For the Selection stage we have developed a top $k$ selection strategy – the TopKSelectionStage class. This class sorts the set of candidates in terms of the quality assigned to each LO, and then selects the $k$ best LOs. The parameter $k$ is provided by the developer of the recommender and it may be defined when configuring it.
3.3.2 Implementation of a filter for the Filtering stage
The filter implemented relies on the existence of a learning path in the ontology that establishes the order in which the concepts should be learned in a formal learning context. This learning path and the competence level attained by a given student in each concept allows the identification of the concepts attained –concepts already explored by the student–, the concepts that the student can learn –concepts ready to be discovered– and the concepts that the student is not yet ready to learn –unreachable concepts. Using this classification, the filter discards from the retrieved set those LOs that cover unreachable concepts for the target student. The ReachableConceptsFilter class inherits from the Filter class and it implements this strategy through the filter method, which checks whether an LO covers any unreachable concept.

3.3.3 Implementation of quality metrics
Finally, we have implemented several individual quality metrics and aggregation functions that allow us to combine individual quality metrics in order to provide complex quality metrics.

The first quality metric implemented assesses the utility of each LO based on the similarity between the concepts that the LO covers and the query concepts. We decided to employ the quality metric that we had previously defined in (González-Calero et al. 1999). This metric computes the similarity between the set of the query concepts and the set of concepts associated with an LO by using the hierarchical structure defined in the ontology. This metric has been implemented in the QuerySimilarity class that inherits from the Utility class.

Another individual quality metric measures the pedagogical utility (PU) that an LO \( L \) has for a student \( S \). In order to compute pedagogical utility, we have employed an instructional strategy that promotes filling the student’s gaps in knowledge by including remedial knowledge. The goal is to assign high \( PU \) values to an LO if the student has a poor knowledge of most of the concepts covered by this LO. This way, the LO will help students enhance their knowledge of these concepts and then to achieve their long-term learning goals. This metric has been implemented in the PedagogicalUtility class, which inherits from the Utility class.

Regarding the aggregation functions, we have included the WeightedMeanMetric and HarmonicWeightedMeanMetric classes, which extend the AggregationFunction class within the framework. The former computes the utility as the weighted mean of all the utilities that make up the quality function. The latter combines the utilities by using the harmonic weighted mean metric. Both aggregation functions force the recommender developer to provide the weight of each individual utility in the resulting quality metric.
4 Using the framework to develop knowledge-based recommenders

Next we illustrate how to use our framework to develop two different knowledge-based recommenders. We assume that the knowledge base needed has been previously developed.

The use of our framework necessitates the creation of a subclass of the CBRecommender class and the implementation of its hook methods (see Section 3.2). To complete the implementation there are two alternatives:

- Using the classes described in Section 3.3 to instantiate the recommender stages, the quality metric and the filter. This alternative is exemplified in Section 4.1.
- Implementing new subclasses in order to extend framework functionality. Later, these new classes will be employed as in the previous alternative to build up the recommender. This alternative is exemplified in Section 4.2.

4.1 Case 1: A knowledge-based recommender that combines personalization and pedagogical utility

In this case we describe how to implement a recommender that takes into account the long-term learning goals of a student, in combination with her short-term learning goals, represented by a query. The work in (Ruiz-Iniesta et al. 2009) describes an initial approach to this recommender.

This prototype follows a reactive approach. The recommendation process starts with an explicit query posed by the student, which is generated by using the UserQueryElicitationStage class provided by the framework.

The retrieval stage was instantiated by using the ApproximateRetrievalStage class and we included a filtering stage that uses the ReachableConceptsFilter class explained in Section 3.3.

The quality metric employed prioritizes the LOs that cover concepts similar to the ones in the student query and, at the same time, that show significant pedagogical utility for the student. This quality metric combines the pedagogical utility metric ($PU$) and similarity with the query ($Sim$) as in Equation (1)

$$Quality(L, S, Q) = \alpha \cdot Sim(L, Q) + (1 - \alpha) \cdot PU(L, S) \quad \alpha \in [0, 1]$$

where $L$ represents the LO whose quality is being calculated, $Q$ represents the query and $S$ represents the student profile.

The quality metric is instantiated by using the aggregation function implemented by the WeightedMeanMetric class, where the similarity and the pedagogical utility weights are distributed as shown in equation (1). Similarity is computed by the QuerySimilarity class, while pedagogical utility is calculated by the PedagogicalUtility class. The resulting quality metric is built in the createQualityMetric factory method.
Finally, the top $k$ items are recommended by using an instance of the `TopKSelectionStage` class.

Once we have selected the classes for the stages, the filter and the quality metric, we have to implement the main class that models the recommender. This class, called `CBPersonalizationRecommender` in this example, is a subclass of the `CBRecommender` class. During each recommendation stage, the filter and the quality metric are instantiated by using the factory methods described in Section 3.2. Additionally, it is necessary to identify the user that requests the recommendation by using the `configureRecommendation` method, and to display the final recommendation using the `finishRecommendation` method. Figure 5 sketches the `CBPersonalizationRecommender` class using Java syntax.

```java
class CBPersonalizationRecommender extends CBRecommender {
    boolean configureRecommendation() {
        /* Sets the alpha parameter used by the quality metric
           Sets the user id that requests the recommendation */
    }
    QueryElicitacionStage createQueryElicitationStage() {
        return new UserQueryElicitationStage();
    }
    RetrievalStage createRetrievalStage() {
        return new ApproximateRetrievalStage();
    }
    Filter createFilter() {
        return new ReachableConceptsFilter();
    }
    QualityTO createQualityMetric() {
        Utility pu = new PedagogicalUtility();
        Utility sim = new QuerySimilarity();
        QualityMetric weightedMean = new WeightedMeanMetric();
        weightedMean.add(pu, 1-alpha);
        weightedMean.add(sim, alpha);
        return weightedMean;
    }
    SelectionStage createSelectionStage() {
        // in this example we select the top 10
        return new TopKSelectionStage(10);
    }
    void finishRecommendation(List<LO, RecommendationTO>) {
        // Displays the proposed recommendation to the user
    }
}
```

Fig. 5. Main class of a recommender that combines query similarity and pedagogical utility.

### 4.2 Case 2: A knowledge-based recommender that promotes diversity in the recommendation

In this example we implement a reactive recommender prototype that alleviates the overspecialization problem by including a new selection stage that promotes diversity. The query elicitation stage, the retrieval stage and the filtering stage are instantiated
A framework for the rapid prototyping of knowledge-based recommender systems in the learning domain

by using framework classes as in Case 1. The quality metric only computes the similarity with the query by using the QuerySimilarity class, also provided with the framework.

The selection stage is the novelty of this prototype. This new selection strategy, called diversity-conscious selection, is inspired by one of the diversity-conscious recommendation strategies described in (Ruiz-Iniesta et al. 2011). The selection strategy is implemented so that priority is given to those LOs that are most similar to the query and, at the same time, dissimilar to the rest of LOs already selected to be recommended. This strategy employs the list of LO candidates obtained in the previous step and incrementally builds the recommendation list of $k$ LOs that will ultimately be proposed. During each step, the remaining LOs in the list of candidates are sorted according to a metric that combines the quality assigned to each LO in the previous stage and the diversity from each LO in the current recommendation list. Later, the first LO in the list of candidates is transferred from this list to the current recommendation list. This process is repeated until the recommendation list contains $k$ LOs.

This new selection strategy is not available in the framework. Therefore, we have to implement it. We have implemented the RelDiversitySelection class which inherits from the SelectionStage class. It overrides the hook methods init, end and select. The init method verifies that the stage has been created with a valid $k$. The select method implements the algorithm proposed above. The end method releases the resources used by the stage and prepares the stage for the next recommendation. Finally, we have to create the class that inherits from CBRecommender and builds the recommender by using the selected stages and metrics. Figure 6 details the code that changes from the previous case.

```java
class CB多样性Recommender extends CBRecommender {
    ...
    boolean configureRecommendation() {
        // Identify the user that will request the recommendation
    }
    SelectionStage createSelectionStage() {
        /* in this example we select the top 10 LOs using the diversity-based selection strategy */
        return new RelDiversitySelection(10);
    }
    QualityMetric createQualityMetric() {
        return new QuerySimilarity();
    }
    ...
}
```
5 Conclusions and future work

In this paper, we have presented a framework that allows us to build prototypes of knowledge-based recommender systems of LOs in an easy way. We have identified five stages in the recommendation process, each one considered to be an aspect of variability. The framework was designed so that the developer can easily create recommenders that use alternative strategies, implemented for each stage. It was also considered that the framework can easily be increased by implementing new strategies. Finally, the paper exemplifies the use of the framework for developing two recommenders. The first development uses the classes that the framework provides, while the second one needs to extend the framework by easily including new implementations of some stages.

As far as immediate future work is concerned, we plan to increase the concrete classes of the framework. For instance, we will include classes corresponding to existing diversity algorithms other than the one used in section 4.2. We are also exploring how to extend the framework architecture in order to support hybrid recommendation strategies. In the long term, we are also considering the inclusion of classes that permit the automatization of the framework evaluation tasks.

6 Acknowledgments

This work has been supported by the Spanish Committee of Education and Science project TIN2009-13692-C03-03, IPT-2011-1890-430000 and UCM-BSCH group 921330-1079.

7 References

A framework for the rapid prototyping of knowledge-based recommender systems in the learning domain


