This work is supported by the MID-CBR project of the Spanish Ministry of Education & Science TIN2006-15140-C03-02 and the G.D. of Universities and Research of the Community of Madrid (UCM-CAM-910494 research group grant).

jCOLIBRI2 Tutorial
Juan A. Recio-García
Belén Díaz-Agudo
Pedro González Calero
Technical Report IT/2007/02
Department of Software Engineering and Artificial Intelligence.
University Complutense of Madrid
ISBN: 978-84-691-6204-0
Contents

1 Introduction 8

2 What is Case-Based Reasoning? 9
  2.1 The CBR cycle ................................................. 9

3 The jCOLIBRI CBR framework 11
  3.1 jCOLIBRI2 Architecture ................................. 11

4 Getting started with jCOLIBRI2 14

5 The Travel Recommender CBR application 17

6 Importing jCOLIBRI2 into Eclipse 24
  6.1 Preparing the Travel Recommender files ...................... 25

7 Creating the CBR application 28

8 The Travel Recommender Case Base 33

9 Representing the Cases 35
  9.1 Case Components ........................................... 35
  9.2 Cases and Queries in jCOLIBRI2 .......................... 36
  9.3 Case Components of the Travel Recommender ............... 39
  9.4 Defining new attribute types ............................... 41

10 The two layers persistence architecture of jCOLIBRI2 42
  10.1 The Connectors of jCOLIBRI2 ............................ 42
  10.2 In-memory organization of the cases ....................... 44

11 Loading the Case Base 46
  11.1 Configuring the connector ................................... 47
    11.1.1 The Hibernate configuration file ....................... 48
    11.1.2 Creating the mapping files .......................... 50

12 Using Ontologies in CBR applications 54
  12.1 Case Base persistence in ontologies ...................... 55
  12.2 Computing similarities using ontologies .................. 55
  12.3 Case and Query vocabulary ................................ 56
  12.4 Using an ontology in the Travel Recommender ............. 58

13 Retrieval 61
  13.1 Similarity functions ...................................... 62
  13.2 Cases selection ........................................... 63
  13.3 Using the k-NN retrieval ................................... 64
  13.4 Retrieval in the Travel Recommender application .......... 66
Contents

14 Reuse 67
  14.1 Adapting the Travel Recommender retrieved trips 67

15 Revise 69
  15.1 Travel Recommender revision 69

16 Retain 70
  16.1 Saving the new trips of the Travel Recommender 70

17 Shutting down a CBR application 71

18 Textual CBR 72
  18.1 Semantic retrieval 72
  18.1.1 Representation of the texts 73
  18.1.2 IE methods implementation 75
  18.1.3 Computing similarity 76
  18.1.4 The Restaurant Recommender example 77
  18.2 Statistical retrieval 81
  18.2.1 The Restaurant Recommender using statistical methods 82

19 Evaluation of CBR applications 83

20 Recommenders 86
  20.1 Templates guided design of recommendation systems 86
    20.1.1 One-Off Preference Elicitation 87
    20.1.2 Retrieval 89
    20.1.3 Iterated Preference Elicitation 89
  20.2 Methods for recommender systems 90
    20.2.1 User interaction methods 90
    20.2.2 New Nearest Neighbor similarity measures 91
    20.2.3 Conditional methods 92
    20.2.4 Navigation by Asking methods 92
    20.2.5 Navigation by Proposing 93
    20.2.6 Profile management methods 94
    20.2.7 Collaborative Recommendations 94
    20.2.8 Retrieval methods 95
    20.2.9 Cases selection 97

21 Other Features 99
  21.1 Visualization of a Case Base 99
  21.2 Classification and Maintenance 99

22 Getting support 101

23 Contributing to jCOLIBRI 102
  23.1 Required elements 102
  23.2 Example applications 102
23.3 How to submit a contribution ........................................ 103

24 Versions ChangeLog .................................................. 104
   24.1 Version 2.1 ....................................................... 104
      24.1.1 What’s new in Version 2.1 ............................... 104
   24.2 Version 2.0 ....................................................... 105
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CBR cycle by Aamodt and Plaza</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Two layers architecture of jCOLIBRI2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>jCOLIBRI2 short-cuts under windows</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>jCOLIBRI2 Tester</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>jCOLIBRI2 Tests</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>Define Query step</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Configure Similarity step</td>
<td>19</td>
</tr>
<tr>
<td>8</td>
<td>Retrieved Cases step</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>Adaptation step</td>
<td>21</td>
</tr>
<tr>
<td>10</td>
<td>Revise Cases step</td>
<td>22</td>
</tr>
<tr>
<td>11</td>
<td>Retain Cases step</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>Importing the jCOLIBRI2 project into Eclipse</td>
<td>24</td>
</tr>
<tr>
<td>13</td>
<td>The jCOLIBRI2 project in Eclipse</td>
<td>25</td>
</tr>
<tr>
<td>14</td>
<td>Files to delete in the project</td>
<td>26</td>
</tr>
<tr>
<td>15</td>
<td>Project state to begin the tutorial</td>
<td>27</td>
</tr>
<tr>
<td>16</td>
<td>Cases representation UML diagram</td>
<td>39</td>
</tr>
<tr>
<td>17</td>
<td>Case Base management in jCOLIBRI2</td>
<td>42</td>
</tr>
<tr>
<td>18</td>
<td>Travel Recommender components mapping</td>
<td>50</td>
</tr>
<tr>
<td>19</td>
<td>Case mapping in an ontology</td>
<td>55</td>
</tr>
<tr>
<td>20</td>
<td>OntologyConnector behavior example</td>
<td>56</td>
</tr>
<tr>
<td>21</td>
<td>Concept based similarity functions in jCOLIBRI2</td>
<td>57</td>
</tr>
<tr>
<td>22</td>
<td>Example of application of the similarity functions</td>
<td>58</td>
</tr>
<tr>
<td>23</td>
<td>Defining the Travel Recommender query through an ontology</td>
<td>59</td>
</tr>
<tr>
<td>24</td>
<td>Mapping between data base and ontology</td>
<td>60</td>
</tr>
<tr>
<td>25</td>
<td>Representation of texts for IE.</td>
<td>74</td>
</tr>
<tr>
<td>26</td>
<td>Global view of the representation of texts for IE.</td>
<td>75</td>
</tr>
<tr>
<td>27</td>
<td>Common organization of textual cases</td>
<td>76</td>
</tr>
<tr>
<td>28</td>
<td>Evaluation of a CBR application</td>
<td>84</td>
</tr>
<tr>
<td>29</td>
<td>Single Shot template</td>
<td>87</td>
</tr>
<tr>
<td>30</td>
<td>Conversational templates</td>
<td>88</td>
</tr>
<tr>
<td>31</td>
<td>Preference Elicitation decompositions</td>
<td>88</td>
</tr>
<tr>
<td>32</td>
<td>Visualization of a Case Base</td>
<td>100</td>
</tr>
</tbody>
</table>
## Listings

<table>
<thead>
<tr>
<th>Listings</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>StandardCBRApplication interface</td>
<td>28</td>
</tr>
<tr>
<td>TravelRecommender initial code</td>
<td>29</td>
</tr>
<tr>
<td>TravelRecommender singleton</td>
<td>29</td>
</tr>
<tr>
<td>TravelRecommender GUI code</td>
<td>30</td>
</tr>
<tr>
<td>TravelRecommender main() method</td>
<td>31</td>
</tr>
<tr>
<td>TravelRecommender data base schema</td>
<td>33</td>
</tr>
<tr>
<td>TravelRecommender configure() method (version 1)</td>
<td>33</td>
</tr>
<tr>
<td>CaseComponent interface</td>
<td>35</td>
</tr>
<tr>
<td>Bean example code</td>
<td>35</td>
</tr>
<tr>
<td>Using Attribute example code</td>
<td>36</td>
</tr>
<tr>
<td>CBRQuery code</td>
<td>36</td>
</tr>
<tr>
<td>CBRCase code</td>
<td>37</td>
</tr>
<tr>
<td>TravelDescription initial code</td>
<td>39</td>
</tr>
<tr>
<td>TravelSolution initial code</td>
<td>40</td>
</tr>
<tr>
<td>TypeAdaptor interface</td>
<td>41</td>
</tr>
<tr>
<td>Connector interface</td>
<td>43</td>
</tr>
<tr>
<td>CBRCaseBase interface</td>
<td>44</td>
</tr>
<tr>
<td>TravelRecommender configure() method (version 2) and precycle()</td>
<td>46</td>
</tr>
<tr>
<td>databaseconfig.xml</td>
<td>47</td>
</tr>
<tr>
<td>hibernate.cfg.xml</td>
<td>48</td>
</tr>
<tr>
<td>TravelSolution.hbm.xml</td>
<td>50</td>
</tr>
<tr>
<td>TravelDescription.hbm.xml</td>
<td>51</td>
</tr>
<tr>
<td>Mapping template for user-defined types</td>
<td>52</td>
</tr>
<tr>
<td>Mapping template for enumerates</td>
<td>52</td>
</tr>
<tr>
<td>TravelRecommender configure() method (final version)</td>
<td>58</td>
</tr>
<tr>
<td>NNScoringMethod signature</td>
<td>61</td>
</tr>
<tr>
<td>Local and Global similarity interfaces</td>
<td>62</td>
</tr>
<tr>
<td>Cases selection methods</td>
<td>63</td>
</tr>
<tr>
<td>Example code for the k-NN Retrieval</td>
<td>64</td>
</tr>
<tr>
<td>TravelRecommender.cycle() code (step1)</td>
<td>66</td>
</tr>
<tr>
<td>TravelRecommender.cycle() code (step2)</td>
<td>68</td>
</tr>
<tr>
<td>TravelRecommender.cycle() code (step3)</td>
<td>69</td>
</tr>
<tr>
<td>TravelRecommender.cycle() code (step4)</td>
<td>70</td>
</tr>
<tr>
<td>TravelRecommender.postCycle() code</td>
<td>71</td>
</tr>
<tr>
<td>The Restaurant Recommender precycle using semantic TCBR methods</td>
<td>78</td>
</tr>
<tr>
<td>The Restaurant Recommender cycle using semantic TCBR methods</td>
<td>79</td>
</tr>
<tr>
<td>The Restaurant Recommender using statistical TCBR methods</td>
<td>82</td>
</tr>
<tr>
<td>Evaluation code</td>
<td>84</td>
</tr>
<tr>
<td>Example of an evaluable application</td>
<td>85</td>
</tr>
<tr>
<td>Visualization code</td>
<td>99</td>
</tr>
<tr>
<td>Config file for the Examples application</td>
<td>103</td>
</tr>
</tbody>
</table>
1 Introduction

Case-based reasoning (CBR) has become a mature and established subfield of Artificial Intelligence (AI), both as a mean for addressing AI problems and as a basis for fielded AI technology.

Now that CBR fundamental principles have been established and numerous applications have demonstrated CBR is an useful technology, many researchers agree about the increasing necessity to formalise this kind of reasoning, define application analysis methodologies, and provide a design and implementation assistance with software engineering tools [5, 4, 27, 16, 20]. While the underlying ideas of CBR can be applied consistently across application domains, the specific implementation of the CBR methods—in particular retrieval and similarity functions—is highly customised to the application at hand. Two factors have became critical: the availability of tools to build CBR systems, and the accumulated practical experience of applying CBR techniques to real-world problems.

Our work goes along all these increasing necessities. We have designed a tool to help application designers to develop and quickly prototyping CBR systems. Besides we want to provide a software tool useful for students who have little experience with the development of different types of CBR systems. jCOLIBRI has been designed as a wide spectrum framework able to support several types of CBR systems from the simple nearest-neighbor approaches based on flat or simple structures to more complex Knowledge Intensive ones. It also supports the development of textual and conversational CBR applications [41, 21]. Other features of the framework like: ontology integration, visualization of case bases, evaluation of CBR applications, classification and maintenance methods, ... will be explained in this tutorial.

The framework implementation is evolving as new methods are included. Our (ambitious) goal is to provide a reference framework for CBR development that would grow with contributions from the community. We invite the jCOLIBRI developers to send us their methods to enrich the functionality of the framework and make them available to the whole CBR community.
2. What is Case-Based Reasoning?

Case-based reasoning is a paradigm for combining problem-solving and learning that has become one of the most successful applied subfield of AI of recent years. CBR is based on the intuition that problems tend to recur. It means that new problems are often similar to previously encountered problems and, therefore, that past solutions may be of use in the current situation [12]. CBR is rooted in the works of Roger Schank on dynamic memory and the central role that a reminding of earlier episodes (cases) and scripts (situation patterns) has in problem solving and learning [48].

CBR is particularly applicable to problems where earlier cases are available, even when the domain is not understood well enough for a deep domain model. Helpdesks, diagnosis or classification systems have been the most successful areas of application, e.g., to determine a fault or diagnostic an illness from observed attributes, or to determine whether or not a certain treatment or repair is necessary given a set of past solved cases [54].

Central tasks that all CBR methods have to deal with are [2]: "to identify the current problem situation, find a past case similar to the new one, use that case to suggest a solution to the current problem, evaluate the proposed solution, and update the system by learning from this experience. How this is done, what part of the process that is focused, what type of problems that drives the methods, etc. varies considerably, however".

For a detailed description on these and other CBR related aspects, we address the interested reader to detailed surveys about the CBR field ([2],[13],[32], [12]) and to the last European and international conferences in the field (ECCBR and ICCBR).

2.1 The CBR cycle

At the highest level of generality, a general CBR application can be described by a cycle composed of the following four processes[2]:

- RETRIEVE the most similar case or cases.
- REUSE the information and knowledge in that case to solve the problem.
- REVISE the proposed solution.
- RETAIN the parts of this experience likely to be useful for future problem solving.

A new problem is solved by retrieving one or more previously experienced cases, reusing the case in one way or another, revising the solution based on reusing a previous case, and retaining the new experience by incorporating it into the existing knowledge-base (case-base). In Figure 1, this cycle is illustrated.

An initial description of a problem (top of figure) defines the query. This new case is used to RETRIEVE a case from the collection of previous cases. The retrieved case is combined with the new case - through REUSE - into a solved case, i.e. a proposed solution to the initial problem. Through the REVISE process this solution is tested for
2.1 The CBR cycle

As indicated in the figure, general knowledge usually plays a part in this cycle, by supporting the CBR processes. This support may range from very weak (or none) to very strong, depending on the type of CBR method. By general knowledge we mean general domain-dependent knowledge, as opposed to specific knowledge embodied by cases. For example, in diagnosing a patient by retrieving and reusing the case of a previous patient, a model of anatomy together with causal relationships between pathological states may constitute the general knowledge used by a CBR system. A set of rules may have the same role.
3 The jCOLIBRI CBR framework

jCOLIBRI is an object-oriented framework in Java for building CBR systems that is an evolution of previous work on knowledge intensive CBR [16, 18]. The first version of our software (COLIBRI1) was prototyped in LISP and was far from being usable outside of our own research group. However we learnt good lessons from it, and we have designed jCOLIBRI as a technological evolution of COLIBRI that incorporates the advantages of the software technologies developed during the last few years.

The jCOLIBRI framework has two major releases:

jCOLIBRI version 1: jCOLIBRI version 1 is the first release of the framework. It includes a complete Graphical User Interface that guides the user in the design of a CBR system. This version is recommended for non-developer users that want to create CBR systems without programming any code.

jCOLIBRI version 2: jCOLIBRI version 2 is a new implementation that follows a new and clear architecture divided into two layers: one oriented to developers (finished) and other oriented to designers (future work). This new design is a complete white box framework (see below) open to java developers that want to include the features of jCOLIBRI in their CBR applications.

This tutorial shows how to develop a CBR application using the jCOLIBRI version 2 (jCOLIBR12 ahead). For a reference guide of the first version read [45].

3.1 jCOLIBRI2 Architecture

jCOLIBRI2 is the result of the experience acquired during the development of the first version. It solves many drawbacks like case representation, management of metadata, development problems, etc. But the architecture of this new version is very different (although compatible) as is based on a complete revision of CBR systems and frameworks.

There are many definitions of framework, but the one that is probably most referenced is found in [28]:

A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.

In other words, a framework is a partial design and implementation for an application in a given problem domain.

---

1Cases and Ontology Libraries Integration for Building Reasoning Infrastructures
For classifying a framework we can distinguish two types. A white-box framework is reused mostly by subclassing and a black-box framework is reused through parametrization. The usual development of a framework begins with a design as a white-box architecture that evolves into a black-box one. The resulting black-box framework has an associated builder that will generate the application’s code. The visual builder allows the software designer to connect the framework objects and activate them. The first version of jCOLIBRI is closer to a black-box framework with visual builder and lacks of a clear white-box structure.

The new design of jCOLIBRI attempts to remodel the architecture into a clear white-box system oriented to programmers, and a black-box with builder layer that is oriented to designers.

The key idea in the new design consists of separating core classes and user interface. That separation will give us the two layers architecture shown in the following figure:

![Two layers architecture of jCOLIBRI](image)

The bottom layer contains the basic components of the framework with well defined and clear interfaces. This layer does not contain any kind of graphical tool for developing CBR applications; it is simply a white-box object-oriented framework that must be used by programmers. The top layer contains semantic descriptions of the components and several tools that aid users in the development of CBR applications (black-box with visual builder framework).

The bottom layer has new features that solve most of the problems identified in the first version. It takes advantage of the new possibilities offered by the newest versions of the Java language. The most important change is the representation of cases as Java...
Beans. The persistence of cases is managed by the Hibernate package. Hibernate is a Java Data Objects (JDO) implementation, so it can automatically store Java Beans in a relational data base, using one or more tables. Java Beans and Hibernate are core technologies in the Java 2 Enterprise Edition platform that is oriented to business applications. Using these technologies in jCOLIBRI2 we guarantee the future development of commercial CBR applications using this framework. All these features will be explained in this tutorial.

Regarding the top layer, it is oriented to designers and includes several tools with graphical interfaces. This layer is the black-box version of the framework, helping users to develop complete CBR applications guiding the configuration process. The top layer is still under development, so this tutorial explains how to use the bottom layer of jCOLIBRI2.

A detailed explanation of the features and motivations of the jCOLIBRI2 architecture can be found in [44].
4 Getting started with jCOLIBRI2

The framework can be downloaded from the web page:

http://gaia.fdi.ucm.es/projects/jcolibri

Or directly from the sourceforge.net project page:

http://sourceforge.net/projects/jcolibri-cbr/

There are three main versions of the framework: v1.1, v2.0 and v2.1. This tutorial covers versions 2.0 and 2.1. These two versions are very similar. jCOLIBRI 2.1 extends jCOLIBRI 2.0 including the recommenders package explained in Section 20. There are also a few changes detailed in Section 24. In this tutorial we will refer to both versions as jCOLIBRI2.

Make sure that you download the jCOLIBRI 2.1 version (not the previous ones).

If you download the windows version, a wizard will guide you during the installation process. Or if you use any other operative system just unzip the file in any folder.

The jCOLIBRI2 release contains the following files:

- ./jCOLIBRI2-Tester.bat : The tester application for Windows.
- ./jCOLIBRI2-Tester.sh : The tester application for Unix.
- ./README.txt : This file.
- ./project : Eclipse project file that can be imported into that IDE.
- ./classpath : Eclipse classpath file with the libraries used in jCOLIBRI.
- ./build.xml : Apache Ant build file to compile the sources.
- ./jcolibri2.LICENSE : License agreement.
- ./src/jcolibri : The source code of the jCOLIBRI package.
- ./doc/api/index.html : The index to the javadoc of the jCOLIBRI package.
- ./doc/uml/ : UML class and sequence diagrams.
- ./doc/configfilesSchemas/ : XML schemas for the configuration files of the connectors used to stores persistent data
- ./lib : jCOLIBRI as well as third-party libraries included in this release.
- ./lib/jcolibri2.jar : Compiled jCOLIBRI jar file.
- ./TravelRecommender : Folder for the Travel Recommender example.
- ./TravelRecommender.bat : The Travel Recommender for Windows.
- ./TravelRecommender.sh : The Travel Recommender for Unix.
jCOLIBRI2 requires that Java (1.6 or later) is installed on your computer.

Once the framework is installed, you can launch the framework examples through the start menu (only windows O.S.).

![jCOLIBRI2 short-cuts under windows](image)

These short cuts link to the files located in the installation directory.

There is an application tester that can be launched using the following scripts:

- In Windows: jCOLIBRI2-Tester.bat
- In UNIX: jCOLIBRI2-Tester.sh

The tester let you run each one of the 30 test applications included in the release to exemplify the main features of jCOLIBRI. In the Tester you can access to the javadocs of the tests which are the main source of documentation as shown in Figure 4.

![jCOLIBRI2 Tester](image)

In the Code Examples section of the jCOLIBRI web page (http://gaia.fdi.ucm.es/projects/jcolibri/jcolibri2/examples.html), a table summarizes...
which features are explained by each test (see Figure 5). This table is also shown through the Map button of the Tester application.

The first 16 examples illustrate the behavior of general CBR application meanwhile the following 14 show how to implement recommender systems. The reading of these tests is recommended while studying this tutorial as many sections cite them for details.

There is also an example of using the jCOLIBRI2 jar library in a stand-alone application named "Travel Recommender". It can be launched using the following scripts:

- In Windows: TravelRecommender.bat
- In UNIX: TravelRecommender.sh

This tutorial will show you how to develop the Travel Recommender application following some simple steps.
5 The Travel Recommender CBR application

The Travel Recommender is a well known application example in the CBR community. Therefore, in this tutorial we have chosen this application to show how to implement CBR applications using jCOLIBRI2. Following sections guide the readers in the development process of this application in an incremental way. Before starting with the implementation, this section describes the features and behavior of the Travel Recommender application.

In this application, the case base is composed of several trips. The system receives a desired trip as a query, compares the query with the trips in the case base using a similarity function, and returns the most similar ones. After the retrieval, these most similar cases can be adapted according with the restrictions of the query.

Each case is represented by several attributes: \textit{Id, holiday type, number of persons, region, transportation, duration, season, accommodation type, price and hotel}. In our application, the price and the hotel will be considered as the solution of the case meanwhile the remaining attributes will be the case description.

This way, our cases are divided into: a description used to retrieve similar cases given a query, and a solution that is adapted depending on the values of the query.

Lets look at the different steps of our CBR application. You can execute it through the short-cuts in the windows installation (see Figure 3 or the TravelRecommender.bat and TravelRecommender.sh scripts located in the installation folder.)
Define Query. In this step the user defines her query to the system. She has to define which are the values of the different attributes of a trip: duration, season, transportation, etc.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holiday Type</td>
<td>Skiing</td>
</tr>
<tr>
<td>Number of persons</td>
<td>2</td>
</tr>
<tr>
<td>Region</td>
<td>Spain</td>
</tr>
<tr>
<td>Transportation</td>
<td>Plane</td>
</tr>
<tr>
<td>Duration</td>
<td>?</td>
</tr>
<tr>
<td>Season</td>
<td>July</td>
</tr>
<tr>
<td>Accommodation</td>
<td>ThreeStars</td>
</tr>
</tbody>
</table>

Figure 6: Define Query step
Configure Similarity. Here the user configures the similarity measure used to retrieve the cases most similar to the query. jCOLIBR12 implements several similarity functions that can be used depending on the type of the attribute (integers, strings, etc.). Moreover, developers can define their own similarity measures.

You can also assign a weight to each attribute of the query that will be taken into account when computing the average of all attributes. Also, some similarity functions can have parameters used to configure the similarity measure.

Finally, the $k$ value indicates how many cases must be retrieved. We are using an algorithm named K Nearest Neighbor (k-NN) that computes the similarity of the query with all the cases and then orders the result depending on that similarity value. Then the first $k$ most similar cases are returned.

Figure 7: Configure Similarity step
**Retrieved Cases.** This step shows the documents retrieved by the k-NN method. It shows each case and its similarity to the query.

![Retrieved Cases step](image)

**Figure 8:** Retrieved Cases step
Adaptation. In this step the system adapts the retrieved cases to the requirements of the user depending on the values defined in the query. This stage use to be very domain dependent and will be different in other CBR systems. In our Travel Recommender application we will adapt the price of the trips depending on the number of persons and duration defined in the query.

Our system will use simple direct proportions to perform the adaptation. For example, imagine that the retrieved case has a duration of 7 days and costs 1000. If the user is looking for a 14 days trip we have to adapt the price using a direct proportion: if a 7 days trip costs 1000, then a 14 days trip costs 2000. This process is also repeated for the number of persons in an analogous way.

After adapting the solution of the cases, their description can be substituted by the description of the query. At this point, the system will manage a list of working cases that are different from the cases in the case base. These working cases represent possible solutions to the problem described in the query.

![Figure 9: Adaptation step](image)
Revise Cases. Once cases have been adapted, the user (or a domain expert, in this case the trip agent) would adjust the values of the working cases in a manual way. For example, imagine that the hotel has not available rooms and clients have to go to a similar one. Another situation is that retrieved cases are not similar enough to the requirements of the query and the trip agent has to define manually the solution of the case.

![Figure 10: Revise Cases step](image.png)
Retain Cases. Finally, the trip agent would save the new trip case into the case base for being used in future queries. If it is done, a new Id must be assigned to the trip.

![Figure 11: Retain Cases step](image-url)
6 Importing jCOLIBRI2 into Eclipse

To develop a new CBR application with jCOLIBRI2 we recommend the Eclipse IDE (www.eclipse.org). jCOLIBRI2 includes the required files to import the framework project into Eclipse: .classpath and .project. But if you import the whole project, you will have the source files of the framework into your Eclipse project. The smart solution consists on loading only the jcolibri2.jar file (and related libraries) into your project. That way, you will have only your source files in the Eclipse project.

For teaching purposes, in this tutorial we will load the whole jCOLIBRI project into Eclipse to allow an easier navigation through the source code of the framework. To avoid modifications of the source files of your installation, we will make a copy of the project into the Eclipse workspace when importing the project.

To import jCOLIBRI2 into Eclipse use the File - Import menu. Then choose the “General - Existing projects into workspace” option and finally select the installation folder of the framework into the “Select root directory” field. Make sure that the “Copy projects into workspace” option is checked and click on “Finish”.

![Figure 12: Importing the jCOLIBRI2 project into Eclipse](Image)
6.1 Preparing the Travel Recommender files

Once the project is imported, you can navigate through its contents and source files using the "Package Explorer".

![Package Explorer](image)

**Figure 13:** The jCOLIBRI2 project in Eclipse

### 6.1 Preparing the Travel Recommender files

The complete source code of the travel recommender application is located into the “example” subfolder of the framework and is compressed as `travelrecommender-source.zip`. In the following sections we will create the most important source files from scratch. However, we will not create the code related with the GUI of the application because it is out of the scope of this tutorial.

So, you must decompress the contents of the zip file into the “src” subfolder of your jCOLIBRI2 Eclipse project (not the original installation folder). This project will be located into your Eclipse workspace directory. Make sure that you preserve the folder names when unzipping the file.

Once done, come back to Eclipse, select the jCOLIBRI2 project into the “Package Explorer”, and refresh it using the contextual menu option or the F5 key. Then a new subpackage named `jcolibri.examples.TravelRecommender` will appear the source folder.

As explained before we only keep the GUI files. During the tutorial, we are doing the rest of the files step by step. So, select the following files and delete them: `TravelDescription.java`, `TravelRecommender.java`, `TravelSolution.java`, `databaseconfig.xml`, `hibernate.cfg.xml`, `TravelDescription.hbm.xml`, and `TravelSolution.hbm.xml`. (They are shown in Figure 14).
The contents of your project before beginning with the tutorial is also shown in Figure 15.
6.1 Preparing the Travel Recommender files

**Figure 15:** Project state to begin the tutorial
7 Creating the CBR application

A CBR application in jCOLIBRI must implement or extend the jcolibri.cbrapplications.StandardCBRApplication interface:

```
public interface StandardCBRApplication {
    /**
     * Configures the application: case base, connectors, etc.
     * @throws ExecutionException
     */
    public void configure() throws ExecutionException;

    /**
     * Runs the precycle where typically cases are read and
     * organized into a case base.
     * @return The created case base with the cases in the
     * storage.
     * @throws ExecutionException
     */
    public CBRCaseBase preCycle() throws ExecutionException;

    /**
     * Executes a CBR cycle with the given query.
     * @throws ExecutionException
     */
    public void cycle(CBRQuery query) throws ExecutionException;

    /**
     * Runs the code to shutdown the application. Typically
     * it closes the connector.
     * @throws ExecutionException
     */
    public void postCycle() throws ExecutionException;
}
```

This interface divides the CBR application behavior into 3 steps:

- **Precycle**: Initializes the CBR application, usually loading the case base and pre-computing expensive algorithms (really useful when working with texts). It is executed only once.
- **Cycle**: Executes the CBR cycle. It is executed many times.
- **Postcycle**: Post-execution or maintenance code.

Now, let's create the main class of our Travel Recommender. The name of the class must be `TravelRecommender.java` of the package...
jcolibri.examples.TravelRecommender. Include also the main() method in the class:

**Listing 2: TravelRecommender initial code**

```java
public class TravelRecommender implements StandardCBRApplication {
    public void configure() throws ExecutionException {
    }
    public void cycle(CBRQuery query) throws ExecutionException {
    }
    public void postCycle() throws ExecutionException {
    }
    public CBRCaseBase preCycle() throws ExecutionException {
    }
    public static void main(String[] args) {
    }
}
```

To ease the access to this class from the different GUI frames and to ensure that there is only one instance of this class we are going to implement a singleton pattern. Include the following code into the class:

**Listing 3: TravelRecommender singleton**

```java
private static TravelRecommender _instance = null;
public static TravelRecommender getInstance() {
    if (_instance == null)
        _instance = new TravelRecommender();
    return _instance;
}
private TravelRecommender() {
}
```

This way, the unique instance of the TravelRecommender class must be accessed through the getInstance() method.

As the application uses several frames we need a main frame that acts as the parent of all of them. We are going to create a simple main frame that only contains the jCOLIBRI2 logo. Then we will have a separated frame for each step of the application. Include the following code in the class:
This code shows an useful class of jCOLIBRI2 named jcolibri.util.FileIO. This class finds a file in the src, bin or main folders of the project even if it is packaged into a jar file. This is very useful if your project must access to external files.
Now let's create the main method of the Travel Recommender application. It config-
ures the application, executes the precycle (only once), executes the cycle several times
and finally calls the postcycle code:

**Listing 5: TravelRecommender main() method**

```java
public static void main(String[] args) {
    // Obtain TravelRecommender object
    TravelRecommender recommender = getInstance();
    // Show the main frame
    recommender.showMainFrame();

    try {
        // Configure the application
        recommender.configure();
        // Execute the Precycle
        recommender.preCycle();

        // Create the frame that obtains the query
        QueryDialog qf = new QueryDialog(main);

        // Main CBR cycle
        boolean cont = true;
        while (cont) {
            // Show the query frame
            qf.setVisible(true);
            // Obtain the query
            CBRQuery query = qf.getQuery();

            // Call the cycle
            recommender.cycle(query);

            // Ask if continue
            int ans = java.awt.JOptionPane.showConfirmDialog(null,
                                                            "CBR cycle finished, query again?", "Cycle finished",
                                                            java.awt.JOptionPane.YES_NO_OPTION);
            cont = (ans == java.awt.JOptionPane.YES_OPTION);
        }

        // Execute postcycle
        recommender.postCycle();
    } catch (Exception e) {
        // Errors
        org.apache.commons.logging.LogFactory.getLog(
            TravelRecommender.class).error(e);
        java.awt.JOptionPane.showMessageDialog(null, e.
                                               getMessage());
    }
}
```
System.exit(0);
}

This code shows how to access the unique instance of the `TravelRecommender` class through the `getInstance()` method. Then the main frame is shown. After that, the CBR application is configured and the precycle is executed. That code is executed only once.

Then we create the frame that obtains the query: `QueryDialog`. As explained before, this class must receive the main frame in the constructor. This frame (shown in Figure 6) returns the query defined by the user.

`jCOLIBRI2` defines a class to store the queries named `jcolibri.cbrcore.CBRQuery`. It will be explained later in Section 9.2.

Finally, the code shows that `jCOLIBRI2` uses the Apache Log4j library to manage the logging messages.
8 The Travel Recommender Case Base

The case base of the application is stored in a Data Base. jCOLIBRI2 can manage any Data Base Manager (DBM) because it uses internally the Hibernate (www.hibernate.org) middleware. Hibernate is a powerful, high performance object/relational persistence and query service. This project and a critical component of broadly used JBoss J2EE server supporting persistence in Data Bases or XML files among many other features like its own object-oriented query language. Using Hibernate, jCOLIBRI2 allows the use of any DBM like Oracle, MySQL, etc. This way, you can reuse any data base to create your case base for a CBR application.

The Travel Recommender application uses the data base defined in the travel.sql file located in the jcolibri.example.TravelRecommender package. You can use that file to generate the data base in your favorite DBM.

This file generates a data base named travel that contains a table also named travel. The table contains 1024 trips and is defined with the following schema:

```sql
create table travel(
    caseId VARCHAR(15),
    HolidayType VARCHAR(20),
    Price INTEGER,
    NumberOfPersons INTEGER,
    Region VARCHAR(30),
    Transportation VARCHAR(30),
    Duration INTEGER,
    Season VARCHAR(30),
    Accommodation VARCHAR(30),
    Hotel VARCHAR(50));
```

To simplify the use of the examples, jCOLIBRI2 includes the HSQLDB Data Base Manager (www.hsqldb.org). This DBM is completely implemented in Java and can be easily included in any other Java project. This way, HSQLDB is used by the examples of the framework and by the Travel Recommender application.

To launch the DBM you can use the jcolibri.test.database.HSQLDBserver class. Its init() method initializes the DBM loading the data bases used by the examples. This initialization also includes the tables used by our Travel Recommender defined in travel.sql.

Now you have to add the following code into the TravelRecommender class to lauch and stop the DBM:

```java
public void configure() throws ExecutionException {
```

Group for Artificial Intelligence Applications jCOLIBRI2 Tutorial
try {
   // Emulate database server
   jcolibri.test.database.HSQLDBserver.init();

   // Create the dialogs
   ...
} catch (Exception e) {
   throw new ExecutionException(e);
}

public void postCycle() throws ExecutionException {
   _connector.close();
   jcolibri.test.database.HSQLDBserver.shutDown();
}

Note that this code is not the final code for those methods. They will be extended in following sections.
9 Representing the Cases

Now that we have the data base that contains the cases, we have to represent them in our system. This section details how cases are represented inside a jCOLIBRI2 CBR application.

jCOLIBRI2 represents the cases using Java Beans. A Java Bean is any class that has a get() and set() method for each public attribute. Its modification and management can be performed automatically using a Java technology called Introspection (that is completely transparent for the developer). Using Java Beans in jCOLIBRI2, developers can design their cases as normal Java classes, choosing the most natural design. This simplifies programming and debugging the CBR applications, and the configuration files became simpler because most of the metadata of the cases can be extracted using Introspection. Java Beans also offer automatically generated user interfaces that allow the modification of their attributes and automatic persistence into data bases and XML files. It is important to note that every Java web application uses Java Beans as a base technology, so the development of web interfaces is very straightforward. Moreover, Hibernate uses Java Beans to store the information into a data base. Java Beans and Hibernate are core technologies in the Java 2 Enterprise Edition platform that is oriented to business applications. Using these technologies in jCOLIBRI2 we guarantee the future development of commercial CBR applications using this framework.

9.1 Case Components

The unique restriction of the Java Beans that compose a case is that they must define an Id that identifies them in the table of the DBM. This restriction is defined in the jcolibri.cbrcore.CaseComponent interface:

```java
public interface CaseComponent {
    /**
     * Returns the attribute that identifies the component.
     * An id-attributemust be unique for each component.
     */
    jcolibri.cbrcore.Attribute getIdAttribute();
}
```

The jcolibri.cbrcore.Attribute class represents an attribute of a Java Bean. For example, if you have a Java Bean like this:

```java
public class MyBean{
    int data;
    int getData(){
        return data;
    }
}
```
9.2 Cases and Queries in jCOLIBRI2

In jCOLIBRI2 cases and queries are composed of `CaseComponents`. A case will be divided into four components:

- Description of the problem.
- Solution to the problem.
- Result of applying the solution.
- Justification of the Solution (why this solution was chosen).

These components were chosen after reviewing the applications detailed in [54], but they are not required in some applications, so the developer can leave them empty (only the description component is mandatory).

Every query has always a description, so the `jcolibri.cbrcore.CBRQuery` object defines the queries as:

```java
public class CBRQuery {
    CaseComponent description;

    /**
     * Returns the description component.
     */
    public CaseComponent getDescription() { return description; }

    /**
     * Sets the description component.
     */
    public void setDescription(CaseComponent description) {
    }
}
```

You can create an `Attribute` object that represent the `data` attribute of `MyBean`. It is done with the following code:

```java
Listing 10: Using Attribute example code
Attribute at = new Attribute("data", MyBean.class);
```

The Attribute class allows jCOLIBRI2 to manage the contents of the cases (Java Beans).
If a query contains a description, we can say that a case is a query plus solution, result and justification of the solution. This way, the `jcolibri.cbrcore.CBRCase` extends `jcolibri.cbrcore.CBRQuery` adding those components:

Listing 12: CBRCase code

```java
public class CBRCase extends CBRQuery {
    CaseComponent solution;
    CaseComponent justificationOfSolution;
    CaseComponent result;

    /**
     * Returns the justificationOfSolution.
     */
    public CaseComponent getJustificationOfSolution() {
        return justificationOfSolution;
    }

    /**
     * Sets the Justification of Solution component.
     * @param justificationOfSolution to set.
     */
```
public void setJustificationOfSolution(CaseComponent justificationOfSolution) {
    this.justificationOfSolution = justificationOfSolution;
}

/**
 * Returns the result.
 */
public CaseComponent getResult() {
    return result;
}

/**
 * Sets the Result component
 */
public void setResult(CaseComponent result) {
    this.result = result;
}

/**
 * Returns the solution.
 */
public CaseComponent getSolution() {
    return solution;
}

/**
 * Sets the solution component
 */
public void setSolution(CaseComponent solution) {
    this.solution = solution;
}

public String toString() {
    return super.toString() + 
            "[Solution: " + solution + "]" + 
            "[Sol. Just.: " + justificationOfSolution + "]" + 
            "[Result: " + result + "]";
}

Each CaseComponent bean can have attributes that also are CaseComponents beans. This way, developers can create case structures with nested attributes. This feature is shown in the Test 3 of the examples.

The UML diagram in Figure 16 shows the relationship between cases, queries and case-components.
9.3 Case Components of the Travel Recommender

The cases of the Travel Recommender application are composed by a description and a solution. The description contains the Id, holiday type, number of persons, region, transportation, duration, season and accommodation type attributes. And the solution contains the Id, hotel and price attributes.

To create the description of the case you have to implement a Java Bean that contains those attributes and obeys the `CaseComponent` interface. This class is named `TravelDescription` and must be located under `jcolibri.example.TravelRecommender`. This is a portion of the code:

```java
public class TravelDescription implements jcolibri.cbrcore.CaseComponent {
    public enum AccommodationTypes { OneStar, TwoStars, ThreeStars, HolidayFlat, FourStars, FiveStars };
    public enum Seasons { January, February, March, April, May, June, July, August, September, October, November, December };

    String caseId;
    String HolidayType;
    Integer NumberOfPersons;
    Instance Region;
    String Transportation;
    Integer Duration;
    Seasons Season;
    AccommodationTypes Accommodation;
}
```
9.3 Case Components of the Travel Recommender

```java
public Attribute getIdAttribute() {
    return new Attribute("caseId", this.getClass());
}

public String toString() {
    return "("+caseId+";"+HolidayType+";"+NumberOfPersons+
    ;"+Region+";"+Transportation+";"+Duration+";"+Season
    +";"+Accommodation+");";
}
...";
```

Now you have to add a get() and set() method for each attribute. But, don’t worry because Eclipse does it automatically selecting the menu item: “Source - Generate Getters and Setters...”.

In this description we are using two enumerate types to define the accommodation and season. Also, there is an strange type named Instance that defines the region. This type is defined in jcolibri.datatypes.Instance and represents an instance of an ontology. Now, we are not going into detail with this because it will be explained in Section 12.4.

The solution bean must be created in a similar way. Its name is TravelSolution and this is the code without the getters and setters (that you have to generate):

```java
public class TravelSolution implements jcolibri.cbrcore.CaseComponent {
    String id;
    Integer price;
    String hotel;

    public String toString() {
        return "("+id+";"+price+";"+hotel+")";
    }

    public Attribute getIdAttribute() {
        return new Attribute("id", this.getClass());
    }
...";
```
9.4 Defining new attribute types

Sometimes, the Java built-in types are not enough and the developer has to define her own ones. This is the case of the Region attribute of the TravelDescription bean that uses a custom data type named Instance.

When defining a new type the developer has to specify how it is going to be stored and in the persistence media. This is done by implementing the jcolibri.connectors.TypeAdaptor interface:

```java
public interface TypeAdaptor {
    /**
     * Returns a string representation of the type.
     */
    public abstract String toString();

    /**
     * Reads the type from a string.
     */
    public abstract void fromString(String content) throws Exception;

    /**
     * You must define this method to avoid problems with the database connector (Hibernate)
     */
    public abstract boolean equals(Object o);
}
```

jCOLIBRI2 includes two examples of user-defined data types in the jcolibri.datatypes package. Moreover, the Test 2 of the examples explains this mechanism in detail.
10 The two layers persistence architecture of jCOLIBRI2

The CaseComponents defined in the previous section are used to fill the attributes of the CBRCase objects when loading the cases. Before explaining this mechanism we have to depict the two layers persistence architecture of jCOLIBRI2.

CBR systems must access the stored cases in an efficient way, a problem that becomes more relevant as the size of the Case Base grows. jCOLIBRI (both versions 1 and 2) splits the problem of Case Base management in two separate although related concerns: persistence mechanism and in-memory organization. This architecture is illustrated in Figure 17.

![Figure 17: Case Base management in jCOLIBRI2](image)

10.1 The Connectors of jCOLIBRI2

Persistence is built around connectors. A connector represents the first layer of jCOLIBRI on top of the physical storage. Connectors are objects that know how to access and retrieve cases from the medium and return those cases to the CBR system in a uniform way. The use of connectors give jCOLIBRI flexibility against the physical storage so the system designer can choose the most appropriate one for the system at hand.

jCOLIBRI2 includes the following connectors:

- jcolibri.connectors.DataBaseConnector. Manages the persistence of cases into databases. Internally, it uses the Hibernate library.
- jcolibri.connectors.PlainTextConnector. Manages the persistence of the cases into textual files.
• jcolibri.connectors.OntologyConnector. It uses OntoBridge² to manage case bases stored into ontologies.

The obvious interface for a connector must include methods to read the Case Base into memory and update it back into persistent media. More specifically, jCOLIBRI2 includes an interface named Connector that belongs to package jcolibri.cbrcore. Every connector is supposed to implement the methods defined by this interface:

Listing 16: Connector interface

```java
public interface Connector {

    /**
     * Initialises the connector with the given XML file
     */
    public void initFromXMLfile(java.net.URL file) throws InitializingException;

    /**
     * Cleanup any resource that the connector might be using, 
     * and suspends the service
     */
    public void close();

    /**
     * Stores given classes on the storage media
     */
    public void storeCases(Collection<CBRCase> cases);

    /**
     * Deletes given cases for the storage media
     */
    public void deleteCases(Collection<CBRCase> cases);

    /**
     * Returns all the cases in the storage media
     */
    public Collection<CBRCase> retrieveAllCases();

    /**
     * Retrieves some cases depending on the filter. TODO.
     */
    public Collection<CBRCase> retrieveSomeCases(CaseBaseFilter filter);
}
```

²OntoBridge (http://gaia.fdi.ucm.es/projects/ontobridge/) is a library developed by the GAIA team that eases the management of ontologies.
Connectors are configured through XML configuration files. Each jCOLIBRI2 connector defines the XML schema of its configuration file. These schemes can be found in the documentation.

An interface such that assumes that the whole Case Base can be read into memory for the CBR processes to work with it. However, in a real sized CBR application this approach may not be feasible. For that reason, we are working to extend connector interface to retrieve those cases that satisfy a query expressed in a subset of SQL (retrieveSomeCases(CaseBaseFilter)). This way the designer can decide what part of the Case Base is loaded into memory.

If a developer requires a specific connector, she can create her own one extending the Connector interface. This is shown in the Test 13 of the code examples of the framework.

### 10.2 In-memory organization of the cases

The second layer of Case Base management is the data structure used to organize the cases once loaded into memory. The organization of the case base (linear, k-d trees, case retrieval nets, etc.) may have a big influence on the CBR processes, so the framework leaves open the election of the data structure that is going to be used.

In the same way connectors offer a common interface to the Case Base, the Case Base also implements a common interface for the CBR methods to access the cases. This way the organization and indexation chosen for the Case Base will not affect the implementation of the methods. The interface that defines the in-memory organization of the cases is jcolibri.cbrcore.CBRCaseBase:

```java
public interface CBRCaseBase {

    /**
     * Initializes the case base. This method receives
     * the connector that manages the persistence media.
     */
    public void init(Connector connector) throws jcolibri.exception.InitializingException;

    /**
     * De-Initializes the case base.
     */
    public void close();

    /**
     * Returns all the cases available on this case base
     */
    public Collection<CBRCase> getCases();

    // Other methods...
}
```

Listing 17: CBRCaseBase interface
Analogous to the Connector interface, developers can create their in-memory organizations of cases implementing the `CBRCaseBase` interface. jCOLIBRI2 includes the following Case Bases:

- `jcolibri.casebase.LinealCaseBase`: Basic Lineal Case Base that stores cases into a List.
- `jcolibri.casebase.CachedLinealCaseBase`: Cached case base that only persists cases when closing the application.
- `jcolibri.casebase.IDIndexedLinealCaseBase`: Extension of `LinealCaseBase` that also keeps an index of cases using their IDs.
11 Loading the Case Base

Now, let’s add the code for loading the cases of our Travel Recommender application. We are using the Data Base connector (implemented with Hibernate) and the Lineal Case Base. The first step consists on modifying our TravelRecommender class with the following code:

**Listing 18:** TravelRecommender configure() (version 2) and precycle()

```java
// ** Connector object */
Connector _connector;

// ** CaseBase object */
CBRCaseBase _caseBase;

public void configure() throws ExecutionException {
    try {
        // Emulate data base server
        jcolibri.test.database.HSQLDBserver.init();

        // Create a data base connector
        _connector = new DataBaseConnector();

        // Init the ddbb connector with the config file
        _connector.initFromXMLfile(jcolibri.util.FileIO
                                    .findFile("jcolibri/examples/
                                                TravelRecommender/databaseconfig.xml");

        // Create a Lineal case base for in-memory organization
        _caseBase = new LinealCaseBase();

        // Create the dialogs
    }
    catch (Exception e) {
        throw new ExecutionException(e);
    }

    public CBRCaseBase precycle() throws ExecutionException {
        // Load cases from connector into the case base
        _caseBase.init(_connector);

        // Print the cases
        java.util.Collection<CBRCase> cases = _caseBase.getCases();
        for (CBRCase c: cases)
            System.out.println(c);
        return _caseBase;
    }
```
Firstly, we are creating the two variables that will contain the connector and case base: _connector and _caseBase. They are defined with the type of the interfaces Connector and CBRCaseBase, but in the configure() method we will assign an instance of DataBaseConnector and LinealCaseBase that implement these interfaces. As explained before, each connector can use a xml file that defines its configuration. In this case, we are using the file jcolibri/examples/TravelRecommender/-databaseconfig.xml to configure the Data Base connector. This file is explained in the following subsection.

In the preCycle() method we initializes the case base object with the connector through the init() method. This action will load the cases from the persistence into the memory. Then we can access the cases in the case base object (here to print them to console).

### 11.1 Configuring the connector

The data base connector is configured with the databaseconfig.xml file. So, next step consists on creating this file in the jcolibri.examples.TravelRecommender package.

The schema of this file is explained in the documentation of the connector. Here you should write this:

**Listing 19: databaseconfig.xml**

```xml
<DataBaseConfiguration>
  <HibernateConfigFile>
    jcolibri/examples/TravelRecommender/hibernate.cfg.xml
  </HibernateConfigFile>
  <DescriptionMappingFile>
    jcolibri/examples/TravelRecommender/TravelDescription.hbm.xml
  </DescriptionMappingFile>
  <DescriptionClassName>
    jcolibri.examples.TravelRecommender.TravelDescription
  </DescriptionClassName>
  <SolutionMappingFile>
    jcolibri/examples/TravelRecommender/TravelSolution.hbm.xml
  </SolutionMappingFile>
  <SolutionClassName>
    jcolibri.examples.TravelRecommender.TravelSolution
  </SolutionClassName>
</DataBaseConfiguration>
```

This file defines the following information:

- HibernateConfigFile: locates the configuration file of Hibernate.
11.1 Configuring the connector

- **DescriptionMappingFile**: locates the mapping file of the description.
- **DescriptionClassName**: class that represents the solution.
- **SolutionMappingFile**: locates the mapping file of the solution.
- **SolutionClassName**: class that represents the description.

The solution and description classes are the ones developed in Section 9.3. Using these fields, the connector knows which are the **CaseComponents** that fill the **description**, **solution**, **justificationOfSolution** and **result** attributes of the **CBRCase**.

The mapping files define how to map those classes into the tables of the database. And finally, the Hibernate configuration file configures the connection with the DBMS.

### 11.1.1 The Hibernate configuration file

This file tells Hibernate how to access the DBMS. If you need special requirements read the Hibernate documentation located here: [http://www.hibernate.org/hib_docs/v3/reference/en/html/session-configuration.html](http://www.hibernate.org/hib_docs/v3/reference/en/html/session-configuration.html)

To configure Hibernate to work with our HSQLDB Data Base Manager you must create the file `hibernate.cfg.xml` into the `jcolibri.examples.TravelRecommender` package. Following listing shows the content:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE hibernate−configuration PUBLIC
"−//Hibernate/Hibernate−Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate−configuration−3.0.dtd">
<hibernate−configuration>
    <session−factory>
        <!-- Database connection settings -->
        <property name="connection.driver_class">
            org.hsqldb.jdbcDriver
        </property>

        <property name="connection.url">
            jdbc:hsqldb:hsqldb://localhost/travel
        </property>

        <property name="connection.username">
            sa
        </property>

        <property name="connection.password">
        </property>
    </session−factory>
</hibernate−configuration>
```

---

Group for Artificial Intelligence Applications  
jCOLIBRI2 Tutorial
To use Hibernate with other DBMs, developers should modify the following properties (described in the Hibernate documentation at http://www.hibernate.org/hib_docs/v3/reference/en/html/session-configuration.html#configuration-hibernatejdbc):

- `connection.driver_class`: jdbc driver class of your DBMs (must be included in the classpath).
- `connection.url`: jdbc connection url.
- `username` and `password`.

These are the small changes required to use the Hibernate connector with other DBM. Now, we must define the mapping files.
11.1 Configuring the connector

11.1.2 Creating the mapping files

The mapping files define how to map a Java Bean with a table of the data base. Firstly, you define which table is used to store the bean. Then you configure which column of the table contains each attribute of the bean.

Our cases were composed by a description represented in the TravelDescription class, and a solution represented in the TravelSolution class. We are going to map both components to the same table. Some columns will store the attributes of the description and other will store the attributes of the solution. Figure 18 illustrates this mapping (the caseId column is in blue because it is used by both components).

Let’s begin with the mapping file of the solution. You must create the file TravelSolution.hbm.xml into jcolibri.example.travelrecommender with the following content:

Listing 21: TravelSolution.hbm.xml

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate−mapping PUBLIC
"−//Hibernate/Hibernate−Mapping−DTD//EN"
"http://hibernate.sourceforge.net/hibernate−mapping−3.0.dtd">
<hibernate−mapping default−lazy="false">
  <class
    name="jcolibri.examples.TravelRecommender.TravelSolution"
    table="travel">
    <id name="id" column="caseId"/>
    <property name="price" column="Price"/>
    <property name="hotel" column="Hotel"/>
  </class>
</hibernate−mapping>
```

Figure 18: Travel Recommender components mapping
The **class** tag sets the `jcolibri.examples.TravelRecommender.TravelSolution` class to be stored into the `travel` table.

The **id** attribute of TravelSolution is mapped to the `caseId` column. It is done through the **id** tag that defines the attribute as the primary key of the table.

Finally, the remaining attributes are mapped to the other columns of the table using the **property** tag.

Now, we have to create the mapping file of the description. The file name is `TravelDescription.hbm.xml` and must be located into `jcolibri.example.travelrecommender`:

```xml
<hibernate-mapping default-lazy="false">
  <class name="jcolibri.examples.TravelRecommender.TravelDescription" table="Travel">
    <id name="caseId" column="caseId">
      <generator class="native"/>
    </id>
    <property name="HolidayType" column="HolidayType"/>
    <property name="NumberOfPersons" column="NumberOfPersons"/>
    <property name="Region" column="Region">
      <type name="jcolibri.connector.databaseutils.GenericUserType">
        <param name="className">jcolibri.datatypes.Instance</param>
      </type>
    </property>
    <property name="Transportation" column="Transportation"/>
    <property name="Duration" column="Duration"/>
    <property name="Season" column="Season">
      <type name="jcolibri.connector.databaseutils.EnumUserType">
        <param name="enumClassName">jcolibri.examples.TravelRecommender.TravelDescription$Seasons</param>
      </type>
    </property>
    <property name="Accommodation" column="Accommodation">
      <type name="jcolibri.connector.databaseutils.EnumUserType"/>
    </property>
  </class>
</hibernate-mapping>
```
Here, we are mapping again the `jcolibri.examples.TravelRecommender.TravelDescription` class into the `travel` table using the `class` tag.

Then we map the `caseId` attribute to the column with the same name `caseId`. This attribute is the primary key of the table because we are mapping it using the `id` tag. Inside this tag we found the `generator` tag that defines how to define unique ids for new Beans that will be inserted in the table. Defining this property as `native`, Hibernate delegates this task to the underlying DBM.

After this tag we continue with the `property` tags that map normal attributes to the columns of the database. However there are some of them that have the `type` child tag. Those attributes have a special mapping because are user-defined types like the `region` attribute or because they are enumerates like `season` or `accommodation`.

To map a user-defined type with Hibernate we have to use the `type` tag with the `jcolibri.connector.databaseutils.GenericUserType` value. This class allows to map any user-defined type indicated by the `param` tag. As explained in Section 9.4, the user-defined types in `jCOLIBRI2` must implement the `jcolibri.connector.TypeAdaptor` interface. This way, the general template to map user-defined types is:

```
<property name="beanAttribute" column="tableColumn">
    <type name="jcolibri.connector.databaseutils.GenericUserType">
        <param name="className">
            implementation_of_TypeAdaptor_that_defines_attribute_type
        </param>
    </type>
</property>
```

The other special configuration is the mapping of enumerates. To do this, we have to use the `jcolibri.connector.databaseutils.EnumUserType` class in the `type` tag and then indicate the class of the enumerate into the `param` tag. The general template is:

```
<property name="beanAttribute" column="tableColumn">
    <type name="jcolibri.connector.databaseutils.EnumUserType">
        <param name="enumClassName">
```

---

Group for Artificial Intelligence Applications  jCOLIBRI2 Tutorial
11.1 Configuring the connector

```xml
<property name="enumerate_class_that_defines_attribute_type">
  <param name="type">
    <type />
  </param>
</property>
```

12 Using Ontologies in CBR applications

For the last few years our research group has been working in Knowledge Intensive CBR using ontologies [23, 16, 17, 19]. We claim that ontologies are useful for designing KI-CBR applications because they allow the knowledge engineer to use knowledge already acquired, conceptualized and implemented in a formal language, like DLs based languages, reducing considerably the knowledge acquisition bottleneck. Our approach proposes the use of ontologies to build models of general domain knowledge. Although in a CBR system the main source of knowledge is the set of previous experiences, our approach to CBR is towards integrated applications that combine case specific knowledge with models of general domain knowledge. The more knowledge is embedded into the system, the more effective is expected to be. Semantic CBR processes can take advantage of this domain knowledge and obtain more accurate results.

We state that the formalization of ontologies is useful for the CBR community regarding different purposes, namely:

1. Persistence of cases and/or indexes using individuals or concepts that are embedded in the ontology itself.

2. As the vocabulary to define the case structure, either if the cases are embedded as individuals in the ontology itself, or if the cases are stored in a different persistence media as a data base.

3. As the terminology to define the query vocabulary. The user can express better his requirements if he can use a richer vocabulary to define the query. During the similarity computation the ontology allows to bridge the gap between the query terminology and the case base terminology.

4. Retrieval and similarity [22, 46, 38], adaptation [23] and learning [1].

5. Knowledge reuse between different CBR systems.

jCOLIBRI2 allows to implement CBR systems with all these features or, at least, offers an ontology management architecture that is the basis of this kind of CBR applications.

The ontology support of jCOLIBRI2 is built around the OntoBridge library (http://gaia.fdi.ucm.es/grupo/projects/ontobridge/). This library was developed by the GAIA research group to easily manage ontologies and DLs reasoners. It is base on the JENA\textsuperscript{3} library and allows to reason with several DL reasoners like Pellet\textsuperscript{4}.

\begin{itemize}
\item \textsuperscript{3}http://jena.sourceforge.net/
\item \textsuperscript{4}http://pellet.owldl.com
\end{itemize}
12.1 Case Base persistence in ontologies

jCOLIBRI2 allows to use ontologies as a persistence media for the cases. The connectors architecture of the framework eases the incorporation of this feature as the single requirement is the implementation of a new connector. This connector is `jcolibri.connector.OntologyConnector`. To learn how to use it, read the documentation of the Test 10 in the examples. This example shows how to map a case (similar to the Travel Recommender case of this tutorial) into an ontology. Figure 19 illustrates this process.

![Figure 19: Case mapping in an ontology](image)

To explain in detail how works the `OntologyConnector` let’s use the example shown in Figure 20. A concept of the ontology will define the cases (VACATION_CASE) and the other related concepts define the attributes (CATEGORY, PRICE and HOLIDAY_TYPE). This way, the `OntologyConnector` obtains the instances of the VACATION_CASE concept (case1 and case2) and follows their relationships to obtain the values of the attributes (That will be instances of the concepts that define the attributes: CATEGORY, PRICE and HOLIDAY_TYPE).

12.2 Computing similarities using ontologies

There are different approaches for computing similarity measures based on ontologies that are provided in jCOLIBRI2. It initially offers four functions to compute the concept
12.3 Case and Query vocabulary

Regarding the case vocabulary, there is a direct approach consisting on the use of a domain ontology in an object-oriented way: concepts are types, or classes, individuals are allowed values, or objects, and relations are the attributes describing the objects. There are also simple types like string or numbers that are considered in the traditional way. The case structure is defined using types from the ontology even if the cases are not stored as individuals in the ontology. For example, the Region concept of an ontology can be used as a type where every one of its instances are the type values: Madrid, Barcelona, Sevilla, ...

Regarding the query vocabulary we have two options to define the queries:

- Using exactly the same vocabulary used in the cases, i.e, the same types used in the case structure definition.

- Using the ontology as the query vocabulary, what allows richer queries. The user can express better his requirements if he can use a richer vocabulary to define the query. During the similarity computation the ontology allows to bridge the gap between the query terminology and the case base terminology.
12.3 Case and Query vocabulary

\[ \text{fdeep_basic}(i_1, i_2) = \frac{\max(\text{prof}(\text{LCS}(i_1, i_2)))}{\max C \in \text{CN} \text{prof}(C)} \]

\[ \text{fdeep}(i_1, i_2) = \frac{\max(\text{prof}(\text{LCS}(i_1, i_2)))}{\max(\text{prof}(i_1), \text{prof}(i_2))} \]

\[ \text{cosine}(i_1, i_2) = \frac{\left| \bigcup_{d_i \in t(i_1)} (\text{super}(d_i, CN)) \right| \cap \left| \bigcup_{d_i \in t(i_2)} (\text{super}(d_i, CN)) \right|}{\sqrt{\left| \bigcup_{d_i \in t(i_1)} (\text{super}(d_i, CN)) \right|} \cdot \sqrt{\left| \bigcup_{d_i \in t(i_2)} (\text{super}(d_i, CN)) \right|}} \]

\[ \text{sim}(t(i_1), t(i_2)) = \frac{1}{2 \cdot \left( \left| \bigcup_{d_i \in t(i_1)} (\text{super}(d_i, CN)) \right| \cap \left| \bigcup_{d_i \in t(i_2)} (\text{super}(d_i, CN)) \right| \right)} \]

\[ \text{detail}(i_1, i_2) = \text{detail}(t(i_1), t(i_2)) = 1 - \frac{1}{2 \cdot \left( \left| \bigcup_{d_i \in t(i_1)} (\text{super}(d_i, CN)) \right| \cap \left| \bigcup_{d_i \in t(i_2)} (\text{super}(d_i, CN)) \right| \right)} \]

Where:

- \( CN \) is the set of all the concepts in the current knowledge base
- \( \text{super}(c, C) \) is the subset of concepts in \( C \) which are superconcepts of \( c \)
- \( \text{LCS}(i_1, i_2) \) is the set of the least common subsumer concepts of the two given individuals
- \( \text{prof}(c) \) is the depth of concept \( c \)
- \( t(i) \) is the set of concepts the individual \( i \) is instance of

**Example:** In the travel domain, let's suppose we have an existing case base where it is defined an enumerated type for the Region attribute where the allowed values are countries: Spain, France, Italy and others. Suppose that case \( i \) is a case whose destination is Spain. We do not want to restrict the query vocabulary to the same type but allow broader queries, for example:

- **Query1:** "I want to go to Madrid"
- **Query2:** "My favorite destination is Europe"
- **Query3:** "I would like to travel to Spain"

In the three queries and using the ontology of Figure 22 we could find case \( i \) as a suitable candidate.
12.4 Using an ontology in the Travel Recommender

Our Travel Recommender application uses an ontology of regions used to define the type or the Region attribute. It also defines the query vocabulary of that attribute.

The ontology stores information about travel destinations, is located at http://gaia.fdi.ucm.es/ontologies/travel-destinations.owl and was developed using the PROTEGE\(^5\) ontology editor. Figure 23 shows how this ontology is used to define the query in the Travel Recommender application. When the user clicks on the Region button, a dialog (generated by the OntoBridge library) shows the ontology allowing to choose the desired destination.

To use an ontology in a CBR application, developers only have to initialize OntoBridge. Then, the ontology connector and similarity measures will use that library to operate with the ontology. In the Travel Recommender application, we only have to add the code shown in the following listing to the configure() method of the TravelRecommender class (this listing is the final state of that method):

```java
public void configure() throws ExecutionException {
    try {
        // Emulate database server
        jcolibri.test.database.HSQLDBserver.init();

        // Create a database connector
        _connector = new DataBaseConnector();
        // Init the ddbb connector with the config file
        _connector.initFromXMLfile(jcolibri.util.FileIO
```
Figure 23: Defining the Travel Recommender query through an ontology

```java
// Create a LINEAL case base for in-memory organization
_caseBase = new LINEALCaseBase();

// Obtain a reference to OntoBridge
OntoBridge ob = jcolibri.util.OntoBridgeSingleton.getOntoBridge();
// Configure it to work with the Pellet reasoner
ob.initWithPelletReasoner();
// Setup the main ontology
mainOnto.addSubOntologies(new ArrayList<OntologyDocument>());

// Create the dialogs
similarityDialog = new SimilarityDialog(main);
```
12.4 Using an ontology in the Travel Recommender

```java
resultDialog = new ResultDialog(main);
autoAdaptDialog = new AutoAdaptationDialog(main);
revisionDialog = new RevisionDialog(main);
retainDialog = new RetainDialog(main);

} catch (Exception e) {
    throw new ExecutionException(e);
}
```

The remaining steps to use the ontology have been already implemented. In the TravelDescription code (Listing 13) we defined the type of the Region attribute as Instance. This way, the jCOLIBRI2 methods will connect to OntoBridge to manage that attribute. Moreover, we need to indicate the connector how to manage this attribute, but we did that when defining the mapping file of TravelDescription in TravelDescription.hbm.xml. Then, our connector will use OntoBridge to link the values in the database with the instances of the loaded ontology.

In this case we are not storing the whole case base into the ontology. The ontology only defines the type and values of the Region attribute. The data base connector will read the values in the table and look for an instance with the same name in the ontology (through OntoBridge). Once found, the connector will fill the value of the Region attribute of each case with the corresponding instance of the ontology. This is shown in Figure 24 and the code required to map the database with the ontology appears in Listing 22.

![Figure 24: Mapping between data base and ontology](image.png)
13 Retrieval

At this point, we have completed the configure() and precycle() methods of our Travel Recommender application. Those methods allow us to load the cases from the persistence into memory. The precycle() method loads the cases and stores them into the _caseBase object of our main application (review Listing 18). This section and the following ones show how to fill the cycle() method to perform the 4R’s tasks (retrieval, reuse, revise, and retain).

The retrieval step obtains the most similar cases given a query. The main method of jCOLIBRI2 for computing the retrieval is the jcolibri.method.retrieve.NNretrieval.NNScoringMethod class. This method performs a Nearest Neighbor numeric scoring comparing attributes. It uses global similarity functions to compare compound attributes (CaseComponents) and local similarity functions to compare simple attributes.

For example, the TravelDescription case component of our cases is a compound attribute composed by several simple attributes (season, duration, etc.). So, we assign a global similarity function to the description like the average function. Then, simple similarity functions like equal, numeric interval, enumerate distance, ... are configured for each simple attribute. The NNScoringMethod will compute the similarity of each simple attribute and then compute the global similarity: the average of the simple similarities.

The configuration of those similarity functions is stored in the jcolibri.method.retrieve.NNretrieval.NNConfig object. The values configured in this object are:

- Global similarity function for the description.
- Global similarity functions for each compound attribute (CaseComponents excepting the description).
- Local similarity functions for each simple attribute.
- Weight for each attribute.

Following listing shows the signature of the NNScoringMethod:

```
Listing 26: NNScoringMethod signature

/**
 * Performs the NN scoring over a collection of cases comparing them with a query.
 * This method is configured through the NNConfig object.
 */
public static Collection<RetrievalResult> evaluateSimilarity(
    Collection<CBRCase> cases, CBRQuery query, NNConfig simConfig)
```

Group for Artificial Intelligence Applications

jCOLIBRI2 Tutorial
The parameters of this method are very intuitive: the cases to compare with the query, the query and the similarity configuration. The method returns a collection of `jcolibri.method.retrieve.RetrievalResult` objects. These objects contain the retrieved case and a double that represents the similarity of that case to the query.

### 13.1 Similarity functions

There are many similarity functions in the `jcolibri.method.retrieve.NNretrieval.similarity` package. Each local similarity measure can only be applied to some datatypes. This way there are local similarities applicable to integers, strings, instances, ...

Developers can define their own similarity measures implementing the interfaces: `jcolibri.method.retrieve.NNretrieval.similarity.GlobalSimilarityFunction` and `jcolibri.method.retrieve.NNretrieval.similarity.LocalSimilarityFunction`. The signatures of both interfaces are:

```java
public interface GlobalSimilarityFunction {

 /**
  * Computes the global similarity between two compound attributes.
  * It requires the NNConfig object that stores the configuration of its contained attributes.
  * @param componentOfCase compound attribute of the case
  * @param componentOfQuery compound attribute of the query
  * @param _case case being compared
  * @param _query query being compared
  * @param numSimConfig Similarity functions configuration
  * @return a value between [0..1]
  */
  public double compute(CaseComponent componentOfCase, CaseComponent componentOfQuery, CBRCase _case, CBRQuery _query, NNConfig numSimConfig);
}

public interface LocalSimilarityFunction {

 /**
  * Computes the similarity between two objects
  * @param caseObject object of the case
  * @param queryObject object of the query
  * @return a value between [0..1]
  */
```

**Listing 27: Local and Global similarity interfaces**
public double compute(Object caseObject, Object queryObject)
    throws jcolibri.exception.NoApplicableSimilarityFunctionException;

/**
 * Indicates if the function is applicable to two objects
 */
public boolean isApplicable(Object caseObject, Object queryObject);

As the previous listing shows, the local similarity functions cannot access to the other attributes of the case to compute their measures. This can be a problem in some applications where the similarity of two attributes depends on the value of other attributes in the same case component. To solve this drawback, jCOLIBRI2 includes the jcolibri.method.retrieve.NNretrieval.similarity.InContextLocalSimilarityFunction abstract class that incorporates information about the context (CaseComponent) of the attribute. Extended information about this class can be found in its documentation.

13.2 Cases selection

Most similar cases must be selected once they have been scored according to their similarity with the query. Usually, only the top k most similar cases are selected. This retrieval process that combines Nearest Neighbor scoring and top k selection is commonly called k-NN retrieval.

The jcolibri.method.retrieve.selection.SelectCases class includes basic methods to select cases:

Listing 28: Cases selection methods

//
*/
public static Collection<CBRCase> selectAll(Collection<RetrievalResult> cases)

/**
 * Selects top K cases
 * @param cases to select
 * @param k is the number of cases to select
 * @return top k cases
 */
public static Collection<CBRCase> selectTopK(Collection<RetrievalResult> cases, int k)
Previous listing shows that there are two versions of the methods: one returning CBRCASE objects and other returning RetrievalResult object.

There are another more sophisticated ways to select cases. Sometimes it is important to select cases that are similar to the query but also diverse among them. jCOLIBRI2 includes some of these methods in the jcolibri.method.retrieve.selection. As these methods were included in the recommenders extension of the 2.1 version, they are detailed in Section 20.2.9.

Note that there are important changes in the k-NN retrieval implementation from version 2.0 to version 2.1. These changes are detailed in Section 24.

### 13.3 Using the k-NN retrieval

There are many examples in the documentation of jCOLIBRI2 that show how to configure the NNScoringMethod. In the Travel Recommender example this code is included in the jcolibri.examples.TravelRecommender.gui.SimilarityDialog.getSimilarityConfig() method. As this code obtains the similarity values from the graphical widgets, here we will show how to configure and execute the NN method in a direct way. The following code is extracted from the first Test of the documentation:

**Listing 29: Example code for the k-NN Retrieval**

```java
public void cycle(CBRQuery query) throws ExecutionException {
    // First configure the NN scoring
```
NNConfig simConfig = new NNConfig();
// Set the average() global similarity function for the
description of the case
simConfig.setDescriptionSimFunction(new Average());
// The accommodation attribute uses the equal() local
similarity function
simConfig.addMapping(new Attribute("Accomodation",
    TravelDescription.class), new Equal());
// For the duration attribute we are going to set its local
similarity function and the weight
Attribute duration = new Attribute("Duration",
    TravelDescription.class);
simConfig.addMapping(duration, new Interval(31));
simConfig.setWeight(duration, 0.5);
// HolidayType --> equal()
simConfig.addMapping(new Attribute("HolidayType",
    TravelDescription.class), new Equal());
// NumberOfPersons --> equal()
simConfig.addMapping(new Attribute("NumberOfPersons",
    TravelDescription.class), new Equal());
// Price --> Interval()
simConfig.addMapping(new Attribute("Price", TravelDescription.
    class), new Interval(4000));

// Execute NN
Collection<RetrievalResult> eval = NNScoringMethod.
    evaluateSimilarity(_caseBase.getCases(), query, simConfig)
    ;

// Select k cases
eval = SelectCases.selectTopKRR(eval, 5);

// Print the retrieval
System.out.println("Retrieved Cases:");
for (RetrievalResult nse: eval)
    System.out.println(nse);

// Finally remove the retrieval info (the similarity value)
// and get only the cases
Collection<jcolibri.cbrcore.CBRCase> retrievedCases =
    RemoveRetrievalEvaluation.removeRetrievalEvaluation(eval);
}

The code is very simple: the NNConfig class has methods to set the similarity function and weight for the attributes. The attributes of a bean are represented using the jcolibri.cbrcore.Attribute class as explained in Section 9.1. Once configured, NNScoringMethod.evaluateSimilarity() is executed obtaining a list of RetrievalResult objects that contain the most similar cases to the query. Fi-
nally, we obtain the 5 most similar cases using `SelectCases.selectTopKRR()`. This method returns the top k cases as `RetrievalResult` objects (keeping the similarity value) although there is another method that removes that information and only returns `CBRCase` objects (`SelectCases.selectTopK()`).

### 13.4 Retrieval in the Travel Recommender application

The retrieval code in the Travel Recommender application is very simple because the `SimilarityDialog` window gives us the `NNConfig` object and the k value. So, we only have to compute the k-NN, show the result using the `ResultDialog` window and finally, remove the retrieval evaluation to obtain the list of cases. Add this code into the `cycle()` method of the `TravelRecommender` class:

```java
public void cycle(CBRQuery query) throws ExecutionException {
    // Obtain configuration for k-NN
    similarityDialog.setVisible(true);
    NNConfig simConfig = similarityDialog.getSimilarityConfig();
    simConfig.setDescriptionSimFunction(new Average());

    // Execute NN
    Collection<RetrievalResult> eval = NNScoringMethod.
        evaluateSimilarity(_caseBase.getCases(), query, simConfig);

    // Select k cases
    Collection<CBRCase> selectedcases = SelectCases.selectTopK(
        eval, similarityDialog.getK);

    // Show result
    resultDialog.showCases(eval, selectedcases);
    resultDialog.setVisible(true);
    ...}
```

With this code we have implemented the functionality shown in Figures 7 and 8.
14 Reuse

The reuse step (also named adaptation step) adapts the solution of the retrieved cases to the requirements of the query. This step is very domain dependent and use to vary depending on the application. jCOLIBRI2 leaves this step open to developers. They should create their own adaptation methods customized for the application.

Anyway, jCOLIBRI2 offers two basic adaptation methods:

- `jcolibri.method.reuse.DirectAttributeCopyMethod`. This method copies the value of an attribute in the query to an attribute of a case.

Besides these methods, the `jcolibri.method.reuse.classification` package includes some classification reuse methods implemented by Lisa Cummins & Derek Bridge. (University College Cork, Ireland).

Ontologies can be used to guide the adaptation of the cases. There are some experimental methods (not included in the framework) that are explained in [43].

Once the retrieved cases are adapted, their description can be substituted by the description of the query. This way, we obtain a list of cases with the same description than the query. This step is performed by the `jcolibri.method.reuse.CombineQueryAndCasesMethod` method, and is optional depending on the application.

After adapting the cases, the CBR system proposes them as a suggested solution of the problem (review Figure 1). Then, the system user (or a domain expert) could revise these solutions in the following step.

### 14.1 Adapting the Travel Recommender retrieved trips

In this tutorial we are using the `NumericDirectProportionMethod` method to adapt the solution of the trips according with the description of the query.

We are going to use this method to compute the following proportions:

- **NumberOfPersons** and **Price**. Imagine that the number of persons configured in the query is 4 and the value of this attribute in the case is 2. In the solution of the case, the price is 1000, so we have to compute a direct proportion among these attributes to adapt the solution to the requirements of the query. After executing the method, the new value of the price in the case solution will be 2000.

- **Duration** and **Price**. This adaptation is analogous to the previous one, but using the `Duration` attribute instead of `NumberOfPersons`.
To perform these adaptations in the Travel Recommender application append the following code to the `cycle()` method:

```
Listing 31: TravelRecommender.cycle() code (step2)

...  
// Show adaptation dialog
autoAdaptDialog.setVisible(true);

// Adapt depending on user selection
if (autoAdaptDialog.adapt_Duration_Price())
{
    // Compute a direct proportion between the "Duration" and "Price" attributes.
    NumericDirectProportionMethod.directProportion(new Attribute("Duration", TravelDescription.class),
                                                   new Attribute("price", TravelSolution.class), query, selectedcases);
}

if (autoAdaptDialog.adapt_NumberOfPersons_Price())
{
    // Compute a direct proportion between the "Duration" and "Price" attributes.
    NumericDirectProportionMethod.directProportion(new Attribute("NumberOfPersons", TravelDescription.class),
                                                   new Attribute("price", TravelSolution.class), query, selectedcases);
}
...
```

This code shows the dialog in Figure 9 and performs the required adaptations.
15 Revise

In the revise step the proposed solution is tested for success, e.g. by being applied to the real world environment or evaluated by a domain expert, and repaired if failed.

This step is also very domain dependent and may change among applications. jCOLIBRI2 only includes a method to define new Ids to the cases as they will be stored into the data base during the following step and they cannot use the original Id of the retrieved cases. This method is `jcolibri.method.revise.DefineNewIdsMethod`.

There are also some classification revise methods implemented by Lisa Cummins & Derek Bridge. (University College Cork, Ireland) in the `jcolibri.method.revise.classification` package.

15.1 Travel Recommender revision

The revise step in the Travel Recommender is extremely simple because the `jcolibri.examples.TravelRecommender.gui.RevisionDialog` class is in charge of presenting the cases for edition and modify them according with the user selections.

The code is:

```
// Revise
revisionDialog.showCases(selectedCases);
revisionDialog.setVisible(true);
```

These methods will show the window illustrated in Figure 10.
16 Retain

In the retain step useful new cases are stored in the case base for future reuse. This way the CBR system has learned a new experience.

jCOLIBRI2 includes the jcolibri.method.retain.StoreCasesMethod to include new cases into the case base. The new added cases will be stored in the persistence media depending on the chosen implementation of CBRCaseBase. The LinealCaseBase class will store the cases directly into the persistence layer, but the CachedLinealCaseBase will keep the new cases into memory and save them only when closing the CBR application (this is, when the postCycle() method is invoked).

16.1 Saving the new trips of the Travel Recommender

The following code shows the retain window in the application and calls the StoreCasesMethod method. The retain window allows the user to select which cases must be added to the case base and to define a new Id for each one. Here, this retain dialog is in charge of defining the Ids instead of using the jcolibri.method.revise.DefineNewIdsMethod method.

```java
Listing 33: TravelRecommender.cycle() code (step4)

// Retain
retainDialog.showCases(selectedCases, _caseBase.getCases().size());
retainDialog.setVisible(true);
Collection<CBRCase> casesToRetain = retainDialog.getCasestoRetain();
_caseBase.learnCases(casesToRetain);
```

This code code shows the the window illustrated in Figure 11.

With this retain step we have completed the code of the cycle() method of the Travel Recommender application.
17 Shutting down a CBR application

The `postcycle()` method of the CBR applications in jCOLIBRI2 is in charge of shutting down the CBR system or invoke maintenance tasks. This method usually calls the `close()` method of the connector to store (if required) the learned cases into the persistence media.

In the Travel Recommender application we call the `close()` method of the connector and also the `shutdown()` method of the database manager.

```
Listing 34: TravelRecommender.postCycle() code

    public void postCycle() throws ExecutionException {
        _connector.close();
        jcolibri.test.database.HSQLDBserver.shutDown();
    }
```

This method completes the tutorial. Now the Travel Recommender code should compile and work properly. If you find some problems read the complete source code of the located into the “example” subfolder of the framework and compressed as `travelrecommender-source.zip`. 
18 Textual CBR

Textual Case-Based Reasoning (TCBR) is a subfield of CBR the cases are available in textual format. This type of CBR systems is very interesting because in most of the domains where CBR can be applied the available experiences are in textual format. Some examples are: laws, medicine, help-desk, etc.

There does not appear to be a standard or consensus about the structure of a textual CBR system. This is mainly due to the different knowledge requirements in application domains. For classification applications typically only a basic stemmer algorithm and a cosine similarity function is needed, while with other applications more intense NLP derived structures are employed (see [9] and [10]). Although a common functionality for TCBR systems is difficult to establish, several researchers have attempted to define the different knowledge requirements for TCBR([24],[55]).

To support the development of TCBR systems, jCOLIBRI2 includes a complete extension with useful methods for this kind of applications. As adaptation continues being a domain specific task, the framework only includes methods for the retrieval step.

There are two big groups of retrieval algorithms that can be used in TCBR:

- The first group is based on IE methods to capture features from the text and then perform standard Nearest Neighbor similarity based computation over those features. With these methods developers extract the information contained in the texts and represent them in a structured way that allows to apply the typical retrieval and reuse methods in common CBR applications.

- The second one is composed by the broadly applied IR algorithms used by search engines and based in the Vector Space Model. These methods can achieve very good results in the retrieval step, but make difficult the adaptation because there is not a structure representation of the cases.

We could refer to the first group as “semantic” retrieval because it tries to capture the semantics of the texts and the second as “statistical” retrieval because it only takes into account the frequency of terms.

18.1 Semantic retrieval

The first set of textual methods included in jCOLIBRI2 belong to the semantic retrieval group, and are based in the Lenz layered model [33]. This model divides the text processing into several stages:
1. **Keyword Layer.** This layer separates texts into terms, removes stop-words, stem terms and calculates statistics about frequency of terms. It also proposes a part-of-speech tagger in this layer that could be useful by the following ones. This layer is domain-independent, so it can be shared between applications.

2. **Phrase Layer.** Recognises domain-specific phrases using a dictionary. Here, the problems are that some parts of the phrase can be separated and that the dictionary must be built manually.

3. **Thesaurus Layer.** This layer identifies synonyms and related terms. Methods implemented in this layer must be reusable in the query stage of the CBR cycle. WordNet can be used as an english thesaurus. This phase is domain-independent.

4. **Glossary Layer.** Is the domain-specific version of the thesaurus layer. So it is desirable to define a common interface for both layers. The main difficulty with this layer resides in the glossary acquisition.

5. **Feature Value Layer.** With semi-structured cases, this layer extracts features about the case and stores it as <attribute,value> pairs in the case representation. It is also domain-specific.

6. **Domain Structure Layer.** Uses the previous layer to classify documents in a high level. It assigns "topic" features to the cases that can be useful in the indexing process.

7. **Information Extraction Layer.** Some parts of the texts can be better represented with a structured approximation. This layer accomplish this task. (note that this functionality can overlap with the two previous layers).

The last IE layer applies user defined rules to extract the features using the information obtained in previous layers. This way, developers obtain a structured representation of cases that can be managed by standard similarity matching techniques from CBR.

### 18.1.1 Representation of the texts

jCOLIBRI2 has the generic `jcolibri.datatypes.Text` object to store texts into cases. These objects are managed by the methods of the `jcolibri.extensions.textual` package.

This object is not enough to manage the required information defined by the Lenz steps. So, there is a subclass named `jcolibri.extensions.textual.IE.representation.IEText` (texts for Information Extraction) that allows it.

An IE text receives its content as String and later a method will organize this content. This way, a text is composed by paragraphs, paragraphs by sentences and sentences by tokens as shown in Figure 25

Tokens represent a word in the text. These objects store information like:
• If the token is a stop word (word without sense).
• If the token is a main name inside the sentence.
• The stemmed word.
• The Part-Of-Speech tag of the token.
• A list of relations with other similar tokens.

The organization in paragraphs, sentences and tokens is performed by specific methods depending on the chosen implementation.

The information extracted from the text is stored in the IEtext object. There are several types of information that will be obtained by dedicated methods:

• Phrases identified in the text.
• Features: identifier-value pairs extracted from the text.
• Topics: combining phrases and features a topic can be associated to a text. A topic is a classification of the text.

Phrases and Features are stored using the objects implemented in the jcolibri.extensions.textual.IE.representation.info sub-package. That package stores three objects that aid in the representation of the extracted information:

• PhraseInfo: stores extracted phrases.
• FeatureInfo: stores extracted features.
• WeightedRelation: represents a weighted relation between two tokens. These relations are found by the glossary and thesaurus methods.

Figure 26 illustrates the complete organization:
18.1 Semantic retrieval

18.1.2 IE methods implementation

jCOLIBRI includes several implementations of the Lenz layers. Some methods have been implemented in a general way. Other methods use the Maximum Entropy algorithms implemented in the OpenNLP package. And finally, another group of methods use the GATE library for text processing.

The OpenNLP\(^6\) library includes methods that use Maximum Entropy algorithms that have been implemented and applied to Natural Language tasks. OpenNLP is divided into independent layers that can be used separately, providing a stop-word remover, sentence detector, part-of-speech tagger, grammar layer, ...

GATE\(^7\) is an infrastructure for developing software components that process human language. This library is broadly used for IE applications and includes many components that are used in jCOLIBRI2 to implement the Lenz layers: POS tagger, stemmer, features and phrases extractors, etc.

Each group of methods can only work with certain textual objects. The OpenNLP implementation has its own specialization of IEText named IETextOpenNLP, and the GATE implementation has the IETextGate object:

Following table summarizes the jCOLIBRI2 methods for TCBR using Information Extraction:

---

\(^6\)http://opennlp.sourceforge.net

\(^7\)http://gate.ac.uk
18.1 Semantic retrieval

<table>
<thead>
<tr>
<th>Implementation</th>
<th>OpenNLP</th>
<th>GATE</th>
<th>Generic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Textual Object</td>
<td>IETextOpenNLP</td>
<td>IETextGate</td>
<td>IETextOpenNLP, IETextGate, IEText</td>
</tr>
<tr>
<td>Package</td>
<td>IE.opennlp</td>
<td>IE.gate</td>
<td>IE.common</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layers</th>
<th>OpenNLP</th>
<th>GATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organize Text</td>
<td>OpenNlpSplitter</td>
<td>GateSplitter</td>
</tr>
<tr>
<td>Keyword: StopWords</td>
<td></td>
<td>StopWordsDetector</td>
</tr>
<tr>
<td>Keyword: Stemmer</td>
<td></td>
<td>TextStemmer</td>
</tr>
<tr>
<td>Keyword: POS tagging</td>
<td>OpenNlpPOSStagger</td>
<td>GatePOSStagger</td>
</tr>
<tr>
<td>Keyword: Main Names</td>
<td>OpenNlpMainNamesExtractor</td>
<td>PhrasesExtractor</td>
</tr>
<tr>
<td>Phrase</td>
<td></td>
<td>GlossaryLinker</td>
</tr>
<tr>
<td>Glossary</td>
<td></td>
<td>ThesaurusLinker</td>
</tr>
<tr>
<td>Thesaurus Value</td>
<td></td>
<td>FeaturesExtractor</td>
</tr>
<tr>
<td>Domain Structure</td>
<td>GateFeaturesExtractor</td>
<td>DomainTopicClassifier</td>
</tr>
<tr>
<td>Information Extraction</td>
<td></td>
<td>BasicInformationExtractor</td>
</tr>
</tbody>
</table>

18.1.3 Computing similarity

The methods of the IE extension extract information from texts and store it into the other attributes of the case. These attributes can be compared using normal similarity functions as shown in Figure 27

![Figure 27: Common organization of textual cases](image)

Textual attributes can also be compared using specific similarity functions located in the package `jcolibri.method.retrieve.NNretrieval.similarity.local.textual`. Most of them are applicable to any Text attribute, but some few are only applicable to IEText objects (or its subclasses) because require information stored in the tokens.

**Cosine**

\[
\text{cosine}(t_1, t_2) = \frac{|t_1 \cap t_2|}{\sqrt{|t_1| \cdot |t_2|}}
\]
Dice (similarity.local.textual.DiceCoefficient)

\[ \text{dice}(t_1, t_2) = \frac{2 \times |\{t_1 \cap t_2\}|}{|t_1| + |t_2|} \]

Jaccard (similarity.local.textual.JaccardCoefficient)

\[ \text{jaccard}(t_1, t_2) = \frac{|\{t_1 \cap t_2\}|}{|t_1| \cup |t_2|} \]

Overlap (similarity.local.textual.OverlapCoefficient)

\[ \text{overlap}(t_1, t_2) = \frac{|\{t_1 \cap t_2\}|}{\min(|t_1|, |t_2|)} \]

Compression (similarity.local.textual.compressionbased.CompressionBased)

\[ \text{CDM}(x, y) = \frac{C(xy)}{C(x) + C(y)} \]

where \( C(x) \) is the size of string \( x \) after compression (and \( C(y) \) similarly) and \( C(xy) \) is the size, after compression, of the string that comprises \( y \) concatenated to the end of \( x \).

Developed and implemented by Derek Bridge. See following papers [30, 14] and framework documentation.

Normalised Compression (similarity.local.textual.compressionbased.NormalisedCompression)

\[ \text{NCD}(x, y) = \frac{C(xy) - \min(C(X), C(y))}{\max(C(x), C(y))} \]

where \( C(x) \) is the size of string \( x \) after compression (and \( C(y) \) similarly) and \( C(xy) \) is the size, after compression, of the string that comprises \( y \) concatenated to the end of \( x \).

Developed and implemented by Derek Bridge. See following papers [34, 15] and framework documentation.

18.1.4 The Restaurant Recommender example

To illustrate the textual methods of jCOLIBRI2, we have developed a restaurant adviser system. The entire case base contains roughly 100 different restaurants extracted from an online magazine. This recommender is implemented using both the semantic and statistical methods. The complete implementation of the Restaurant Recommender application can be found in the Tests 13a and 13b of the framework examples.
Following listings contains the most important code of the Restaurant Recommender implementation that uses the semantic textual methods of jCOLIBRI2. In this case we are using the OpenNLP version of the methods.

In the precycle, the recommender application executes the textual methods over the complete case base. It illustrates the advantages of having a precycle in the CBR system because this computation is an very takes a long time but is only executed once.

**Listing 35: The Restaurant Recommender precycle using semantic TCBR methods**

```java
public CBRCaseBase preCycle() throws ExecutionException {
    // In the precycle we pre-compute the information extraction in the case base

    // Initialize Wordnet
    ThesaurusLinker.loadWordNet();
    // Load user-specific glossary
    GlossaryLinker.loadGlossary("jcolibri/test/test13/glossary.txt");
    // Load phrases rules
    PhrasesExtractor.loadRules("jcolibri/test/test13/phrasesRules.txt");
    // Load features rules
    FeaturesExtractor.loadRules("jcolibri/test/test13/featureRules.txt");
    // Load topic rules
    DomainTopicClassifier.loadRules("jcolibri/test/test13/domainRules.txt");

    // Obtain cases
    _caseBase.init(_connector);
    Collection<CBRCase> cases = _caseBase.getCases();

    // Perform IE methods in the cases

    // Organize cases into paragraphs, sentences and tokens
    OpenNlpSplitter.split(cases);
    // Detect stopwords
    StopWordsDetector.detectStopWords(cases);
    // Stem text
    TextStemmer.stem(cases);
    // Perform POS tagging
    OpenNlpPOStagger.tag(cases);
    // Extract main names
    OpenNlpMainNamesExtractor.extractMainNames(cases);
    // Extract phrases
    PhrasesExtractor.extractPhrases(cases);
    // Extract features
    FeaturesExtractor.extractFeatures(cases);
}
```
In the cycle(), the application only has to execute the TCBR methods over the query. However, there are some methods that relate query and case terms that can only be applied in this cycle instead of the precycle.

The semantic TCBR methods extract the information from the text into the case (or query). This way, we can compute a typical k-NN similarity measure to obtain the most suitable restaurant for the query. Following listing shows the code:

Listing 36: The Restaurant Recommender cycle using semantic TCBR methods

```java
public void cycle(CBRQuery query) throws ExecutionException {
    Collection<CBRCase> cases = _caseBase.getCases();

    // Perform IE methods in the cases
    OpenNLPSplitter.split(query);
    StopWordsDetector.detectStopWords(query);
    TextStemmer.stem(query);
    OpenNLPPOSTagger.tag(query);
    OpenNLPMainNamesExtractor.extractMainNames(query);

    // Now that we have the query we relate cases tokens with the query tokens
    GlossaryLinker.LinkWithGlossary(cases, query);
    ThesaurusLinker.linkWithWordNet(cases, query);

    // Extract phrases
    PhrasesExtractor.extractPhrases(query);

    // Extract features
    FeaturesExtractor.extractFeatures(query);

    // Classify with a topic
    DomainTopicClassifier.classifyWithTopic(query);
}
```
// Perform IE copying extracted features or phrases into other attributes of the query
BasicInformationExtractor.extractInformation(query);

// Now we configure the k-NN retrieval with some user-defined similarity measures
NNConfig nnConfig = new NNConfig();
nnConfig.setDescriptionSimFunction(new Average());

nnConfig.addMapping(new Attribute("location", RestaurantDescription.class), new Equal());

// To compare text we use the OverlapCoefficient
nnConfig.addMapping(new Attribute("description", RestaurantDescription.class), new OverlapCoefficient());

// This function takes a string with several numerical values and computes the average
nnConfig.addMapping(new Attribute("price", RestaurantDescription.class), new AverageMultipleTextValues(1000));

// This function takes a string with several words separated by whitespaces, converts it to a set of tokens and computes the size of the intersection of the query set and the case set normalized with the case set
nnConfig.addMapping(new Attribute("foodType", RestaurantDescription.class), new TokensContained());
nnConfig.addMapping(new Attribute("food", RestaurantDescription.class), new TokensContained());
nnConfig.addMapping(new Attribute("alcohol", RestaurantDescription.class), new Equal());
nnConfig.addMapping(new Attribute("takeout", RestaurantDescription.class), new Equal());
nnConfig.addMapping(new Attribute("delivery", RestaurantDescription.class), new Equal());
nnConfig.addMapping(new Attribute("parking", RestaurantDescription.class), new Equal());
nnConfig.addMapping(new Attribute("catering", RestaurantDescription.class), new Equal());

Collection<RetrofitResult> res = NNScoringMethod.evaluateSimilarity(cases, query, nnConfig);
res = SelectCases.selectTopKRR(res, 5);

// Show the result
...

}
18.2 Statistical retrieval

The second set of textual methods, added to jCOLIBRI2, belong to the statistical approach that has given such good results in the IR field. jCOLIBRI2 methods are based on the Apache Lucene search engine [25].

Lucene uses a combination of the Vector Space Model (VSM) of IR and the Boolean model to determine how relevant a given document is to a user’s query. In general, the idea behind the VSM is the more times a query term appears in a document relative to the number of times the term appears in all the documents in the collection, the more relevant that document is to the query. It uses the Boolean model to first narrow down the documents that need to be scored based on the use of boolean logic in the query specification. Lucene also adds some capabilities and refinements onto this model to support boolean and fuzzy searching, but it essentially remains a VSM based system at its heart.

The main advantages of this search method is the good results and the applicability to non-structured texts. The big drawback is the lack of knowledge about the semantics of the texts that complicates the adaptation stage.

The Lucene retrieval method of jCOLIBRI2 is jcolibri.method.retrieve.LuceneRetrievial. This function can be applied to any Text subclass as it does not require any kind of extracted information. This method retrieves cases computing only the similarity of a given attribute. This way, the other attributes of the case are not used.

To compute a complete k-NN retrieval using all the attributes of the case, there is a similarity function that uses Lucene to compare textual attributes: jcolibri.method.retrieve.NNretrieval.similarity.local.textual.LuceneTextSimilarity.

These Lucene methods require an index to be created in the pre-cycle by the method jcolibri.method.precycle.LuceneIndexCreator.

Although statistical IR methods give good retrieval results they do not provide any kind of explanation about the returned documents. One way for solving this problem is to cluster the retrieval results into groups of documents with common information. Usually, clustering algorithms like hierarchical clustering or K-means [57] group the documents but they don’t provide a comprehensive description of the resulting clusters. Lingo [39] is a clustering algorithm implemented in the Carrot2 framework that allows the grouping of search results but also gives the user a brief textual description of each cluster. Lingo is based on the Vector Space Model, Latent Semantic Indexing and Singular Value Decomposition to ensure that there are human readable descriptions of the clusters and then assign documents to each one. jCOLIBRI2 provides wrapper methods to hide the complexity of the algorithm and allows a simple way for managing Carrot2.

8http://www.carrot2.org
Read the documentation of the `jcolibri.extensions.textual.carrot2` package for details.

This labeled-clustering algorithm could be applied to TCBR in the retrieval step to make it easier to choose the most similar document. Moreover, in [42] we present an alternative approach that uses the labels of the clusters to guide the adaptation of the texts.

### 18.2.1 The Restaurant Recommender using statistical methods

Test 13b shows how to implement the Restaurant Recommender application using statistical methods. Following listing shows an excerpt of the code:

```java
public CBRCaseBase preCycle() throws ExecutionException
{
    _caseBase.init(_connector);

    // Here we create the Lucene index
   .luceneIndex = jcolibri.method.precycle.LuceneIndexCreator.
    createLuceneIndex(_caseBase);

    return _caseBase;
}

public void cycle(CBRQuery query) throws ExecutionException
{
    Collection<CBRCase> cases = _caseBase.getCases();

    NNConfig nnConfig = new NNConfig();
    nnConfig.setDescriptionSimFunction(new Average());

    // We only compare the "description" attribute using Lucene
    Attribute texturalAttribute = new Attribute("description",
        RestaurantDescription.class);
    nnConfig.addMapping(texturalAttribute, new
        LuceneTextSimilarity(luceneIndex, query, texturalAttribute,
        true));

    Collection<RetrievalResult> res = NNScoringMethod.
    evaluateSimilarity(cases, query, nnConfig);
    res = SelectCases.selectTopKRR(res, 5);

    // Show the result

    ...
```
19 Evaluation of CBR applications

jCOLIBRI2 includes an extension for evaluating CBR applications. This functionality is found in the jcolibri.evaluation package and is composed by the following components:

- **jcolibri.evaluation.Evaluator**: This abstract class defines the common behavior of an evaluator. It is initialized with the CBRAplication to be evaluated and allows to access the EvaluationReport object that contains the result of the evaluation.

  This abstract class is extended by the following evaluators:

  - **jcolibri.evaluation.evaluators.LeaveOneOutEvaluator**: This method uses all the cases as queries. It executes so many cycles as cases in the case base. In each cycle one case is used as query.

  - **jcolibri.evaluation.evaluators.HoldOutEvaluator**: This method splits the case base in two sets: one used for testing where each case is used as query, and another that acts as normal case base. This process is performed several times.

  - **jcolibri.evaluation.evaluators.NFoldEvaluator**: This evaluation method divides the case base into several random folds (indicated by the user). For each fold, their cases are used as queries and the remaining folds are used together as case base. This process is performed several times.

  - **jcolibri.evaluation.evaluators.SameSplitEvaluator**: This method splits the case base in two sets: one used for testing where each case is used as query, and another that acts as normal case base. This method is different of the other evaluators because the split is stored in a file that can be used in following evaluations. This way, the same set is used as queries for each evaluation.

Developers can extend the Evaluator class to create their own evaluators.

- **jcolibri.evaluation.EvaluationReport**: This class stores the result of an evaluation. It is configured and filled by an Evaluator. This info is also used to represent graphically the result of an evaluation. The stored information can be:
  - Several series of data. The length of the series is the number of executed cycles. And several series can be stored using different labels.
  - Any other information. The put/getOtherData() methods allow to store any other kind of data.
  - Number of cycles.
  - Total evaluation time.
  - Time per cycle.
**jcolibri.evaluation.tools.EvaluationResultGUI** - Class that visualizes the result of an evaluation in a Swing frame. It generates a chart with the evaluation result and other information returned by the evaluator.

The evaluation of a CBR application is shown by the test 8 of the examples. The following code is an excerpt of the code:

```java
Listing 38: Evaluation code

... HoldOutEvaluator eval = new HoldOutEvaluator();
    eval.init(new EvaluableApp());
    eval.HoldOut(5, 1);
    System.out.println(Evaluator.getEvaluationReport());
    jcolibri.evaluation.tools.EvaluationResultGUI.show(Evaluator.getEvaluationReport(), "Test8 Evaluation", false);
... 
```

As the listing shows, the `HoldOutEvaluator` is initialized with a common `CBRApplication` (`EvaluableApp`) through the `init()` method inherited from the `Evaluator` class. Then, the evaluator is invoked with the corresponding parameters. Finally, the `EvaluationResultGUI` class visualizes the result as shown in 28.

![Figure 28: Evaluation of a CBR application](image)

The application to be evaluated is in charge of storing the data in the `EvaluationReport`. Each time that the `cycle()` method of the evaluated application is invoked, this method must store the evaluation of that cycle into the report.
For example, in the Test 8, we are evaluating the similarity value of the most similar case:

**Listing 39: Example of an evaluable application**

```java
public class EvaluableApp implements StandardCBRApplication {
    public void cycle(CBRQuery query) throws ExecutionException {
        NNConfig simConfig = new NNConfig();
        ...

        Collection<RetrievalResult> eval = NNScoringMethod.
        evaluateSimilarity(_caseBase.getCases(), query, simConfig);

        // Now we add the similarity of the most similar case
        // to the serie "Similarity".
        Evaluator.getEvaluationReport().addDataToSeries("Similarity",
                new Double(eval.iterator().next().getEval()));
    }
}
```

Group for Artificial Intelligence Applications  jCOLIBRI2 Tutorial
20 Recommenders

jCOLIBRI 2 includes an extension to implement recommendation systems. This extension has been designed having in mind the future design process of CBR systems in jCOLIBRI.

We propose a flexible way to design CBR systems in future versions of jCOLIBRI using a library of templates obtained from a previously designed set of CBR systems. In case-based fashion, jCOLIBRI will retrieve templates from a library of templates (i.e. a case base of CBR design experience); the designer will choose one, and adapt it.

We represent templates graphically as shown in following Figures 29 and 30. Each rectangle in the template is a subtask. Simple tasks (shown as blue or pale grey rectangles) can be solved directly by a method included in this extension. Complex tasks (shown as red or dark grey rectangles) are solved by decomposition methods having other associated templates. There may be multiple alternative methods to solve any given task. These methods are usual java methods of the classes in the framework.

Before implementing this extension, we developed templates for recommender systems and then generated the methods that solve every task. That allowed us to develop many recommender systems that are included in the jcolibri.test.recommenders package. By now, templates are only a graphical representation of CBR systems, although in a short future they will be used to generate them.

The jCOLIBRI team thanks to Derek Bridge his collaboration and supervision during the development of this extension.

20.1 Templates guided design of recommendation systems

This section introduces the templates guided design of recommendation systems. Following text is an extract of [40].

In [40] we have done an analysis of recommender systems that is based in part on the conceptual framework described in the review paper by Bridge et al. [8].

The framework distinguishes between collaborative and case-based, reactive and proactive, single-shot and conversational, and asking and proposing. Within this framework, the authors review a selection of papers from the case-based recommender systems literature, covering the development of these systems over the last ten years. We take the systems’ interaction behaviour as the fundamental distinction from which we construct recommenders:

- Single-Shot Systems make a suggestion and finish. Figure 29 shows the template for this kind of system, where One-Off Preference Elicitation and Retrieval are complex tasks that are solved by decomposition methods having other associated templates.
20.1 Templates guided design of recommendation systems

After retrieving items, Conversational Systems (Figure 30) may invite or allow the user to refine his/her current preferences, typically based on the recommended items. Iterated Preference Elicitation might be done by allowing the user to select and critique a recommended item thereby producing a modified query, which requires that one or more retrieved items be displayed (Figure 30 left). Alternatively, it might be done by asking the user a further question or questions thereby refining the query, in which case the retrieved items might be displayed every time (Figure 30 left) or might be displayed only when some criterion is satisfied (e.g. when the size of the set is “small enough”) (Figure 30 right). Note that both templates share the One-Off Preference Elicitation and Retrieval tasks with single-shot systems.

20.1.1 One-Off Preference Elicitation

We can identify three templates by which the user’s initial preferences may be elicited:

- One possibility is Profile Identification where the user identifies him/herself, e.g. by logging in, enabling retrieval of a user profile from a profile database. This profile might be a content-based profile (e.g. keywords describing the user’s long-term interests, or descriptions of items consumed previously by this user, or descriptions of sessions this user has engaged in previously with this system); or it might be a collaborative filtering style of profile (e.g. the user’s ratings for items). The template allows for the possibility that the user is a new user, in which case there may be some registration process followed by a complex task that elicits an initial profile.

- An alternative is Initial Query Elicitation. This is itself a complex task with multiple alternative decompositions. The decompositions include: Form-Filling (Figure 31 left) and Navigation-by-Asking (i.e. choosing and asking a question) (Fig-
Figure 30: Conversational templates

Figure 31: Preference Elicitation decompositions

Various versions of the Entrée system [11] offered interesting further methods: Identify an Item (where the user gives enough information to identify an item that s/he likes, e.g. a restaurant in his/her home town, whose description forms the basis of a query, e.g. for a restaurant in the town being visited); and Select an Exemplar (where a small set of contrasting items is selected and displayed, the user chooses the one most similar to what s/he is seeking, and its description forms the basis of a query).

- The third possibility is Profile Identification & Query Elicitation, in which the previous two tasks are combined.
20.1.2 Retrieval

Because we are focussing on case-based recommender systems (and related memory-based recommenders including collaborative filters), Retrieval is common to all our recommender systems. Retrieval is a complex task, with many alternative decompositions. The choice of decomposition is, of course, not independent of the choice of decomposition for One-Off Preference Elicitation and Iterated Preference Elicitation. For example, if One-Off Preference Elicitation delivers a ratings profile, then the method chosen for achieving the Retrieval task must be some form of collaborative recommendation.

The following is a non-exhaustive list of papers that define methods that can achieve the Retrieval task: Wilke et al. 1998 [56] (similarity-based retrieval using a query of preferred values); Smyth & McClave 2001 [53] (diversity-enhanced similarity-based retrieval); McSherry 2002 [36] (diversity-conscious retrieval); Bridge & Ferguson 2002 [7] (order-based retrieval); McSherry 2003 [37] (compromise-driven retrieval); Bradley & Smyth 2003 [6] (where user profiles are mined and used); Herlocker et al. 1999 [26] (user-based collaborative filtering); Sarwar et al. 2001 [47] (item-based collaborative filtering). In all these ways of achieving Retrieval, a scoring process is followed by a selection process. For example, in similarity-based retrieval (k-NN), items are scored by their similarity to the user’s preferences and then the k highest-scoring items are selected for display; in diversity-enhanced similarity-based retrieval, items are scored in the same way and then a diverse set is selected from the highest-scoring items; and so on. Note also that there are alternative decompositions of the Retrieval task that would not have this two-step character. For example, filter-based retrieval, where the user’s preferences are treated as hard constraints, conventionally does not decompose into two such steps. On the other hand, there are recommender systems in which Retrieval decomposes into more than two steps. For example, in some forms of Navigation-by-Proposing (see below), first a set of items that satisfy the user’s critique is obtained by filter-based retrieval, then these are scored for similarity to the user’s selected item, and finally a subset is chosen for display to the user.

20.1.3 Iterated Preference Elicitation

In Iterated Preference Elicitation the user, who may or may not have just been shown some products (Figure 30), may, either voluntarily or at the system’s behest, provide further information about his/her preferences. Alternative decompositions of this task include:

- Form-Filling where the user enters values into a form that usually has the same structure as items in the database (Figure 31 left). We have seen that Form-Filling is also used for One-Off Preference Elicitation. When it is used in Iterated Preference Elicitation, it is most likely that the user edits values s/he previously entered into the form.

- Navigation-by-Asking is another method that can be used for both One-Off Preference Elicitation and for Iterated Preference Elicitation. The system refines the
20.2 Methods for recommender systems

This section describes the new methods included in jCOLIBRI 2.1. There are several examples that illustrate how to use these methods in the jcolibri.test.recommenders package (read the documentation of the package for details).

20.2.1 User interaction methods

**ObtainQueryMethod**: Obtains the query showing a form. There are two different versions: showing initial values (either default or from a previous iteration) and showing empty form. Optional parameters (both defined by the designer):

- **Hidden Attributes**: These attributes are not shown to the user. This is useful when using the McSherry’s similarity functions [37] that don’t take into...
account the user preferences (see following subsection for details).

**Attributes labels**. Labels to be shown when asking for an attribute. Is useful when using filter based similarity because instead asking for the *price* the designer may ask for the *maximum price*.

**DisplayCasesMethod** : Shows the case in a panel. Useful when working with composed cases. This method returns a `jcolibri.method.gui.casesDisplay.UserChoice` object that encapsulates three different user answers: Quit, Buy and Refine Query.

The Refine Query result is used in conversational systems that redefine the query (usually in Navigation by Proposing). It is completely useless in one-shot systems. So this method has three different display options:

- In BASIC mode the dialog only shows the buy and quit buttons.
- The REFINE_QUERY option returns a `UserChoice.REFINE_QUERY` value.
- The SELECT_CASE option returns a `UserChoice.REFINE_QUERY` value together with a case selected from the list.

**DisplayCasesInTableMethod** : Similar to the previous one. This method shows the cases in a table. This way is not very suitable for composed cases.

**UserChoice** : Object that encapsulates the user answer when cases are shown. This object keeps an internal integer with possible values: QUIT, BUY and REFINE_QUERY.

It also contains the chosen case from the list. If the answer is BUY, the selected case is the final result. If the answer is REFINE_QUERY, the selected case can be used in Navigation by Proposing to elicit the query. The subclass CriticalUserChoice is an extension that also contains the critiques to the chosen case.

### 20.2.2 New Nearest Neighbor similarity measures

In following formulas `c.a` represents an attribute of the case and `q.a` an attribute of the query.

**Inreca Less is Better** :

\[
\text{sim}(c.a, q.a) = \begin{cases} 
1 & \text{if } (c.a < q.a) \\
\text{jump } \ast & \left(\frac{\max(a) - c.a}{\max(a) - q.a}\right) 
\end{cases}
\]

**Inreca More is Better** :

\[
\text{sim}(c.a, q.a) = \begin{cases} 
1 & \text{if } (c.a > q.a) \\
\text{jump } \ast & \left(1 - \frac{q.a - c.a}{q.a}\right) 
\end{cases}
\]

**McSherry Less is Better** : (note that the user query value is ignored)

\[
\text{sim}(c.a, q.a) = \frac{\max(a) - c.a}{\max(a) - \min(a)}
\]
20.2 Methods for recommender systems

**McSherry More is Better** : (note that the user query value is ignored)

\[ sim(c.a, q.a) = 1 - \frac{(\max(a) - c.a)}{(\max(a) - \min(a))} \]

**Table** : reads a table data from a csv file. Axes values must be strings or enumerations.

### 20.2.3 Conditional methods

**Continue** : Receives an *UserChoice* and returns true if the value is *Edit Query*.

**BuyOrQuit** : Receives an *UserChoice* object and returns true or false depending on its value (*Quit or Buy*).

**DisplayCasesIfNumber** : Returns true if the number of cases received is inside a range. Optionally this method can show a message. Useful in conversational B systems when it is used with FilterBased retrieval (with k-NN it has no sense).

**DisplayCasesIfSimil** : Returns true if the retrieved cases have a minimum similarity. Useful only with k-NN.

### 20.2.4 Navigation by Asking methods

**ObtainQueryWithAttributeQuestionMethod** Asks the user for the value of an attribute. This method is used in Navigation by asking. It only shows the available options, removing the values that don’t appear in the working cases set. Customs labels are allowed.

**Information Gain** Returns the attribute with more information gain in a set of cases [3][50]. Used in Navigation by Asking with ObtainQueryWithAttributeQuestionMethod.

\[
Gain(A) = - \sum_{j=1}^{m} \frac{|C^j|}{|C|} \cdot \log_2 \frac{|C^j|}{|C|}
\]

where \( C \) is partitioned according to the attribute \( A \) into \( m \) subsets \( C = C^1 \cup \cdots \cup C^m \) such that the attribute value of all cases in \( C^j \) is \( v_j \).

**Similarity Influence Measure** Selects the attribute that has the highest influence on the k-NN similarity. The influence on the similarity can be measured by the expected variance of the similarities of a set of selected cases. See [3][31][49] for details.

\[
SimVar(q, A, C) = \sum_v p_v \cdot Var(q_{A=v}, c)
\]

where:

- \( v \) are the possible values of the attribute \( A \).
- \( q_{A=v} \) is the query once assigned \( v \) to the attribute \( A \).
- \( p_v = |C^v| / |C| \).
• \( Var(q, C) = \frac{1}{|C|} \cdot \sum_{c \in C} (\text{sim}(q, c) - \mu)^2 \), where \( \mu \) is the average similarity for the cases in \( C \).

When using Attribute Selection and Filter based retrieval, the numerical attributes must be transformed into ranges, and the Filter based retrieval only uses EqualTo() predicates.

### 20.2.5 Navigation by Proposing

**DisplayCasesTableWithCritiquesMethod** : Displays the cases in a table allowing the user to buy one item, finish, or critique the selected item. Critiques are configured by the designer providing a list of **CritiqueOptions**. This method returns a **CriticalUserChoice** that is an extension of **UserChoice** storing also the critiques of the selected item. This method enables or disables the critiques depending on the values of the working cases. For example, it has no sense to show a “cheaper” button if there are not cheaper cases. Usually, displayed cases are the same than working cases, but when using diversity algorithms only three of the working cases are displayed.

**CritiqueOption** : Object that encapsulates the possible critiques of a case. It stores the label of the button, the criticized attribute, and a Filter-Based Retrieval predicate to perform the critique.

**CriticalUserChoice** : Extension of **UserChoice** that also stores the critiques over the selected case.

In the Iterated preference elicitation of Navigation by Proposing there are several methods to modify the query (see [35]:

- **More Like This** replaces current query with the description of the selected case.
- **Partial More Like This** partially replaces current query with the description of the selected case. It only transfers a feature value from the selected case if none of the rejected cases have the same feature value.
- **Weighted More Like This** transfers all attributes from the selected case to the query but weights them given preference to diverse attributes among the proposed cases. The new weights are stored into a **NNConfig** object, so this strategy should be used with NN retrieval.
- **Less Like This** is a simple one: if all the rejected cases have the same feature-value combination, which is different from the preferred case then this combination can be added as a negative condition. This negative condition is coded as a **NotEqualTo(value)** predicate in a **FilterConfig** object. The query is not modified. That way, this method should be used together with **FilterBasedRetrieval**.
- **More + Less Like This** combines both More Like This and Less Like This. It copies the values of the selected case into the query and returns a **FilterConfig**
object with the negative conditions. This method should be used together with Filtered and NN retrieval.

### 20.2.6 Profile management methods

**CreateProfile** : Obtains a user profile (query) using the FormFilling method and stores it in a xml file. This method is not part of the CBR cycle. It is executed in the PreCycle or in a separate application.

**ObtainQueryFromProfile** : Obtains the query form the xml file generated by CreateProfile.

### 20.2.7 Collaborative Recommendations

These kind of recommendations are based on past experiences of other users. They need a specific case base implementation that manages cases composed by a description, a solution and a result. The description usually contains the information about the user that made the recommendation, the solution contains the information about the recommended item, and the result stores the rating value for the item (the value that the user assigns to an item after evaluating it). This way, this case base implementation stores cases as a table:

<table>
<thead>
<tr>
<th>User1</th>
<th>Item1</th>
<th>Item2</th>
<th>Item3</th>
<th>Item4</th>
<th>Item5</th>
<th>...</th>
<th>ItemM</th>
</tr>
</thead>
<tbody>
<tr>
<td>User2</td>
<td>rating12</td>
<td></td>
<td>rating14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User3</td>
<td>rating21</td>
<td>rating23</td>
<td></td>
<td></td>
<td>rating2N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>rating33</td>
<td>rating34</td>
<td></td>
<td>rating3N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UserN</td>
<td>ratingN2</td>
<td></td>
<td>ratingN5</td>
<td></td>
<td>ratingNN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The values of the first column and row contain the ids of the description and solution components of the case. These ids must be integer values. The ratings are obtained from an attribute of the result component.

Note that these cases base allows to have different cases with the same description (because each user can make several recommendations).

The behavior of collaborative recommenders can be split into three steps [26]:

1. Weight all users with respect to similarity with the active user.
2. Select a subset of users as a set of predictors.
3. Normalize ratings and compute a prediction from a weighted combination of selected neighbors’ ratings.

The case base implementation of the collaborative package performs the first step. The other two final steps are performed by the collaborative retrieval method.

See [29] for further details.
**MatrixCaseBase** : Specific implementation of CBRCaseBase to allow collaborative recommendations. As there are several ways to compute the first step of the behavior of collaborative recommenders. This class provides most of the code, but it must be specialized as in PearsonMatrixCaseBase. The subclasses must implement the abstract methods defined here. These methods return the similarity among neighbors. This similarity value is stored into SimilarityTuple objects.

Ratings must be sorted by neighbors to allow an efficient comparison. Looking to the previous table this means that cases are organized by rows. This process is performed internally in this class.

There are also two internal classes (CommonRatingTuple and CommonRatingsIterator) that allow to efficiently obtain the common ratings of two users. This will be used by subclasses when computing the neighbors similarity. The code of these classes is an adaptation of the one developed by Jerome Kelleher and Derek Bridge for the Collaborative Movie Recommender project at University College Cork (Ireland).

**PearsonMatrixCaseBase** : Extension of the MatrixCaseBase that computes similarities among neighbors using the Pearson Correlation:

\[
\text{sim}(a, u) = \frac{\sum_{i=1}^{m} (r_{a,i} - \bar{r}_a)(r_{u,i} - \bar{r}_u)}{\sigma_a\sigma_b} \times \frac{s}{f}
\]

where: \(a\) and \(u\) are the compared neighbors, \(m\) the number of common items. \(\bar{r}\) denotes a mean value and \(\sigma\) denotes a standard derivation, and these are computed on co-rated items only. The Pearson correlation is weighted by a factor \(\frac{s}{f}\) where \(s\) is the number of co-rated items and \(f\) is defined by the designer. This decreases the similarity between users who have fewer than \(f\) co-rated items.

**CollaborativeRetrievalMethod** : This method returns cases depending on the recommendations of other users. It uses a PearsonMatrix case base to compute the similarity among neighbors. Then, cases are scored according to a rating that is estimated using the following formula:

\[
p_{a,i} = \bar{r}_a + \frac{\sum_{u=1}^{n} (r_{u,i} - \bar{r}_u)(\text{sim}(a, u))}{\sum_{u=1}^{n} \text{sim}(a, u)}
\]

where \(n\) is the number of users.

### 20.2.8 Retrieval methods

**Filter Based Retrieval** : Retrieves cases which attributes comply some conditions. It computes the boolean AND operator over the condition of each attribute. The evaluation of each attribute is configured with predicates:

- Equal.
- NotEqual.
• QueryLessOrEqual.
• QueryLess.
• QueryMoreOrEqual.
• QueryMore.
• OntologyCompatible: To use with instances. Returns true if the case instance is under the query instance.

Expert Clerk Retrieval: This is the method of the ExpertClerk system [51]. This algorithm chooses the first case that is closed to the median of all cases. Then the remaining are selected taking into account negative and positive characteristics. A characteristic is an attribute that exceeds a predefined threshold with respect to the median case. It is positive if is greater than the value of the median. And negative otherwise. The number of positive plus the negative characteristics is used to rank the cases and obtain the retrieved cases.

The first sample product (1st-SP) is the case closest to the median of the cases. Let \( C = c_1, c_2, \ldots, c_k \) be the set of cases and let \( c_i = v_{i1}, v_{i2}, \ldots, v_{in} \) be the set of attribute values of a case \( c_i \). Then, the median \( c_{med} \) of \( C \) is calculated by:

\[
c_{med} = \left( \frac{1}{k} \sum_{j=1}^{k} v_{j1}, \frac{1}{k} \sum_{j=1}^{k} v_{j2}, \ldots, \frac{1}{k} \sum_{j=1}^{k} v_{jn} \right)
\]

The distance (\( D \)) between \( c_{med} \) and a case \( c_i \) is given by:

\[
D(c_{med}, c_i) = \frac{\sum_{j=1}^{n} W_j \times d(c_{med,j}, c_i,j))}{\sum_{j=1}^{n} W_j}
\]

where \( c_{med,j} \) is the j-th attribute of \( c_{med}, v_{i,j} \) is the j-th attribute value of \( c_i \), and \( W_j \) is the j-th attribute weight.

In case an attribute is an enumerative type, the most dominant value among attribute values is chosen as a median.

1st-SP is a record whose distance \( D(c_{med}, c_i) \) from \( c_{med} \) is the shortest among \( c_j (1 \leq j \leq k) \).

Then positive and negative characteristics of retrieved records are generated, and following cases are selected. For each retrieved record, attribute distance \( AD_j \) between each attribute value and that of \( c_{med} \) is calculated by:

\[
AD_j(c_i,j, c_{med,j}) = W_j \times (c_i,j - c_{med,j})
\]

If the absolute value of \( AD_j(c_i,j, c_{med,j}) \) exceeds a predefined threshold, \( c_i,j \) is regarded as a characteristic of the record \( c_i \), and is called a positive characteristic if the value of \( c_i,j \) is more highly ranked than that of \( c_{med,j} \), otherwise it is called a negative characteristic. The total number of characteristics is the sum of positive characteristics and negative characteristic.

The k-1 records having the maximum number of characteristics are also retrieved together with 1st-SP.
20.2.9 Cases selection

Select All or top-K cases These are the most basic methods that select all cases or only the k cases with highest similarity. They are implemented in jcolibri.method.retrieve.selection.SelectCases.

Compromise Driven Selection Cases selection approach that chooses cases according to their compromises with the user’s query. The compromises are those attributes that do not satisfy the user’s requirement. That way cases are selected according to the number of compromises and their similarity. See [37] for details.

\[
\text{Q: query, Candidates: cases}
\]

\[
\text{CDR}(Q, \text{Candidates})
\]

\[
\text{RS} = \{
\}
\]

\[
\text{while (|Candidates|>0) }
\]

\[
\quad \text{\text{C}1 = first(Candidates)};
\]

\[
\quad \text{RS += \{C\}};
\]

\[
\quad \text{like(C1,Q) += \{C1\}};
\]

\[
\quad \text{covered(C1) += \{C1\}};
\]

\[
\quad \text{for (C2 : Candidates)}
\]

\[
\quad \quad \text{if ( compromises(C1,Q) subset_or_equal compromises(C2,Q) )}
\]

\[
\quad \quad \quad \text{covered(C1) += \{C2\}};
\]

\[
\quad \quad \text{if (compromises(C1,Q) equal compromises(C2,Q))}
\]

\[
\quad \quad \quad \text{like(C1) += \{C2\}};
\]

\[
\quad \}
\]

\[
\quad \text{Candidates -= covered(C1)};
\]

\[
\}
\]

\[
\quad \text{return RS} ;
\]

The += and -= operations are set addition and subtractions. The compromises set of a case is defined by:

\[
\text{compromises}(C,Q) = \{a \in A : C.a fails to satisfy the preference of the user}\}
\]

where \(A\) is the set of attributes of the case and \(C.a\) is the value of the attribute \(a\) for a case \(C\).

BoundedRandomSelection Is the fist algorithm proposed in [53]. It is the simplest diversity strategy: select the k cases at random from a larger set of the b·k most similar cases to the query.

\[
\text{t: query, C: case-base, k: #results, b: bound}
\]

\[
\text{BoundedRandomSelection(t,C,k,b)}
\]

\[
\quad C' = bk cases in C that are most similar to t
\]

\[
\quad R = k random cases from C'
\]

return R
20.2 Methods for recommender systems

**GreedySelection** This method (see [53]) incrementally builds a retrieval set, \( R \). During each step the remaining cases are ordered according to their quality with the highest quality case added to \( R \).

The quality metric combines diversity and similarity. The quality of a case \( c \) is proportional to the similarity between \( c \) and the query, and to the diversity of \( c \) relative to those cases so far selected in \( R = r_1, ..., r_m \).

\[
\text{Quality}(t, c, R) = \text{Similarity}(t, c) \times \text{RelDiversity}(c, R)
\]

\[
\text{RelDiversity}(c, R) = 0 \text{ if } R = \{\}
\]

\[
= \frac{\sum_{i=1}^{m} (1 - \text{Similarity}(c, r_i))}{m}, \text{ otherwise}
\]

The pseudocode of the algorithm is:

```
t: query, C: case-base, k: #results, b: bound
GreedySelection(t,C,k){
    R = {}
    for(i=1 to k){
        Sort C by Quality(t,c,R) for each c in C
        R = R + First(C)
        C = C - First(C)
    }
    return R
}
```

This algorithm is very expensive. It should be applied to small case bases.

**BoundedGreedySelection** Tries to reduce the complexity of the greedy selection algorithm first selecting the best \( bk \) cases according to their similarity to the query and then applies the greedy selection method to these cases (See [53]).

```
t: query, C: case-base, k: #results, b: bound
BoundedGreedySelection(t,C,k,b){
    C' = bk cases in C that are most similar to t
    R = {}
    for(i=1 to k){
        Sort C' by Quality(t,c,R) for each c in C'
        R = R + First(C')
        C' = C' - First(C')
    }
    return R
}
```
21 Other Features

jCOLIBRI2 includes several other features implemented by external contributors. This section briefly describes these features.

21.1 Visualization of a Case Base

Located in jcolibri.extensions.visualization.CasesVisualization, this class is a wrapper to the InfoVisual library developed by Josep Lluis Arcos (IIA-CSIC).

This visualization tool computes the distance among all cases and visualizes them according to that distance. Cases must be identified by a “class” tag used to identify the cases (assign the same color). This tag is obtained from the Id attribute of the solution of the cases.

The visualization behavior is explained in the Test 9 of the examples. Following listing shows an excerpt of the code and Figure 32 is a snapshot of the visualization.

Listing 40: Visualization code

```
// Configure connector and case base
Connector _connector = new PlainTextConnector();
    _connector.initFromXMLfile(jcolibri.util.FileIO.findFile("jcolibri/test/test9/plaintextconfig.xml"));
CBRCaseBase _caseBase = new LinealCaseBase();

// Load cases
_caseBase.init(_connector);

// Configure NN
NNConfig simConfig = new NNConfig();
...

// Visualize the case base
jcolibri.extensions.visualization.CasesVisualization.
    visualize(_caseBase.getCases(), simConfig);
...
```

21.2 Classification and Maintenance

jCOLIBRI2 includes several classification and maintenance methods developed by Lisa Cummins and Derek Bridge (University College Cork, Ireland). The classification methods are located in
the packages `jcolibri.method.reuse.classification` and `jcolibri.method.revise.classification`.

Regarding the maintenance, the `jcolibri.method.maintenance` package includes methods that decide which cases should be removed from a case base to improve the accuracy of the CBR application. Besides these methods, there are other classes that evaluate the maintenance process. They are located in the `jcolibri.extensions.maintenance_evaluation` package.

Read the documentation of those packages for details. Moreover, Tests 7, 14 and 15 show how to use these features.
22 Getting support

The first place you should consult for support is the documentation supplied with jCOLIBRI2 in the doc directory. There you will find the complete API documentation with descriptions of all classes and files in the framework. This folder also contains class and sequence UML diagrams of the most important components of jCOLIBRI2.

The other major source of information are the tests included in the framework. These tests serve as “programming recipes” and show how to use the components of jCOLIBRI2 to implement CBR applications with different characteristics.

If you cannot solve your problem/question with the provided documentation, the preferred way of getting support is the developers mailing list:

http://sourceforge.net/mailarchive/forum.php?forum_name=jcolibri-cbr-developers

You can post by sending your question to:

jcolibri-cbr-developers@lists.sourceforge.net

Finally, you can subscribe to the list and receive messages from other developers (do not worry, it is a low volume list):

https://lists.sourceforge.net/lists/listinfo/jcolibri-cbr-developers
23 Contributing to jCOLIBRI

A contribution is a set of methods or classes that extend the functionality of the framework and are not bounded into the main release of jCOLIBRI. In earlier versions of the framework, contributions were called extensions and were distributed into the main release. However, the increasing interest of the community has driven us to create this new way for including third party code, keeping its own authorship and license.

In this section you will find some simple steps to submit new contributions.

23.1 Required elements

To submit a contribution you must define and include the following elements:

- Name of your contribution.
- Complete source code of your contribution. Source code should be documented according to the JavaDoc standard. Please, add the author’s name and affiliation in the beginning of each file.
- The license file. We recommend to adopt LGPL as it is the jCOLIBRI's license. In general terms, this license allows the commercial use of the software, although it maintains the authorship of the code. Of course, you can define your custom license.
- A text or html file with the credits of the contribution: authors, affiliation, etc.
- An image file of your affiliation logo.
- Contact or technical support e-mail.
- Some example applications.

We will compile and check the provided source code and generate the binary files. Documentation will be also generated into a separate folder named “doc”. This content will be packaged into a .jar file and published together with the license and credits files. Users will use the contribution by importing this .jar package into their projects.

The credits and logo files will be used to add a record in the jCOLIBRI contributions web page at http://gaia.fdl.ucm.es/projects/jolibri/jolibri2/contributions.html. This record will include a link to download the contribution from the jCOLIBRI project at SourceForge.net.

23.2 Example applications

jCOLIBRI includes an application named “jCOLIBRI2 Tester” (“Examples” in the Windows Menu) that shows how to use certain components or methods of the framework.
23.3 How to submit a contribution

If you develop a new contribution, you should create some simple examples that will be incorporated into this application.

An example is defined with a text file containing:

- Name.
- Short description. Allows html tags to decorate the text.
- Class to execute.
- Path to the most important documentation files (several lines).

This information must be included into a text file where each example is separated by a line containing the <example> tag. Here there is an example (extracted from the examples.config file in jcolibri/test/main):

```
Listing 41: Config file for the Examples application

Test 1 – Database – kNN
Test 1 shows how to use a simple database (Hibernate) ...
jcolibri.test.test1.Test1
doc/api/src-html/jcolibri/test/test1/Test1.html
doc/api/jcolibri/test/test1/Test1.html
doc/api/jcolibri/test/test1/TravelDescription.html
doc/api/jcolibri/connector/DataBaseConnector.html
doc/api/jcolibri/method/retrieve/NNretrieval/NNScoring ...
<example>
Test 2 – Enumerated types
Test 2 extends Test 1 to show the use of enumerated ...
jcolibri.test.test2.Test2
doc/api/src-html/jcolibri/test/test2/Test2.html
doc/api/jcolibri/test/test2/Test2.html
doc/api/jcolibri/test/test2/TravelDescription.html
doc/api/jcolibri/test/test2/MyStringType.html
doc/api/jcolibri/connector/DataBaseConnector.html
doc/api/jcolibri/method/retrieve/NNretrieval/NNScoring ...
<example>
Test 3 – Compound attributes
...
```

23.3 How to submit a contribution

Please, pack everything into a zip file and send it to jcolibri [at] fdi.ucm.es. You can use that address to solve any question about the submission.

Thank you for your interest and contribution to jCOLIBRI.
24 Versions ChangeLog

This section summarizes the main changes among versions.

24.1 Version 2.1

There is an important change in the Nearest Neighbour retrieval implementation. In version 2.0 there is a KNNRetreivalMethod class that receives a KNNConfig object. This configuration object contains the number of cases to be selected (the k value). As jCOLIBRI 2.1 includes several new selection algorithms, the k parameter has been removed from the KNNConfig object. This way, the Nearest Neighbor method always returns all cases (into RetrievalResult objects) and then, a selection algorithm must be executed.

This change implies some important modifications of the names:

<table>
<thead>
<tr>
<th>version 2.0</th>
<th>version 2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>jcolibri.retrieval.KNNretrieval</td>
<td>jcolibri.retrieval.NNretrieval</td>
</tr>
<tr>
<td>KNNConfig</td>
<td>NNConfig</td>
</tr>
<tr>
<td>KNNRetrievalMethod</td>
<td>NNScoringMethod</td>
</tr>
</tbody>
</table>

Some other classes have been reallocated into other packages.

24.1.1 What’s new in Version 2.1

Version 2.1 includes methods for developing recommendation systems. Some of the implemented methods can be used in general CBR applications and other are specific for recommender systems.

General CBR methods:
- Filtering Retrieval method.
- XML utils to serialize cases and queries.
- Methods to obtain the query graphically (using forms).
- Methods to display cases.
- Cases retrieval using diversity.
- Cases selection using diversity.

Specific methods for recommendation systems:
- Methods to implement the Expert Clerk recommender.
• Methods to implement collaborative recommenders.
• Methods to obtain the query from user profiles.
• Methods to implement Navigation by Asking recommenders.
• Methods to implement Navigation by Proposing recommenders.
• Local similarity measures for recommender systems.

jCOLIBRI 2.1 includes 14 new examples that implement different recommenders.

24.2 Version 2.0

Main release of the framework and starting point of this changelog.
References


[27] M. Jaczynski and B. Trousse. An object-oriented framework for the design and


