Indiana University Case-Based Reasoning Framework

TECHNICAL REPORT 617
IUCBRF: A Framework For Rapid And Modular Case-Based Reasoning System Development
Report Version 1.0[*]

Steven Bogaerts and David Leake
Computer Science Department
Lindley Hall, Indiana University
150 S. Woodlawn Avenue
Bloomington, IN 47405, U.S.A.
{*sbogaert, leake*}*@cs.indiana.edu*

July 22, 2005

[*] Copyright (c) 2005 by Steven Bogaerts and David Leake. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/). Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

Abstract

The Indiana University Case-Based Reasoning Framework (IUCBRF) is a freely available open-source framework, written in Java, to facilitate the development of case-based reasoning (CBR) systems. IUCBRF provides code to handle many of the general, domain-independent aspects of CBR systems, freeing developers to consider only the domain-dependent aspects of the system. The framework is designed to facilitate fast and modular development of CBR systems, providing a foundation for code sharing by those developing CBR systems. This report discusses the capabilities and benefits of IUCBRF and provides an introduction to the framework for prospective users. JavaDoc documentation and information on requesting the IUCBRF code are available by following the IUCBRF link from the IU CBR resources page, http://www.cs.indiana.edu/~leake/cbr_resources.

Contents

1 Introduction 5

2 Motivations for Developing the Framework 6
- 2.1 Furthering Code Reuse . . . 6
- 2.2 Serving as a Pedagogical Tool . . . 6

3 Capabilities Summary 7

4 Framework Design 8
- 4.1 Achieving Independence from the Domain and From Other Components . . . 9
- 4.2 Isolating Dependent Code . . . 9

5 Components of a CBR System Developed with IUCBRF 9
- 5.1 Processes . . . 11
  - 5.1.1 Retrieval . . . 11
  - 5.1.2 Adaptation . . . 11
  - 5.1.3 CCBR Components . . . 12
  - 5.1.4 Maintenance . . . 12
  - 5.1.5 Performance Monitor . . . 13
- 5.2 Representation . . . 13
  - 5.2.1 Features . . . 14
  - 5.2.2 Cases . . . 14
  - 5.2.3 Case Base / Indexing . . . 15
  - 5.2.4 Domains . . . 16
- 5.3 Utilities . . . 17
  - 5.3.1 System Tester . . . 17
  - 5.3.2 Random Generation . . . 17
  - 5.3.3 Standard GUI Components . . . 18

6 The Quadrant Example 20

7 Running the Quadrant Example 26

8 Components Under Development 26
- 8.1 Domain Generation And Exploration . . . 27
- 8.2 D-Tree Indexing Of A Case Base . . . 27
- 8.3 Database Connectivity . . . 27
- 8.4 Performance Monitoring . . . 27

9 Related Work 28

10 Conclusion 29

11 Acknowledgments 30

A IUCBRF License **31**

B Samples Of Source Code **35**

B.1 Quadrant Example Source Code 35

B.2 QuadrantCCBR Example Source Code 42

B.3 Realtor Example Source Code 49

1 Introduction

Whether a case-based reasoning (CBR) system is designed for research, for an application, or for educational purposes, there is a divergence between the true goals of the effort (try a new technique, apply CBR to a particular domain, etc.) and the need to develop the basic underlying system infrastructure required by any CBR system. This infrastructure requires basic components to support fundamental processes such as retrieval, similarity assessment, case adaptation, and maintenance, and, if desired, support for Conversational CBR (CCBR). Likewise, development must address general issues of case base structure and problem and solution representation, which may arise across broad classes of goals and task domains. Once the system has been built, the developer must also code the procedures required to test system performance, which may be largely task-independent.

The Indiana University Case-Based Reasoning Framework (IUCBRF, which we will informally refer to as "the framework") is a freely-available open-source framework of Java classes for CBR systems. It provides the basic components for a Java implementation of CBR systems. The system is designed to enable the developer's focus to quickly shift to the true goals of the work, to provide a shared basis for research and experimentation, and to facilitate students' learning about CBR by making it easy for them to experiment with using and refining CBR systems. The framework's components are designed to minimize the need for customization in typical systems, while at the same time allowing for significant customization should the CBR system designer have special needs.

The IUCBRF framework is work under development. The code is subject to change, and we expect both the code and this report to be updated over time. However, even in its present form it provides a robust set of capabilities. The framework has been successfully used as the foundational CBR engine in a range of projects, including distributed collaborative maintenance of naval ship equipment (Leake *et al.* 2005), network fault analysis, visualization of case base properties, a calendar scheduling system, and a system for recommending codes for earthquake modeling (Aktas *et al.* 2004). The framework is also intended as a starting point for pedagogical uses (Bogaerts & Leake 2005), and includes code for complete implementations of simple demonstration systems to illustrate its use and the CBR process. In addition, it provides initial capabilities for automatically generating simple test sets of problems and gathering performance statistics to aid in evaluating alternative approaches.

This document provides a high- and mid-level discussion of IUCBRF, to prepare the reader for further investigation of the low-level Javadoc documentation available on-line and comments in the code itself. Those interested in simply applying the framework as rapidly as possible, rather than in understanding its motivations, may wish to start with Section 5 (p. 9), in conjunction with the example in Section 6 (p. 20), and its associated source code in Appendix B.1. JavaDoc documentation and information on requesting the IUCBRF code are available by following the IUCBRF link from the IU CBR resources page, http://www.cs.indiana.edu/~leake/cbr_resources.

IUCBRF is released under the Open Source License (OSL), and may be used under the conditions of that license, provided in appendix A. *We ask that all publications describing results obtained using IUCBRF code acknowledge use of the framework and cite this technical report.* Parties who wish to use IUCBRF without the restrictions of the OSL may contact the Indiana University Research and Technology Transfer Corporation (http://iurtc.iu.edu/) to arrange alternative licensing arrangements.

This document assumes a familiarity with the basic tenets of CBR and the processes and design considerations for CBR systems. For those wishing to learn more about CBR, numerous introductions exist, including (Kolodner 1993; Leake 1996a; Riesbeck & Schank 1989; Watson 1997). Two overviews available on-line are (Leake 1996b; Aamodt & Plaza 1994). Conversational CBR is described in detail in (Aha, Breslow, & Munoz-Avila 2001).

IUCBRF has chiefly been designed and implemented by Steven Bogaerts under the guidance of David Leake. Some code has been donated by Thomas Thomas, Neil Briscombe, and Peter Siniakov, acknowledged where appropriate in the source code. Individuals interested in collaboration on refinements or extensions to the framework are encouraged to contact Steven Bogaerts (sbogaert@cs.indiana.edu).

2 Motivations for Developing the Framework

The IUCBRF project has two primary goals: to facilitate the development of CBR systems through code reuse, and to facilitate the teaching of CBR by removing much of the overhead of implementing pedagogical CBR systems, enabling students to learn by experiments focused on specific aspects of primary interest.

2.1 Furthering Code Reuse

While each CBR system may have different requirements, many necessary components have significant commonalities. Unfortunately, it can be exceedingly difficult to reuse existing code. One impediment to code reuse is that system designs often include domain-dependent implementations of components. For example, the retrieval process may be based on intimate knowledge of the case base structure, or the adaptation component may assume a particular format for the retrieved cases. Modifying this code for use in a new domain is likely to be problematic.

A second impediment is that components of CBR systems may be tied to each other. That is, one component may make assumptions about the implementation of another component. If the new system does not use the same techniques for each component as the old system, it is less likely that the components of the old system will be applicable.

To the extent that CBR systems can be built from domain-independent components decoupled from one another, the design of new CBR systems should require less rewriting of common code, and it should be possible to swap components in and out of the system without requiring detailed knowledge of other components of the system. This should enable designers to spend less time laying the foundation of a CBR system, researchers to spend more time investigating the theoretical issues of interest, and practitioners to build running systems more quickly.

2.2 Serving as a Pedagogical Tool

The existence of standard, modular CBR code can be useful in educational settings as well, by providing a starting point for students studying CBR, machine learning or artificial intelligence. In an introductory artificial intelligence course, such code can be used to illustrate the purposes of the various components of a CBR system, and, using concrete examples, explore the change in performance that occurs when different techniques are used for a component. This can all be accomplished without the significant coding that would typically be required, enabling students to learn the concepts through experience within the time frame of a homework assignment, rather than a semester project. More advanced artificial intelligence courses can also make use of such a framework. For example, a system could be put in place with which the students could implement techniques from current research papers, or from their own ideas, without needing to worry about coding details orthogonal to the topic of interest. For a more complete discussion of the application of IUCBRF to CBR instruction, see (Bogaerts & Leake 2005).

3 Capabilities Summary

To facilitate code reuse, the framework provides standard implementations of components, with well-defined interfaces for custom implementations. Implementations exist for components such as domain definition, case base storage, retrieval, adaptation, performance monitoring, and maintenance, as well as complete GUIs and component parts for use in custom GUIs.

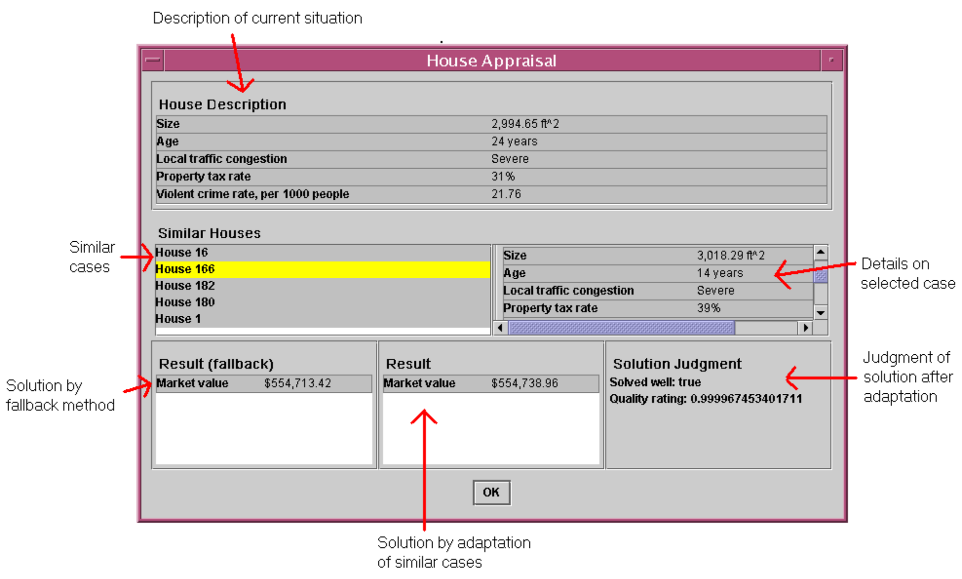
For example, figure 1 shows a standard GUI, the episode summary panel, for the simple sample realtor domain (which performs house appraisal) provided with IUCBRF. This figure provides illustrations of:

- Problem and solution feature definition
- Similarity metrics and retrieval
- Adaptation technique of retrieved cases
- Hooks for alternative problem-solving approaches
- Solution quality

In particular, the screenshot shows:

- A problem description for the current situation (description of the house to be appraised)
- A list of the most relevant cases, with the selected case displayed
- Solutions generated by the system, both by using CBR and by applying an independent "reference method," used to evaluate solution accuracy.
- The result of comparing the CBR solution and reference solution to assess solution quality.

There are standard implementations for the above and many other functions, and the framework is also designed to be easily extendable for the creation of domain-dependent implementations. Note such custom extensions are more likely to be needed for some features than others. For example, simple weighted average and majority vote adaptation techniques are available, but many domains will require more refined approaches.


Description of current situation
House Appraisal
Similar cases
Details on selected case
Solution by fallback method
Judgment of solution after adaptation
Solution by adaptation of similar cases
OK

House Description

Size	2,994.65 ft^2
Age	24 years
Local traffic congestion	Severe
Property tax rate	31%
Violent crime rate, per 1000 people	21.76

Similar Houses

House 16
House 166
House 182
House 180
House 1

Size	3,018.29 ft^2
Age	14 years
Local traffic congestion	Severe
Property tax rate	39%

Result (fallback)

Market value	$554,713.42

Result

Market value	$554,738.96

Solution Judgment

Solved well: true
Quality rating: 0.999967453401711

Figure 1: Episode Summary Of A Realtor Example

4 Framework Design

The general design of IUCBRF is as follows. There are several *component packages*, each of which minimally includes a *base component class*. This class implements the general operations (if any) common to all implementations of that component type, and sets the foundation for the implementation of other common functions that are implementation-dependent. A *component subclass* extends the base component class, implementing some or all of the base component class' abstract methods.

All classes in the framework are *domain-independent*. It is the designer's responsibility to implement a few remaining *domain-dependent* methods in the framework, as well as any additional domain-dependent components that the system may need.

As discussed previously, two typical challenges to reusing code from a previous CBR system are:

1. The existing system is tied to a domain
2. Components of the system are tied to each other

IUCBRF aims to decouple the problem from the domain and the components from each other. By starting from general code that can be applied to many domains, with many possible combinations of components, systems can be developed much more quickly than a system from scratch. In general terms, this is accomplished through *polymorphism* - the ability of an object to have a standard interface, yet take many forms, depending on the domain and system properties. Object-oriented *design patterns* (Gamma *et al.* 1995) may also be used to handle details in some situations.

4.1 Achieving Independence from the Domain and From Other Components

To illustrate how IUCBRF maintains independence from the domain, consider the example of features in problem descriptions (a similar discussion applies for solution descriptions). A problem consists primarily of a collection of features. A feature is any class that implements the `Feature` interface, which requires each feature to be able to perform basic operations, such as comparing itself to another feature.

When two problems must be compared, their feature collections can be iterated, and the corresponding features compared. The code governing this process remains the same whether working in a domain of 5 integer features, or a domain of 500 features of many types, including user-defined types. The details of the domain are hidden from the other components of the code as much as possible.

Similarly, the implementation details of one component are hidden from the other components as much as possible. For example, the collection of features in a problem may be implemented in any way, provided that an iterator on those features can be created, and a retrieval component can examine the features and measure similarity between each corresponding pair. The retrieval component need not consider how features are stored or how they are compared. These details are hidden and can be changed without affecting the retrieval component or other outside components.

4.2 Isolating Dependent Code

Occasionally methods in a component **must** know details of the domain or the implementation of some other component in use. For example, see the pseudo code in figure 2. Here, a specialized adaptation technique `AdaptationTechnique1` requires some knowledge about the structure of problem descriptions. The portion of the code that works with this knowledge (`doDomainDependentOperations`) is declared abstract and passed to a subclass. The subclass `DomainDependentAdaptation`, implemented by the designer, provides the details of the domain-dependent operations safely, because the entire class is assumed to be domain-dependent. This situation of passing on details to a subclass is an instance of the *template* design pattern, in which the skeleton of an algorithm is defined in an abstract class, deferring some steps to particular implementations in subclasses (Gamma *et al.* 1995).

In IUCBRF, only `doDomainDependentOperations` must be implemented in the `DomainDependentAdaption` class. The code for `doCommonOperations` will already be inherited from `AdaptationTechnique1`.

In some circumstances, the domain-dependent operations (such as those in `doDomainDependentOperations`) may include constructing an object of a domain-dependent type. This is an example of the *factory* design pattern, in which the interface for object creation is defined, but subclasses decide exactly how to instantiate (Gamma *et al.* 1995).

5 Components of a CBR System Developed with IUCBRF

The components of a CBR system developed with IUCBRF can be divided into three basic categories: 1) Processes, 2) Representation, and 3) Utilities.

```
// IUCBRF code
public abstract class AdaptationTechnique1
extends Adaptation
{
    public void doSomeWork() {
      // work independent of the specific Solution implementation
      doCommonOperations();
      // work dependent on the Solution implementation
      doDomainDependentOperations();
      doCommonOperations();
    }
    public abstract void doDomainDependentOperations();
    ...
} // end class AdaptationTechnique1
//////////////////////////////////////////////////////////////////////
// domain-dependent code - implemented by the designer
public class DomainDependentAdaptation
extends AdaptationTechnique1
{
    public void doDomainDependentOperations() {
    // safe to make assumptions about problem structure here,
    // because this is domain dependent code
    }
    ...
} // end class DomainDependentAdaptation
```

Figure 2: Pseudo code for isolating dependent code

To build a system using IUCBRF, the designer must specify the needed components, using standard implementations in the framework and any customized implementations that are needed. Many standard techniques for these components are already implemented in the framework, and are ready to use with little or no additional coding by the designer, but there is always the option to override portions of a technique or to design new techniques that implement a component's interface or extend a standard component.

The framework contains several completely-implemented examples, illustrating functionality in toy domains and providing an easy starting point for the developer or student who would like to develop a CBR system by case-based reasoning—adapting a similar example to fit his or her needs. One of the examples is a very simple system to take as input a point on the x-y plane, and return the quadrant in which it lies. In this domain, a problem is an ordered pair (x, y), and a solution is that point's quadrant (1, 2, 3, or 4, counterclockwise from upper right). We use this domain as a running example. Where applicable in the following, the corresponding component used in the quadrant system is discussed. A full discussion of the quadrant example is provided in Section 6, with the complete code in Appendix B.1.

5.1 Processes

Components related to standard CBR processes and their supporting tools include:

- Retrieval
- Adaptation
- Conversational interactions (for CCBR)
- Maintenance
- Performance Monitoring

5.1.1 Retrieval

The pre-implemented retrieval in IUCBRF is a k-nearest-neighbor (k-NN) algorithm. That is, the k cases with a problem description nearest to the current problem, according to a similarity criterion, are retrieved. The designer has the option of saving the distance of each retrieved case from the current problem, as additional information to be used during adaptation (e.g., for distance-weighted averaging). In the Quadrant example, 5-nearest neighbor is used, as specified in the `QuadrantSystem` constructor.

5.1.2 Adaptation

IUCBRF provides a small set of domain-independent case adaptation techniques, along with a standard interface from which arbitrary user-defined adaptation techniques can be implemented to function with the rest of the framework-based system. The standard adaptation approaches are:

- No Adaptation: The “null adapter” simply returns the solution of the first retrieved case as the system solution.
- Weighted Average: Given a list of cases and weights, returns a weighted average of the solution values as the adapted solution.
- Weighted Majority: Given a list of cases and weights, takes a weighted vote and returns the winning value. In the event of a tie, whichever value is associated with a higher-ranked case is declared the winner.

The weighted adapters assume that the solution type for this domain can be converted to a `double` value. The weights used can either be provided statically, or can be determined dynamically in each problem solving episode. One example of dynamic weighting is in the distance weighted adapter. This adapter uses the distances of retrieved cases (saved in the retrieval step) to determine the weight of that case, with closer neighbors given higher weights. A distance weighted adapter can be used with a weighted average adapter, a weighted majority adapter, or any custom weighted adapter. The Quadrant example uses a distance weighted majority adapter (a majority vote), specified in the `QuadrantSystem` constructor.

Note that in terms of the "Retrieve, Reuse, Review, Revise, Retain" (Aamodt & Plaza 1994) description of the CBR cycle, the adaptation portion of the framework encompasses the Reuse, Review, and Revise steps: it first obtains a solution and then perhaps enters a cycle of reviewing and revising that solution.

5.1.3 CCBR Components

Conversational CBR systems incrementally refine case selections by interactively eliciting additional information from the user. To support this process, the framework provides components for case list refinement and question selection.

Because the Quadrant example is not a CCBR system, the following sections do not apply to it. For a CCBR example, please see the "Conversation" example provided with the code.

Case List Refiner The case list refiner determines the contents and order of the list of potentially useful cases given the conversation so far. There is one case list refiner currently implemented. In this refiner, the difference between each case and the current problem is calculated. Those that are above a threshold are removed from further consideration in this episode. Remaining cases are ordered by distance from the problem, shortest distance first.

Question Selector The question selector determines which question should be asked next in the conversation, based on the questions and answers so far, the cases under consideration, and the domain. The framework currently contains three implementations:

- Ordered
- Flow Chart
- Frequency

The *ordered question selector* contains a list of the questions to be asked in order. It merely asks the next question on the list.

The *flow chart question selector* determines the next question to ask by referring to a flow chart provided by the designer. The flow chart specifies which question to ask next depending on the previous question and the answer provided. Note that this approach is a generalization of the ordered question selector.

The *frequency question selector* determines which unknown feature in the current problem has a known value in the largest number of cases under consideration. The question corresponding to this feature, as specified in the domain, is the next question asked.

5.1.4 Maintenance

The framework provides simple facilities for case base maintenance include case removal and addition. Triggering mechanisms (Leake & Wilson 1998) for both operations can be customized, and two implementations currently exist. One simple implementation, "null maintenance", never adds nor removes any cases. Another implementation, "basic maintenance" periodically checks for infrequently-used cases to remove, and accepts for addition any new case corresponding to a problem that was just solved successfully.

No maintenance is used in the Quadrant example, as is specified by the `NullMaintenance` value in the `QuadrantSystem` constructor.

5.1.5 Performance Monitor

Each CBR system using this framework has a customizable performance monitor, which tracks the performance of the CBR system in several ways. This data could be used to guide maintenance or to evaluate the overall success of CBR components. The performance monitor tracks:

- System age, measured by the number of problems processed
- The number of cases in the case base
- The average retrieval time
- The average adaptation time
- System competence - the percentage of problems for which a solution was found
- Solved-well rating - the percentage of problems for which an acceptable solution was found
- Average quality - the average rating for found solutions

Some additional performance data is tracked for each individual case; see the Case section for details.

The standard performance monitor is used in the Quadrant example, as specified in the `Quad-rantSystem` constructor.

5.2 Representation

Components related to representation include the following:

- Features
- Cases
- Case Base / Indexing
- Domain

5.2.1 Features

Features are the primary placeholders for information about both problems and solutions. The framework implements the following feature types:

- Double
- Integer
- String (with some special handling for long strings)
- Boolean
- The set Yes, No
- The set very mild, mild, moderate, severe, very severe
- Term vector
- Internet address

Any customized feature type can be created by extending an existing feature, the `AbstractFeature` class, or the `AbstractFSMFeature` class (for features with values from a finite set—FSM stands for "Finite Set Member"), or by implementing the `Feature` interface.

Every instance of a feature class refers to a specific feature, with the general template for that feature kept in a feature specification that resides with the domain. (See 5.2.4.) These feature specifications are tied to the feature instances via a unique feature key.

Features are organized in a feature collection, which is a strongly-typed collection of features. The particular data structure or collection organization technique used is customizable, and for example may be a vector or hash map.

In the Quadrant example, the problem features, corresponding to the `x` and `y` coordinate, are both instances of the `DoubleFeature` class. The solution feature, corresponding to the quadrant, is an instance of the `IntegerFeature` class. This is set up in `QuadrantSystem.setupDomain`.

5.2.2 Cases

A case contains the following:

- Problem
- Solution
- Inactive Contexts
- Use Counts
- Time Of Creation

- Source

Both a problem and a solution consist mainly of a feature collection.

Each case also has a set of *inactive contexts*. This describes the contexts of system operation (e.g. “Normal”, “Testing”) in which a case should not be considered. A case is treated as absent from the case base during such contexts. For example, to ignore a case for leave-one-out testing, the “Leave One Out Testing” context is temporarily added to the set of inactive contexts for that case. (A simple leave-one-out test example is provided in Appendix B.1.) The designer is free to create custom contexts.

A case also maintains a *use count* and a *successful use count*. Each time a case is retrieved, its use count is incremented. If the resulting solution is deemed of high quality, then each contributing case’s successful use count is incremented. This data can be useful for maintenance purposes. For example, rarely used or rarely successful cases may be targets for removal.[1]

Cases record the point in the system’s history they were added (the *time of creation*). This is recorded in terms of the number of problems previously processed (e.g., a given case might have been after “212 problems processed”). This data can be useful for maintenance purposes. For example, older cases may be considered more likely to be out of date and targeted for removal.

Each case is also associated with a *source*. For example, if a case came from a successful problem solving episode, then the case source would be “system-generated.” More specific sources can also be created. Source data may be useful in maintenance. For example, if an overwhelming majority of failed problem solving episodes involve cases from a particular source, then there may be some anomaly at that source leading to untrustworthy cases, and all cases from the source should be reexamined. Case source information could also be useful in multi-case-base-reasoning (MCBR), where methods could be applied to respond to the differences between sources (Leake & Sooriamurthi 2004).

The Quadrant example demonstrates a minimal system, and so it does not make use of case contexts, use counts, or historical data. The source of all cases in the initial case base is “Randomly Generated”, but a “System Generated” case is added when the system solves the problem presented to it.

5.2.3 Case Base / Indexing

IUCBRF currently implements two simple indexing schemes:

- Flat case base
- B-tree-backed case base

A simple flat case base (for example, as specified in `QuadrantSystem.setupCaseBase`), stores all cases in an unordered list. This is suitable for small systems and systems in an early prototype stage, before more robust and scalable indexing and storage schemes can be developed.

[1]Note that the implementation of *successful use count* ignores considerations of credit/blame assignment, which may give unexpected results when solutions are generated from multiple cases. For example, if 5-NN correctly classifies a case based on four cases voting for category membership and one voting against, all five cases—including the dissenting vote—have their successful use count incremented.

The framework also provides the option of storing the cases using a B-tree-backed case base. With this approach, selected features are identified by the system designer as indices to be held in memory and used for similarity comparisons in retrieval. When the most similar cases are identified according to these indices, the full cases are retrieved from a B-tree-backed file. Essentially, the fact that a B-tree indexes the file allows for much faster access than a standard sequential or binary search. For a detailed discussion on B-trees, see (Cormen *et al.* 2001). Thus, by this approach the full cases are not held in memory, and the similarity comparisons are fast, because they are done on structures in memory, yet the full cases can still be relatively quickly retrieved once the nearest ones have been found according to their indices.

For additional indexing schemes under development, see section 8.

5.2.4 Domains

A key part of building a CBR system is to specify the domain in which the system will operate. IUCBRF domain specifications include information about the:

- Problem and solution representations
- Problem similarity measure
- Problem space (optional)
- Reference method (optional)
- Solution quality criterion (optional)

The *problem and solution representations* consist of the number and type of features, the organization of those features, which of those features are used as indices, textual descriptions of each feature, and, for CCBR systems, the question corresponding to each feature.

The standard form of the *problem similarity measure* is a set of weights for a weighted Euclidean distance over the problem features. There is also the possibility of specifying more refined measures.

The *problem space* is an optional set of probability distributions on the problem features, used for randomly generating new problems.

A *reference method* can also optionally be specified in a domain, to be used to generate "correct" solutions against which to compare those generated by CBR. A standard method, supported by IUCBRF, partitions the problem space into equivalence classes (either with tools provided by IUCBRF or by some other means), each with a prototype problem and a set of weights, one per problem feature. These weights are used to weight a linear combination of the problem features to obtain a solution. This standard reference method can be disabled or overridden in favor of an alternative problem solving method, if available.

When a reference method has been defined, a *solution quality criterion* can also be specified, to automatically determine the quality of a solution generated by the CBR system, for evaluation purposes.

In the Quadrant example, significant portions of the problem and solution representations are already handled by the framework. The problem space generation capabilities are not used in the

Quadrant example. The `QuadrantDomain` defines its own simple reference solution method and solution quality techniques, rather than using the more general predefined ones.

The other components of the CBR system use the domain specifications to perform their tasks, but their implementation does not depend on a particular domain specification. Some components are built to operate only with CCBR systems, rather than a standard CBR system. Aside from this exception, all components in the framework are domain independent. Of course, the designer is free to write components specific to a particular domain, but this may not be necessary.

5.3 Utilities

Components provided to support testing and use of systems built with the framework include:

- System Tester
- Random Generator for domains, problems, cases, and weights
- Standard GUI Components

5.3.1 System Tester

IUCBRF includes pre-defined procedures for two categories of tests to run on a system: random problem tests and leave-one-out tests. A random problem test simply randomly generates a specified number of problems according to a specified set of distributions, and runs the system on each one. A leave-one-out test temporarily removes a case from the case base, and then presents the associated problem to the system, which uses the remainder of the case base to obtain a solution. This is done for each case in the case base. For both of these categories of tests, a performance monitor describing the results is created. A simple leave-one-out test example is provided in appendix B.1.

A system tester is not used in the Quadrant example. Rather, a single problem is manually given to the system.

5.3.2 Random Generation

IUCBRF provides many facilities for random generation of various objects for use in a CBR system. This generation depends on probability distributions. Distributions already implemented by the framework include normal, uniform, and binomial. The following components can be generated:

- Domains
- Problems
- Cases
- Weights

To use a *problem generator* requires specifying a distribution for each feature of the problem. Note that a distribution generates double values, so there is an assumption that a conversion exists from double values into problem feature values.

A *case generator* requires the use of a problem generator, as well as a domain with a reference method. The problem generator is used to generate a problem, and the reference method is used to obtain a solution to that problem.

A *weights generator* is also provided, which can generate weight values according to specified distributions. This could be used, for example, to randomly assign weights to an adapter or similarity metric.

The Quadrant example uses a case generator to populate the case base. The case generator is defined together by `QuadrantSystem.constructProblemGenerator`, and `QuadrantSystem.setupCaseBase`. The problem generator is also used to test the system, by `QuadrantTestClass`.

5.3.3 Standard GUI Components

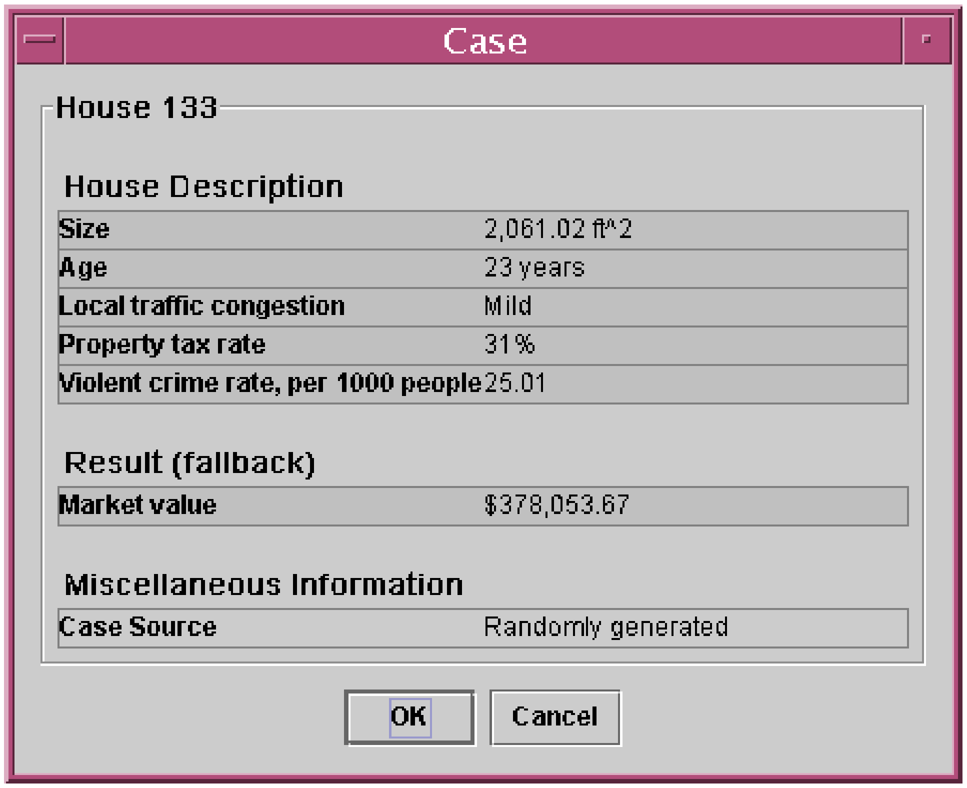
Many standard GUI components are included in the framework, which may be used to provide a complete GUI for a system. They may also be used in conjunction with designer-defined custom GUI components.

Episode Summary Panel The episode summary panel, shown in Figure 1 and discussed on page 7, displays much information related to a single problem-solving episode. The episode summary is the GUI used in the Quadrant example. This panel is one of the larger GUI components, in turn making use of several smaller components. The designer can present any of the subcomponents (say, the problem description) in a custom, designer-defined way and yet still use the standard representations of the other components.

Case Representation Panel The Case class contains methods to represent a case as a Java Swing object. Figure 3 provides an example of a case representation panel. This panel itself contains modular panels displaying representations of the problem and solution of the case.

List Viewer Panel Any class that implements `SwingRepresentable` or `SwingRepresentableWithDomain` can make use of the `ListViewer` and `ListViewerWithDomain` classes. These classes present selectable lists of swing-representable objects, with the selected object shown to the right. `Case` is one of the classes that is swing-representable, and figure 4 shows the use of a list viewer to view a list of cases.

Input Panels Input panels are extensions of the Java Swing class `JPanel`. They are designed to be plugged into a GUI to allow the input of some object. They handle the details of prompting for values, displaying an appropriate value input mechanism, and constructing the desired object based on the input. All currently defined feature types have a corresponding input panel, as do feature collections and cases. Figure 5 shows a simple feature collection input panel for the Realtor example, which itself uses the feature input panels of the corresponding features defined in the Realtor domain.



House 133

House Description

Size	2,061.02 ft^2
Age	23 years
Local traffic congestion	Mild
Property tax rate	31%
Violent crime rate, per 1000 people	25.01

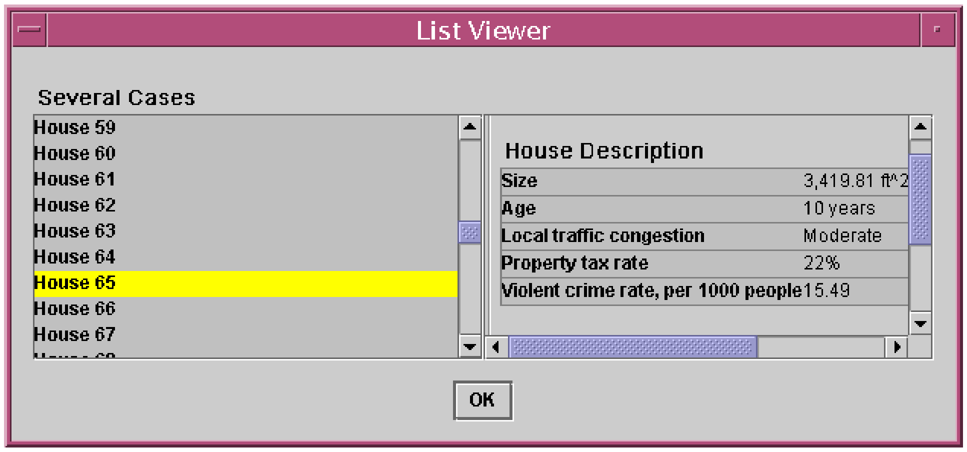
Result (fallback)

Market value	$378,053.67

Miscellaneous Information

Case Source	Randomly generated

Figure 3: Standard Case GUI Representation



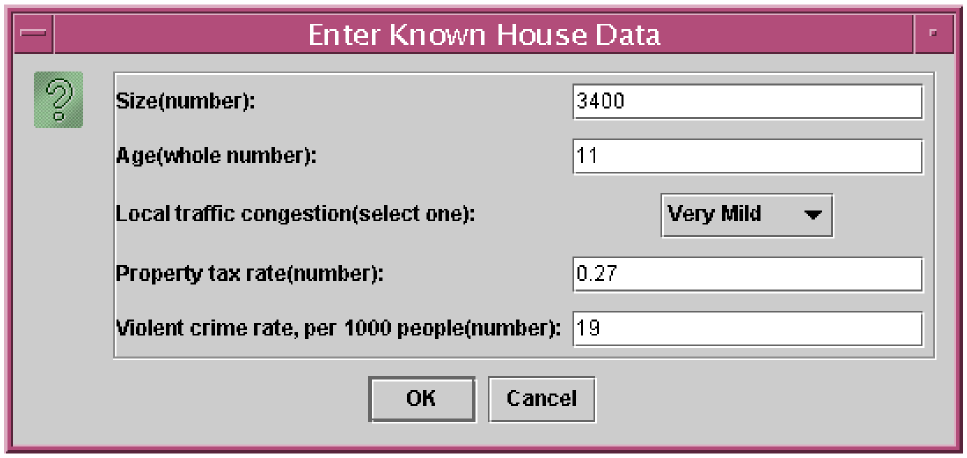
Several Cases

House 59
House 60
House 61
House 62
House 63
House 64
House 65
House 66
House 67

House Description

Size	3,419.81 ft^2
Age	10 years
Local traffic congestion	Moderate
Property tax rate	22%
Violent crime rate, per 1000 people	15.49

Figure 4: Standard List Viewer GUI Representation



Size(number): 3400
Age(whole number): 11
Local traffic congestion(select one): Very Mild
Property tax rate(number): 0.27
Violent crime rate, per 1000 people(number): 19

Figure 5: Input Panel GUI

CCBR GUI Components Some additional GUI components useful in CCBR systems are included in the framework:

- Main CCBR Panel
- Question And Answer
- Feature Editor

The main *CCBR panel* is a `JPanel` designed to contain the basic GUI needed in a CCBR system. As can be seen in figure 6, it contains four sections: a display of the current question and answer panel, the partial problem description obtained from the conversation so far, a case list panel for the cases currently under consideration, and a display of the currently selected case.

The *question and answer* panel is a `JPanel` for displaying a question, the appropriate input panel for the answer type, and a submit button. The input can be conveniently obtained from this panel, in the form of a feature. This panel is shown in the top left quadrant of figure 6.

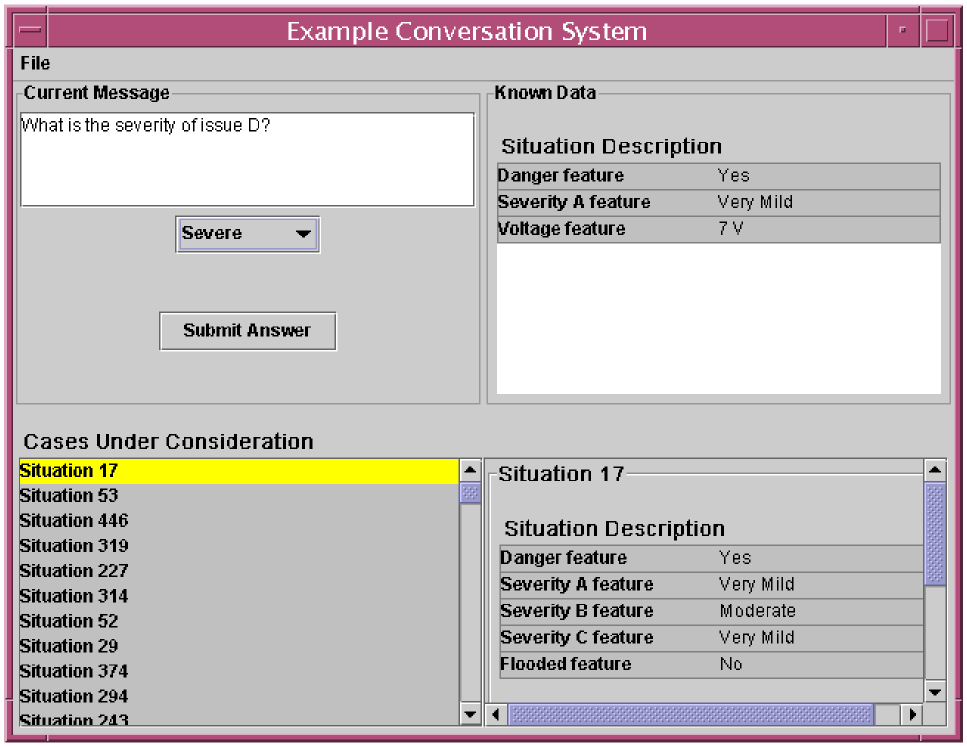
The *feature editor*, shown in figure 7, provides a GUI for a designer to define new features to be used in future conversations. The description and question text is entered and the answer type is specified.

6 The Quadrant Example

There are several examples provided with IUCBRF, to demonstrate basic system construction and use. These are extensively commented and provide a good starting point for understanding how the framework is used. The appendix contains some of these examples, for convenient reference.

Here, we provide a line by line discussion of the simple Quadrant example and its associated conversational CBR version, QuadrantCCBR. The complete code with comments is provided in the appendix and the source distribution.

In this domain, problems contain two features, x and y, both doubles in [-1, 1]. The square on the x-y plane corresponding to this problem space is divided into four equal quadrants by the


Example Conversation System

File

Current Message

What is the severity of issue D?

Severe

Submit Answer

Known Data

Situation Description

Danger feature	Yes
Severity A feature	Very Mild
Voltage feature	7 V

Cases Under Consideration

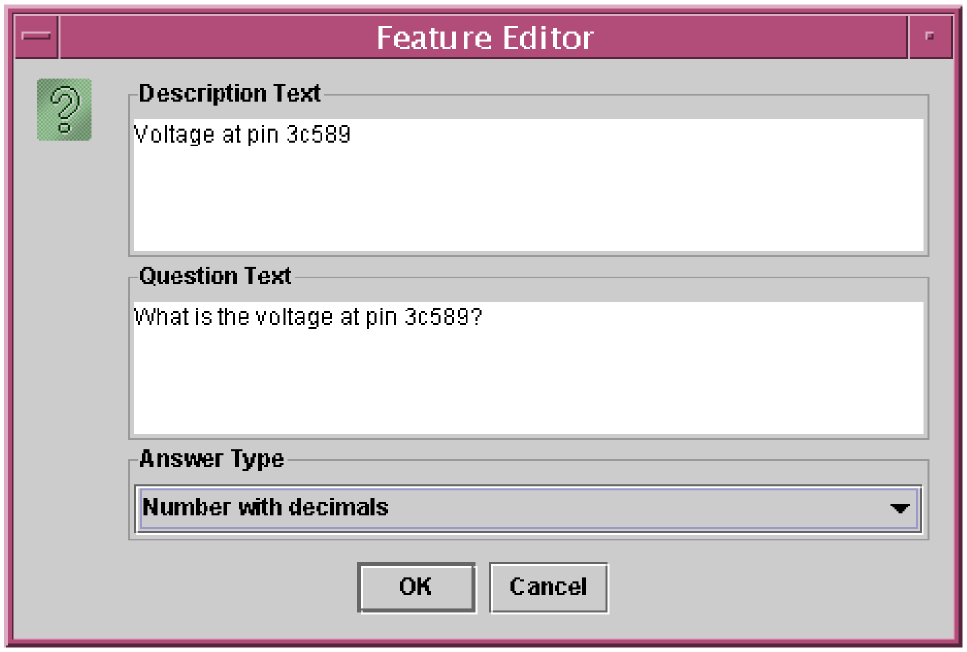
Situation 17
Situation 53
Situation 446
Situation 319
Situation 227
Situation 314
Situation 52
Situation 29
Situation 374
Situation 294
Situation 243

Situation 17

Situation Description

Danger feature	Yes
Severity A feature	Very Mild
Severity B feature	Moderate
Severity C feature	Very Mild
Flooded feature	No

Figure 6: The main CCBR Panel for a simple domain


Feature Editor

Description Text

Voltage at pin 3c589

Question Text

What is the voltage at pin 3c589?

Answer Type

Number with decimals

OK Cancel

Figure 7: A basic feature editor

coordinate axes. These quadrants are numbered 1, 2, 3, and 4, counterclockwise from upper right, as in basic algebra.

A case consists of a problem description and solution, where the problem is a pair (x,y), and the solution is its quadrant (1, 2, 3, or 4). In the non-CCBR version, when a new problem is presented to the system, the five nearest cases are retrieved and adapted by taking a weighted majority vote of the quadrants, using inverse distance from the target problem as weight. The ”winning” quadrant is said to be the quadrant of the unknown problem. Of course, CBR is not at all necessary for this domain, but it provides a very simple first example.

In the CCBR version, the user is first asked for the x coordinate. A preliminary list of similar cases is presented. The user is then asked for the y coordinate. The preliminary list of similar cases is then pared down further based on this complete information.

The Quadrant example source code includes the following:

- QuadrantTestClass.java - contains a main method
- QuadrantSystem.java - defines the domain-dependent subclass of `CBRSystem`
- QuadrantDomain.java - defines some details of the domain not defined in the standard `Domain` class

The QuadrantCCBR example source code includes the following:

- QuadrantCCBRTestClass.java - contains a main method
- QuadrantCCBRSystem.java - defines the domain-dependent subclass of `CCBRSystem`
- QuadrantCCBRDomain.java - defines some details of the domain not defined in the standard `CCBRDomain` class

We begin with `QuadrantTestClass`. This demonstrates at a high level the ease of using the basic functionality for a CBR system built with the framework. The class consists entirely of a `main` method:

```
public class QuadrantTestClass {
  public static void main(String[] args) {
    ....
  }
}
```

The first step is to construct a quadrant system. It is put in “GUI Mode” so that a GUI presentation of any results are shown.

```
    QuadrantSystem sys = new QuadrantSystem();
    sys.setGUIMode(true);
```

A problem description is needed to test the system. The simplest way to generate problems is to use a random problem generator, which is defined in the `QuadrantSystem` class. We then give the problem to the system to solve. Because we turned on GUI mode, a display of the results will appear:

```
    Problem p = sys.constructProblemGenerator().generateProblem();
    Solution s = sys.solve(p);
```

We choose to add the problem and resulting solution, as a new case, to the case base:

```
sys.getCB().addCase(new Case(p, s));
```

Finally, Java requires that we explicitly call `exit` because a separate window was displayed:

```
System.exit(0);
```

This is all that is required for a basic use of the quadrant system.

QuadrantCCBRTestClass is slightly simpler, because no generation of a problem is required. The main method consists entirely of:

```
QuadrantCCBRSystem sys = new QuadrantCCBRSystem();
sys.setGUIMode(true);
sys.initiateConversation();
```

The call to the standard `initiateConversation` method takes over, presenting a GUI and guiding the CCBR process.

Next we consider what happens within the `QuadrantSystem` class. `QuadrantSystem` extends `CBRSystem`, and so most functionality comes from that class. The constructor begins with a call to the locally-defined `setupDomain` method, which begins by constructing a `QuadrantDomain` object, to be discussed momentarily:

```
QuadrantDomain domain = new QuadrantDomain();
```

Next we tell the domain what problem features will be used. We also set two private instance variables, the `FeatureKey` objects `xKey` and `yKey`, which are used internally by this system to obtain the x and y coordinates of a case. The arguments to `addProblemFeature` are the name of the feature, the type (as a `String`), its weight in the similarity measure, and whether or not it is an index (to be used for retrieval).

```
xKey = domain.addProblemFeature("x",
                                "edu.indiana.iucbrf.feature.DoubleFeature",
                                1,
                                true);
yKey = domain.addProblemFeature("y",
                                "edu.indiana.iucbrf.feature.DoubleFeature",
                                1,
                                true);
```

We also register the one solution feature required by this domain, which indicates the quadrant:

```
domain.addSolutionFeature("Quadrant",
                          "edu.indiana.iucbrf.feature.IntegerFeature");
```

We set up the reference method (see discussion on page 16 concerning the reference method) for this system. This method merely initializes the `xKey` and `yKey` private instance variables in the `QuadrantDomain` class.

```
domain.prepareReferenceSolution(xKey, yKey);
```

Finally, we register the constructed domain with this system instance:

```
setDomain(domain);
```

The Domain object is now prepared.

Next, the QuadrantSystem constructor calls the locally-defined setupCaseBase method. The first step is to construct a case generator so the case base can be randomly populated:

```
CaseGenerator cg = new CaseGenerator(constructProblemGenerator(), getDomain());
```

The local constructProblemGenerator method simply maps each feature key to a distribution for random generation of that feature.

```
public ProblemGenerator constructProblemGenerator() {
  HashMap argMap = new HashMap(2);
  argMap.put(xKey, new UniformDistribution(-1, 1));
  argMap.put(yKey, new UniformDistribution(-1, 1));

  return new ProblemGenerator(argMap, getDomain());
}
```

The final step of setupCaseBase is to construct and register the case base with the system. A simple flat case base structure is used here:

```
setCB(new FlatCaseBase(cg, NUM_CASES));
```

The remaining steps of the QuadrantSystem constructor are simple. There will be no maintenance in this simple system:

```
setMaintenance(new NullMaintenance());
```

The retrieval will be 5-nearest neighbor. The problem differentiation technique is defined in the domain:

```
Retrieval retrieval = new kNN(getDomain().getProblemDifferentiator(), 5);
```

We would like to keep track not only of the top 5 cases, but also their differences from the current problem:

```
retrieval.setTrackDifferences(true);
```

We register the retrieval algorithm with the system:

```
setRetrieval(retrieval);
```

Adaptation will be a weighted majority vote, with closer cases' votes more significant:

```
setAdaptation(
  new DistanceWeightedAdapter(
    "edu.indiana.iucbrf.adaptation.WeightedMajorityAdapter"));
```

A basic performance monitor will track system activity:

```
setPerformanceMonitor(new PerformanceMonitor());
```

This completes the construction of the QuadrantSystem object.

The differences in preparing the QuadrantCCBRSystem object are minor. Instead of extending CBRSystem, it extends CCBRSystem. In addition, the QuadrantCCBRDomain is used instead of QuadrantDomain. The question associated with a feature must be specified along with the other feature information. So in setupDomain:

```
QuadrantCCBRDomain domain = new QuadrantCCBRDomain();
xKey = domain.addProblemFeature("x",
                                "What is the x coordinate?",
                                "edu.indiana.iucbrf.feature.DoubleFeature",
                                1,
                                true);
yKey = domain.addProblemFeature("y",
                                "What is the y coordinate?",
                                "edu.indiana.iucbrf.feature.DoubleFeature",
                                1,
                                true);
...
```

All other steps are identical in setting up the domain for `QuadrantCCBRSystem`.

Although the `kNN` retrieval class could be used for the QuadrantCCBR example as well, it makes a little more sense to use a threshold retrieval, where any case farther than 0.1 away is eliminated. Thus, in the `QuadrantCCBRSystem` constructor:

```
Retrieval retrieval =
  new ThresholdRetrieval(getDomain().getProblemDifferentiator(),
                         0.1);
```

There are also two additional components to set in the `QuadrantCCBRSystem`, the conversation block selector and the case list refiner. The `OrderedConversationBlockSelector` instantiated below will present questions (corresponding to features) in the order that they were added to the domain via `addProblemFeature` (called in the `setupDomain` method). The `ThresholdCaseListRefiner` will refine the case list originally obtained by the `ThresholdRetrieval` object in the first step of the conversation, when new information is entered into the system by the user.

```
setConversationBlockSelector(
   new OrderedConversationBlockSelector((CCBRDomain) getDomain()));
setCaseListRefiner(new ThresholdCaseListRefiner(0.1));
```

All that remains is to examine `QuadrantDomain` and `QuadrantCCBRDomain`, which are identical except for class name, and that `QuadrantDomain` extends `Domain`, while `QuadrantCCBRDomain` extends `CCBRDomain`. By these extensions, most functionality is defined in the super class. All that remains is to define the reference method and the means of determining the quality of a solution.

Recall that the reference method is used for testing the system. It determines the solution in the non-CBR way, here by examining the signs of the coordinates. We first obtain double representations of the values of the two problem features:

```
public Solution getReferenceSolution(Problem p) {
  double xVal = ((Double)p.getFeature(xKey).getValue()).doubleValue();
  double yVal = ((Double)p.getFeature(yKey).getValue()).doubleValue();
```

Next we determine the quadrant using if statements:

```
  int solutionVal;
  if (yVal > 0) {
    if (xVal > 0)
      solutionVal = 1;
    else
      solutionVal = 2;
  } else {
    if (xVal > 0)
```

```
        solutionVal = 4;
      else
        solutionVal = 3;
    }
```

We construct a Solution object using the Domain object's ComponentFactory:

```
Solution result = getComponentFactory().constructSolution(solutionVal);
```

We indicate that this solution was obtained via a reference method, and not, for example, by adaptation in CBR:

```
result.setIsReferenceSolution(true);
```

The solution object is returned:

```
return result;
```

Finally, the solution quality object determines whether a problem was "solved well" according to the domain's specifications. In the QuadrantDomain, a simple equality check with the expected solution (via the reference method) is done.

```
public SolutionQuality determineSolutionQuality(Problem p,
                                                Solution s,
                                                Solution expectedSolution) {
  boolean solvedWell;

  if (s.equals(expectedSolution))
    solvedWell = true;
  else
    solvedWell = false;

  return new SolutionQuality(solvedWell, solvedWell ? 1 : 0);
}
```

See the general Domain discussion above for more information on solution quality criteria.

This is all that is required to use IUCBRF to make a simple CBR system for the quadrant domain. All other functionality and options for various components are provided by IUCBRF.

7 Running the Quadrant Example

Running the quadrant example is very simple. With the CLASSPATH pointing to the directory containing the edu directory, run

java edu.indiana.iucbrf.examples.quadrant.QuadrantTestClass

from any directory. All other examples are run in a similar manner.

8 Components Under Development

IUCBRF is an ongoing effort. This section describes some components that are under development but not yet fully robust and complete. The domain generation and exploration, and d-tree indexing components are not included in the distributed system. Preliminary implementations of database connectivity and performance monitoring components are included in the distributed system.

8.1 Domain Generation And Exploration

In (Aha 1992), David Aha suggests that a domain can be considered a single point in a *domain space* (or in his terminology, a *database characterization space*), with its location determined by attributes such as the number and type of features available in a problem description, the format of a solution, and the underlying real-world interactions of the domain. Aha goes on to describe an exploration of this domain space, for the purpose of characterizing how different types of domains affect system performance.

In IUCBRF, the concrete specification of a domain is suitable for generating points in the domain space, and conducting exploration of the domain space, and some code has been developed towards supporting such exploration, in either a random or systematic fashion.

8.2 D-Tree Indexing Of A Case Base

As alluded to in 5.2.3, additional indexing approaches are under development. One such approach is d-tree-based, in which each non-leaf node asks a question (corresponding to a feature) that ultimately leads to a leaf (equivalence class) of cases. A d-tree provides a fast way (without having to examine all cases) to get to a subset of the case base that is theoretically most likely to be useful for a given problem. Three ways to obtain a d-tree indexing are under development:

1. Complete, explicit specification
2. Induction using ID3 (Quinlan 1993)
3. ID3 and multi-interval discretization of continuous-valued features (Fayyad & Irani 1993)

8.3 Database Connectivity

Storing cases in a database rather than solely in memory can provide data persistence after system shutdown, and can ease memory requirements. Some database facilities related to data persistence are already implemented. For example, the domain description can be stored and reloaded in a manner largely transparent to the system designer, though some issues of convenience remain to be resolved.

Of course, an even more important use of a database is to avoid the unreasonable requirement of storing all cases in memory, or even in B-tree-backed files during system operation. This functionality is currently being developed and refined using a complex and data-intensive domain.

Database connectivity can also aid in more flexible and efficient indexing of cases, as allowed by a database management system's querying capabilities.

8.4 Performance Monitoring

As discussed in 5.1.5, the performance monitor tracks the performance of the CBR system in several categories. One feature of this tool under development is a graphical display of a performance report. Potential future features of this report include the ability to view trends in performance via plots, as well as view visualizations of the case base.

9 Related Work

As discussed in (Jaczynski & Trousse 1998), traditional CBR tools are applications, often suitable for non-programmers, with which specific classes of CBR systems can be developed. If the functionality provided by such a tool is suitable for the designer's goals, then these tools can be very useful in aiding that task. These tools often allow limited customization by setting parameters, and may include some code libraries for connecting the tool to a larger system. However, even these code libraries typically do not allow complete customization of the system, allowing instead access only to the system's inputs and outputs, and so it may not be possible to build some envisioned CBR systems with such a tool.

In contrast, IUCBRF is aimed at programmers, although for basic systems the programming requirements are not large, even to build a more traditional CBR tool— the underlying mechanisms for system definition and operation are already in place. Analogous to traditional CBR tools, IUCBRF is customizable by parameter setting in the code. However, because it is open source, it also allows complete flexibility. Designer extensions can be made at any point, not merely at inputs and outputs or via parameter settings.

NaCoDAE (Breslow & Aha 1998) is an example of a more traditional CBR tool. It is not primarily a code library but rather an application suitable for activities including case library creation and editing, case, feature, and solution browsing, problem solving using CCBR, and collection and examination of problem solving history for a system. Behavior of the system is customizable in some ways via the setting of various parameters within the application, including display preferences, selection from provided similarity metrics, and thresholds governing the question ratings, similarity ratings, and the definition of "equality".

CBR*Tools, described in (Jaczynski & Trousse 1998), is an object-oriented library for CBR system development in the same spirit as IUCBRF. It is designed as a framework with the same philosophy towards code reuse and extension of abstract classes, which can be done in case representation, case storage, retrieval, and adaptation. Based on additional documentation in (Trousse 2001), it appears that CBR*Tools may have more predefined retrieval techniques, including not only k-NN but also k-d trees, prototypes filtering, and hash tables. While IUCBRF certainly allows for extensions of these types, they are not currently implemented. However, IUCBRF provides a number of capabilities (discussed in this document) beyond those documented in CBR*Tools, including:

- Explicit domain representation, to assist in maintaining a separation between various components, allowing implementation of domain-independent components, and allowing dynamic changes to the domain description

- Structured feature representation in custom collection classes

- General feature specification representation

- Explicit CCBR support and CCBR-specific functionality

- Implementations of system maintenance, performance monitoring, and testing facilities

- Preimplemented standard GUI components

JColibri (Bello-Tomás, González-Calero, & Díaz-Agudo 2004) is another tool with similarities to CBR*Tools and IUCBRF. It too supports code reuse and independence from the domain through object-oriented design. An interesting feature of JColibri is the provision of a GUI to guide the construction of a CBR system when only pre-defined components are used. JColibri documentation also discusses facilities for many of the same components as IUCBRF corresponding to the CBR cycle, but does not mention facilities for CCBR, maintenance, performance monitoring, experimentation, random generation, or standard GUI components (beyond the GUI for system construction).

Other work includes CASUEL, a common case representation language described in (Manago *et al.* 1994). It provides a standard representation and interface for a case base. The cases are represented not as feature lists but rather as a class hierarchy. A goal of CASUEL was to ultimately become the European standard for exchanging case bases.

As discussed previously, one of the aims of IUCBRF is to assist in education of CBR. CBR microprograms ((Riesbeck & Schank 1989; Schank, Riesbeck, & Kass 1994); code available at (Leake 2002)) also provide code developed for that purpose. These microprograms are miniature versions of CBR research projects, intended to convey the major ideas of the research while avoiding the complexity of the details. They can be used to study both the fundamental approaches and the associated research projects, and can be modified by interested individuals—or in homework assignments—to behave in different ways or use different algorithms. This is very similar to the way IUCBRF may be used to study the effects of different algorithms on the CBR process. Of course, these programs are not intended to be foundations for full CBR systems as IUCBRF is. They illustrate key points clearly, but their restrictions may be a source of frustration to some students.

10 Conclusion

IUCBRF is designed to help users exploit CBR components that are independent from their domains and from other components. We expect this to greatly reduce the code that must be written by the designer to build a CBR system. With many standard techniques for various CBR components already implemented, the framework provides a basis that we expect will enable designers to often simply plug in the desired technique, or extend framework classes in order to implement a few simple domain-dependent procedures.

IUCBRF is easily extensible, so that other general techniques can be implemented for future systems. Each component package contains a base component class which outlines the general functionality required by any class of that component type. In addition, each IUCBRF component makes minimal assumptions about the details of the other components. This allows the CBR system designer to swap in and out various components with minimal change to other domain-dependent code. This facilitates experimentation and comparison of various techniques for a component. At times a particular domain may require very domain-dependent implementations of certain components. In this case, the designer can easily extend a component in a domain-dependent manner while maintaining the required interface with other components.

IUCBRF is an ongoing effort. While it has been used extensively in a number of projects, resulting in systems which appear to be quite stable, it is likely that as it is used more extensively for a variety of projects, additional areas for improvement will be discovered. The framework leaves opportunity for enhancement without significant structural change, and we hope that it will prove a useful tool for researchers, educators, and students.

11 Acknowledgments

The authors would like to thank Thomas Thomas, Neil Briscombe, and Peter Siniakov for contributions to the IUCBRF code, Alejandro Valerio for advice on database code enhancements, and Alex Breuer for helpful comments on a draft of this document.

A IUCBRF License

IUCBRF is released under the Open Source License (OSL), and may be used under the conditions of that license, provided in appendix A. Parties who wish to use IUCBRF without the restrictions of the OSL may contact the Indiana University Research and Technology Transfer Corporation (http://iurtc.iu.edu/) to arrange alternative licensing arrangements.

> *We ask that all publications describing results obtained using IUCBRF code acknowledge use of the framework and cite this technical report.*

The IUCBRF license is printed below.

3) Grant of Source Code License. The term "Source Code" means the preferred form of the Original Work for making modifications to it and all available documentation describing how to modify the Original Work. Licensor hereby agrees to provide a machine-readable copy of the Source Code of the Original Work along with each copy of the Original Work that Licensor distributes. Licensor reserves the right to satisfy this obligation by placing a machine-readable copy of the Source Code in an information repository reasonably calculated to permit inexpensive and convenient access by You for as long as Licensor continues to distribute the Original Work, and by publishing the address of that information repository in a notice immediately following the copyright notice that applies to the Original Work.

4) Exclusions From License Grant. Neither the names of Licensor, nor the names of any contributors to the Original Work, nor any of their trademarks or service marks, may be used to endorse or promote products derived from this Original Work without express prior written permission of the Licensor. Nothing in this License shall be deemed to grant any rights to trademarks, copyrights, patents, trade secrets or any other intellectual property of Licensor except as expressly stated herein. No patent license is granted to make, use, sell or offer to sell embodiments of any patent claims other than the licensed claims defined in Section 2. No right is granted to the trademarks of Licensor even if such marks are included in the Original Work. Nothing in this License shall be interpreted to prohibit Licensor from licensing under different terms from this License any Original Work that Licensor otherwise would have a right to license.

5) External Deployment. The term "External Deployment" means the use or distribution of the Original Work or Derivative Works in any way such that the Original Work or Derivative Works may be used by anyone other than You, whether the Original Work or Derivative Works are distributed to those persons or made available as an application intended for use over a computer network. As an express condition for the grants of license hereunder, You agree that any External Deployment by You of a Derivative Work shall be deemed a distribution and shall be licensed to all under the terms of this License, as prescribed in section 1(c) herein.

6) Attribution Rights. You must retain, in the Source Code of any Derivative Works that You create, all copyright, patent or trademark notices from the Source Code of the Original Work, as well as any notices of licensing and any descriptive text identified therein as an "Attribution Notice." You must cause the Source Code for any Derivative Works that You create to carry a prominent Attribution Notice reasonably calculated to inform recipients that You have modified the Original Work.

7) Warranty of Provenance and Disclaimer of Warranty. Licensor warrants that the copyright in and to the Original Work and the patent rights granted herein by Licensor are owned by the Licensor or are sublicensed to You under the terms of this License with the permission of the contributor(s) of those copyrights and patent rights. Except as expressly stated in the immediately proceeding sentence, the Original Work is provided under this License on an "AS IS" BASIS and WITHOUT WARRANTY, either express or implied, including, without limitation, the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF THE ORIGINAL WORK IS WITH YOU. This DISCLAIMER OF WARRANTY constitutes an essential part of this License. No license to Original Work is granted hereunder except under this disclaimer.

8) Limitation of Liability. Under no circumstances and under no legal

theory, whether in tort (including negligence), contract, or otherwise, shall the Licensor be liable to any person for any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or the use of the Original Work including, without limitation, damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses. This limitation of liability shall not apply to liability for death or personal injury resulting from Licensor’s negligence to the extent applicable law prohibits such limitation. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so this exclusion and limitation may not apply to You.

9) Acceptance and Termination. If You distribute copies of the Original Work or a Derivative Work, You must make a reasonable effort under the circumstances to obtain the express assent of recipients to the terms of this License. Nothing else but this License (or another written agreement between Licensor and You) grants You permission to create Derivative Works based upon the Original Work or to exercise any of the rights granted in Section 1 herein, and any attempt to do so except under the terms of this License (or another written agreement between Licensor and You) is expressly prohibited by U.S. copyright law, the equivalent laws of other countries, and by international treaty. Therefore, by exercising any of the rights granted to You in Section 1 herein, You indicate Your acceptance of this License and all of its terms and conditions. This License shall terminate immediately and you may no longer exercise any of the rights granted to You by this License upon Your failure to honor the proviso in Section 1(c) herein.

10) Termination for Patent Action. This License shall terminate automatically and You may no longer exercise any of the rights granted to You by this License as of the date You commence an action, including a cross-claim or counterclaim, against Licensor or any licensee alleging that the Original Work infringes a patent. This termination provision shall not apply for an action alleging patent infringement by combinations of the Original Work with other software or hardware.

11) Jurisdiction, Venue and Governing Law. Any action or suit relating to this License may be brought only in the courts of a jurisdiction wherein the Licensor resides or in which Licensor conducts its primary business, and under the laws of that jurisdiction excluding its conflict-of-law provisions. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any use of the Original Work outside the scope of this License or after its termination shall be subject to the requirements and penalties of the U.S. Copyright Act, 17 U.S.C. ???? 101 et seq., the equivalent laws of other countries, and international treaty. This section shall survive the termination of this License.

12) Attorneys Fees. In any action to enforce the terms of this License or seeking damages relating thereto, the prevailing party shall be entitled to recover its costs and expenses, including, without limitation, reasonable attorneys’ fees and costs incurred in connection with such action, including any appeal of such action. This section shall survive the termination of this License.

13) Miscellaneous. This License represents the complete agreement concerning the subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable.

14) Definition of "You" in This License. "You" throughout this

B Samples Of Source Code

The complete source code for IUCBRF is not provided here. The purpose of the source code in the following sections is to demonstrate to the reader the types of steps to be done to create a CBR system with IUCBRF as the foundation. Included is a minimal system for a quadrant-determining domain, a conversational version of the same system, and a slightly more complex system for a realtor property-price estimation domain. Comments are provided in the code.

At time of writing, the examples of IUCBRF code are up to date. However, they may be superseded as the framework is developed, and as later examples of this document are prepared. For the most up to date examples (and additional examples), please see the framework code itself.

B.1 Quadrant Example Source Code

The Quadrant example source code includes the following:

- QuadrantTestClass.java - contains a main method
- QuadrantSystem.java - defines the domain-dependent subclass of `CBRSystem`
- QuadrantDomain.java - defines some details of the domain not defined in the standard `Domain` class
- QuadrantLeaveOneOutTestClass.java - another main method, this one running a leave-one-out test on the quadrant system

```
/******************************************************************************
 *                        Initial Version By: Steven Bogaerts                 *
 *                        Copyright (c) 2001 IUCBRF Group                     *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                  *
 * sbogaert@cs.indiana.edu                                                    *
 ******************************************************************************/

/*
 * QuadrantTestClass.java
 *
 * Created on October 20, 2003, 2:26 PM
 */
package edu.indiana.iucbrf.examples.quadrant;

import edu.indiana.iucbrf.problem.Problem;
import edu.indiana.iucbrf.solution.Solution;
import edu.indiana.iucbrf.casepackage.Case;

/**
 * This class contains a simple main method that constructs a QuadrantSystem
 * and tests it on a single problem.
 *
 * @author  Steven Bogaerts
 */
public class QuadrantTestClass {
    /** Constructs the system, and tests it on a single problem. */
    public static void main(String[] args) {
        System.out.println("Hello World!");

        // Construct the system.
        QuadrantSystem sys = new QuadrantSystem();

        // Set the system to show a GUI of its status.
        sys.setGUIMode(true);

        /* Generate a problem to test the system on.  This problem generator is
        defined in the QuadrantSystem class. */
        Problem p = sys.constructProblemGenerator().generateProblem();

        // Solve the problem.  The GUI will show what happens.
        Solution s = sys.solve(p);

        // Add the new case to the case base.
        sys.getCB().addCase(new Case(p, s));

        System.exit(0);
    }
}
```

```
/******************************************************************************
 *                       Initial Version By: Steven Bogaerts                  *
 *                       Copyright (c) 2001 IUCBRF Group                      *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                  *
 * sbogaert@cs.indiana.edu                                                    *
 ******************************************************************************/

/*
 * QuadrantSystem.java
 *
 * Created on October 20, 2003, 3:12 PM
 */
package edu.indiana.iucbrf.examples.quadrant;

import edu.indiana.iucbrf.feature.FeatureKey;
import edu.indiana.iucbrf.maintenance.NullMaintenance;
import edu.indiana.iucbrf.retrieval.Retrieval;
import edu.indiana.iucbrf.retrieval.kNN;
import edu.indiana.iucbrf.adaptation.DistanceWeightedAdapter;
import edu.indiana.iucbrf.performancemonitor.PerformanceMonitor;
import edu.indiana.iucbrf.problem.ProblemGenerator;
import edu.indiana.iucbrf.casepackage.CaseGenerator;
import edu.indiana.iucbrf.casebase.FlatCaseBase;
import edu.indiana.iucbrf.cbrsystem.CBRSystem;
import edu.indiana.util.distribution.UniformDistribution;
import java.util.HashMap;

/**
 * This system is intended to be one of the simplest possible uses of the framework.
 * In this domain, problems contain two features, x and y, both doubles in [-1, 1].
 * The square on the x-y plane corresponding to this problem space is divided into
 * four equal quadrants by the coordinate axes.  These quadrants are numbered 1,
 * 2, 3, and 4, counterclockwise from upper right, as in basic algebra.
 * <BR><BR>
 * A case consists of a problem description and solution, where the problem is
 * a pair (x,y), and the solution is its quadrant (1, 2, 3, or 4).  When a new
 * problem is presented to the system, the five nearest cases are retrieved, and
 * adapted by taking a weighted (by distance) majority vote of the quadrants.
 * The "winning" quadrant is said to be the quadrant of the unknown problem.
 * <BR><BR>
 * See comments in code for additional details.
 *
 * @author  Steven Bogaerts
 */
public class QuadrantSystem
    extends CBRSystem {

    /** The number of cases to be in the case base.  Of course, performance of
     *  the system depends on how many cases are available.
     */
    private static final int NUM_CASES = 200;

    /** The keys to the two problem features. */
    private FeatureKey xKey;
    private FeatureKey yKey;

    /** Creates a new instance of QuadrantSystem */
    public QuadrantSystem() {
        // Define the domain, which describes the type of problems the system will solve.
        setupDomain();

        // Create the case base and add cases to it.
        setupCaseBase();

        // Do no maintenance for this system
        setMaintenance(new NullMaintenance());
        // Retrieval will be 5-nearest neighbor
        Retrieval retrieval = new kNN(getDomain().getProblemDifferentiator(), 5);

        /* Keep track not only of the cases, but also of their differences from
         * the current problem.  This information will be used in adaptation.
         */
        retrieval.setTrackDifferences(true);
        setRetrieval(retrieval);

        // The adaptation technique is a distance-weighted majority vote.
        setAdaptation(
            new DistanceWeightedAdapter("edu.indiana.iucbrf.adaptation.WeightedMajorityAdapter"));

        // Track the system performance with the performance monitor
        setPerformanceMonitor(new PerformanceMonitor());
    }

    // Define the domain, which describes the type of problems the system will solve.
    private void setupDomain() {
        QuadrantDomain domain = new QuadrantDomain();

        /* Add the two problem features: x and y.  The textual description,
         * type, and similarity weight are specified.  The return values are keys
         * that will be referred to in additional setup steps.
         */
        xKey = domain.addProblemFeature("x", "edu.indiana.iucbrf.feature.DoubleFeature", 1, true);
        yKey = domain.addProblemFeature("y", "edu.indiana.iucbrf.feature.DoubleFeature", 1, true);

        /* Add the solution feature: quadrant.  A key is assigned, but it is not
         * needed at this time.
         */
        domain.addSolutionFeature("Quadrant", "edu.indiana.iucbrf.feature.IntegerFeature");

        /* The reference solution for this domain requires the xKey and yKey.  See
        introductory paper for details on reference methods. */
        domain.prepareReferenceSolution(xKey, yKey);

        setDomain(domain);
    }

    /** The case base must be populated.  For this example, random generation is
     *  possible and sufficient.
     */
    private void setupCaseBase() {
        // The case generator can be built from the problem generator.
        CaseGenerator cg = new CaseGenerator(constructProblemGenerator(), getDomain());

        /** Construct the case base, with a simple flat structure.  For this
         * constructor, the case generator and the desired number of initial cases
         * can be passed.
         */
        setCB(new FlatCaseBase(cg, NUM_CASES));
    }

    /**  A problem generator requires specification of arguments to send to
     * feature constructors.  Here, random distributions are used.
     */
    public ProblemGenerator constructProblemGenerator() {
        HashMap argMap = new HashMap(2);
        argMap.put(xKey, new UniformDistribution(-1, 1));
        argMap.put(yKey, new UniformDistribution(-1, 1));

        return new ProblemGenerator(argMap, getDomain());
    }
}
```

```
/*******************************************************************************
 *                      Initial Version By: Steven Bogaerts                    *
 *                      Copyright (c) 2001 IUCBRF Group                        *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                   *
 * sbogaert@cs.indiana.edu                                                     *
 *******************************************************************************/

/*
 * QuadrantDomain.java
 *
 * Created on October 20, 2003, 2:51 PM
 */
package edu.indiana.iucbrf.examples.quadrant;

import edu.indiana.iucbrf.problem.Problem;
import edu.indiana.iucbrf.solution.Solution;
import edu.indiana.iucbrf.feature.FeatureKey;
import edu.indiana.iucbrf.domain.Domain;
import edu.indiana.iucbrf.performancemonitor.SolutionQuality;

/**
 * This class extends the standard Domain class, so that a custom reference method
 * can be created for the Quadrant system.  See introductory paper for details
 * on reference methods.
 *
 * @author  Steven Bogaerts
 */
public class QuadrantDomain
    extends Domain {

    /** The feature keys for the x and y problem features. */
    private FeatureKey xKey;
    private FeatureKey yKey;

    /** Prepare the reference solution for use.  This is done by recording the
     *  feature keys needed.
     */
    public void prepareReferenceSolution(FeatureKey xKey,
                                         FeatureKey yKey) {
        this.xKey = xKey;
        this.yKey = yKey;
    }

    /** This method overrides the one involving equivalence classes
     *  in Domain.  It returns the solution to the problem sent to it.
     */
    public Solution getReferenceSolution(Problem p) {

        /* Get the double values.  The DoubleFeature contains a Double, which
         * contains a double.
         */
        double xVal = ((Double)p.getFeature(xKey).getValue()).doubleValue();
        double yVal = ((Double)p.getFeature(yKey).getValue()).doubleValue();

        // Determine the quadrant.
        int solutionVal;

        if (yVal > 0) {
            if (xVal > 0)
                solutionVal = 1;
            else
                solutionVal = 2;
        } else {
            if (xVal > 0)
                solutionVal = 4;
            else
```

```
                solutionVal = 3;
        }

        // Use Domain.constructSolution(double) to build a Solution.
        Solution result = getComponentFactory().constructSolution(solutionVal);

        // The solution is one that was generated by a reference method.
        result.setIsReferenceSolution(true);

        return result;
    }

    /** Overrides Domain.determineSolutionQuality(), to have very strict ratings requirements.
     */
    public SolutionQuality determineSolutionQuality(Problem p, Solution s, Solution expectedSolu-
tion) {
        boolean solvedWell;

        if (s.equals(expectedSolution))
            solvedWell = true;
        else
            solvedWell = false;

        // The rating is 1 if the solutions are equal, 0 otherwise.
        return new SolutionQuality(solvedWell, solvedWell
                                                ? 1
                                                : 0);
    }
}
```

```
/*****************************************************************************
 *                      Initial Version By: Steven Bogaerts                  *
 *                      Copyright (c) 2001 IUCBRF Group                      *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                 *
 * sbogaert@cs.indiana.edu                                                   *
 *****************************************************************************/

/*
 * QuadrantLeaveOneOutTestClass.java
 *
 * Created on August 13, 2004, 11:18 AM
 */
package edu.indiana.iucbrf.examples.quadrant;

import edu.indiana.iucbrf.performancemonitor.PerformanceMonitor;
import edu.indiana.iucbrf.systemtest.LeaveOneOutTest;

/** The class contains a simple main method to run a leave-one-out test on
 *  a quadrant system.
 *
 * @author Steven Bogaerts
 */
public class QuadrantLeaveOneOutTestClass {
    public static void main(String[] args) {
        // Construct the system.
        QuadrantSystem sys = new QuadrantSystem();

        // Create a LeaveOneOutTest instance and run test().
        PerformanceMonitor results = (new LeaveOneOutTest()).test(sys);
        // The results can then be examined with the performance monitor.
        // See PerformanceMonitor class for details.
    }
}
```

B.2 QuadrantCCBR Example Source Code

The QuadrantCCBR example source code includes the following:

- QuadrantCCBRTestClass.java - contains a main method
- QuadrantCCBRSystem.java - defines the domain-dependent subclass of `CBRSystem`
- QuadrantCCBRDomain.java - defines some details of the domain not defined in the standard `CCBRDomain` class

```
/****************************************************************************
 *                       Initial Version By: Steven Bogaerts                  *
 *                       Copyright (c) 2001 IUCBRF Group                      *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                  *
 * sbogaert@cs.indiana.edu                                                    *
 ****************************************************************************/

/*
 * QuadrantCCBRTestClass.java
 *
 * Created on October 20, 2003, 2:26 PM
 */
package edu.indiana.iucbrf.examples.quadrantCCBR;

import edu.indiana.iucbrf.problem.Problem;
import edu.indiana.iucbrf.solution.Solution;
import edu.indiana.iucbrf.casepackage.Case;

/**
 * This class contains a simple main method that constructs a QuadrantSystem
 * and tests it on a single problem.
 *
 * @author  Steven Bogaerts
 */
public class QuadrantCCBRTestClass {
    /** Constructs the system, and tests it on a single problem. */
    public static void main(String[] args) {
        System.out.println("Hello World!");

        // Construct the system.
        QuadrantCCBRSystem sys = new QuadrantCCBRSystem();

        // Set the system to show a GUI of its status.
        sys.setGUIMode(true);

        /* Begin the conversation.  When all questions have been asked, the remaining
         * cases are displayed.
         */
        sys.initiateConversation();
    }
}
```

```
/******************************************************************************
 *                      Initial Version By: Steven Bogaerts                   *
 *                      Copyright (c) 2001 IUCBRF Group                       *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                  *
 * sbogaert@cs.indiana.edu                                                    *
 ******************************************************************************/

/*
 * QuadrantCCBRSystem.java
 *
 * Created on October 20, 2003, 3:12 PM
 */
package edu.indiana.iucbrf.examples.quadrantCCBR;

import edu.indiana.iucbrf.feature.FeatureKey;
import edu.indiana.iucbrf.maintenance.NullMaintenance;
import edu.indiana.iucbrf.retrieval.Retrieval;
import edu.indiana.iucbrf.retrieval.kNN;
import edu.indiana.iucbrf.retrieval.ThresholdRetrieval;
import edu.indiana.iucbrf.adaptation.DistanceWeightedAdapter;
import edu.indiana.iucbrf.performancemonitor.PerformanceMonitor;
import edu.indiana.iucbrf.problem.ProblemGenerator;
import edu.indiana.iucbrf.casepackage.CaseGenerator;
import edu.indiana.iucbrf.casebase.FlatCaseBase;
import edu.indiana.iucbrf.cbrsystem.CCBRSystem;
import edu.indiana.util.distribution.UniformDistribution;
import edu.indiana.iucbrf.ccbr.conversationblock.selector.OrderedConversationBlockSelector;
import edu.indiana.iucbrf.ccbr.caselistrefiner.ThresholdCaseListRefiner;
import edu.indiana.iucbrf.domain.CCBRDomain;
import java.util.HashMap;

/**
 * This system is intended to be one of the simplest possible uses of the framework.
 * In this domain, problems contain two features, x and y, both doubles in [-1, 1].
 * The square on the x-y plane corresponding to this problem space is divided into
 * four equal quadrants by the coordinate axes.  These quadrants are numbered 1,
 * 2, 3, and 4, counterclockwise from upper right, as in basic algebra.
 * <BR><BR>
 * A case consists of a problem description and solution, where the problem is
 * a pair (x,y), and the solution is its quadrant (1, 2, 3, or 4).  When a new
 * problem is presented to the system, the five nearest cases are retrieved, and
 * adapted by taking a weighted (by distance) majority vote of the quadrants.
 * The "winning" quadrant is said to be the quadrant of the unknown problem.
 * <BR><BR>
 * See comments in code for additional details.
 *
 * @author  Steven Bogaerts
 */
public class QuadrantCCBRSystem extends CCBRSystem {

    /** The number of cases to be in the case base.  Of course, performance of
     *  the system depends on how many cases are available.
     */
    private static final int NUM_CASES = 200;

    /** The keys to the two problem features. */
    private FeatureKey xKey;
    private FeatureKey yKey;

    /** Creates a new instance of QuadrantCCBRSystem */
    public QuadrantCCBRSystem() {
        // Define the domain, which describes the type of problems the system will solve.
        setupDomain();

        // Create the case base and add cases to it.
        setupCaseBase();
        // Do no maintenance for this system
        setMaintenance(new NullMaintenance());

        /* Although kNN retrieval would work here as in the Quadrant example,
         * ThresholdRetrieval is a little more appropriate for CCBR in this case.
         */
        Retrieval retrieval = new ThresholdRetrieval(getDomain().getProblemDifferentiator(), 0.1);

        /* Keep track not only of the cases, but also of their differences from
         * the current problem.  This information will be used in adaptation.
         */
        retrieval.setTrackDifferences(true);
        setRetrieval(retrieval);

        // The adaptation technique is a distance-weighted majority vote.
        setAdaptation(
            new DistanceWeightedAdapter("edu.indiana.iucbrf.adaptation.WeightedMajorityAdapter"));

        // Track the system performance with the performance monitor
        setPerformanceMonitor(new PerformanceMonitor());

        /* Use a conversation block selector, which determines what will be
         * presented to the user at each step in the conversaiton.
         * The OrderedConversationBlockSelector instantiated below will present
         * questions (corresponding to features) in the order that they were added
         * to the domain via addProblemFeature() (called in setupDomain() below).
         */
        setConversationBlockSelector(
            new OrderedConversationBlockSelector((CCBRDomain) getDomain()));

        /* Set the case list refiner so that, at any step in the conversation,
         * cases farther than 0.1 away from the target case so far are eliminated.
         */
        setCaseListRefiner(new ThresholdCaseListRefiner(0.1));
    }

    // Define the domain, which describes the type of problems the system will solve.
    private void setupDomain() {
        QuadrantCCBRDomain domain = new QuadrantCCBRDomain();

        /* Add the two problem features: x and y.  The textual description,
         * type, and similarity weight are specified.  The return values are keys
         * that will be referred to in additional setup steps.
         */
        xKey = domain.addProblemFeature("x",
                                        "What is the x coordinate?",
                                        "edu.indiana.iucbrf.feature.DoubleFeature",
                                        1,
                                        true);
        yKey = domain.addProblemFeature("y",
                                        "What is the y coordinate?",
                                        "edu.indiana.iucbrf.feature.DoubleFeature",
                                        1,
                                        true);

        /* Add the solution feature: quadrant.  A key is assigned, but it is not
         * needed at this time.
         */
        domain.addSolutionFeature("Quadrant", "edu.indiana.iucbrf.feature.IntegerFeature");

        /* The reference solution for this domain requires the xKey and yKey.  See
        introductory paper for details on reference methods. */
        domain.prepareReferenceSolution(xKey, yKey);

        setDomain(domain);
    }
```

```
    /** The case base must be populated.  For this example, random generation is
     *  possible and sufficient.
     */
    private void setupCaseBase() {
        // The case generator can be built from the problem generator.
        CaseGenerator cg = new CaseGenerator(constructProblemGenerator(), getDomain());

        /** Construct the case base, with a simple flat structure.  For this
         * constructor, the case generator and the desired number of initial cases
         * can be passed.
         */
        setCB(new FlatCaseBase(cg, NUM_CASES));
    }

    /**  A problem generator requires specification of arguments to send to
     * feature constructors.  Here, random distributions are used.
     */
    public ProblemGenerator constructProblemGenerator() {
        HashMap argMap = new HashMap(2);
        argMap.put(xKey, new UniformDistribution(-1, 1));
        argMap.put(yKey, new UniformDistribution(-1, 1));

        return new ProblemGenerator(argMap, getDomain());
    }
}
```

```
/******************************************************************************
 *                      Initial Version By: Steven Bogaerts                    *
 *                      Copyright (c) 2001 IUCBRF Group                        *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                   *
 * sbogaert@cs.indiana.edu                                                      *
 ******************************************************************************/

/*
 * QuadrantCCBRDomain.java
 *
 * Created on October 20, 2003, 2:51 PM
 */
package edu.indiana.iucbrf.examples.quadrantCCBR;

import edu.indiana.iucbrf.problem.Problem;
import edu.indiana.iucbrf.solution.Solution;
import edu.indiana.iucbrf.feature.FeatureKey;
import edu.indiana.iucbrf.domain.CCBRDomain;
import edu.indiana.iucbrf.performancemonitor.SolutionQuality;

/**
 * This class extends the standard Domain class, so that a custom reference method
 * can be created for the Quadrant system.  See introductory paper for details
 * on reference methods.
 *
 * @author  Steven Bogaerts
 */
public class QuadrantCCBRDomain extends CCBRDomain {

    /** The feature keys for the x and y problem features. */
    private FeatureKey xKey;
    private FeatureKey yKey;

    /** Prepare the reference solution for use.  This is done by recording the
     *  feature keys needed.
     */
    public void prepareReferenceSolution(FeatureKey xKey,
                                         FeatureKey yKey) {
        this.xKey = xKey;
        this.yKey = yKey;
    }

    /** This method overrides the one involving equivalence classes
     *  in Domain.  It returns the solution to the problem sent to it.
     */
    public Solution getReferenceSolution(Problem p) {

        /* Get the double values.  The DoubleFeature contains a Double, which
         * contains a double.
         */
        double xVal = ((Double)p.getFeature(xKey).getValue()).doubleValue();
        double yVal = ((Double)p.getFeature(yKey).getValue()).doubleValue();

        // Determine the quadrant.
        int solutionVal;

        if (yVal > 0) {
            if (xVal > 0)
                solutionVal = 1;
            else
                solutionVal = 2;
        } else {
            if (xVal > 0)
                solutionVal = 4;
            else
                solutionVal = 3;
        }

        // Use Domain.constructSolution(double) to build a Solution.
        Solution result = getComponentFactory().constructSolution(solutionVal);

        // The solution is one that was generated by a reference method.
        result.setIsReferenceSolution(true);

        return result;
    }

    /** Overrides Domain.determineSolutionQuality(), to have very strict ratings requirements.
     */
    public SolutionQuality determineSolutionQuality(Problem p, Solution s, Solution expectedSolu-
tion) {
        boolean solvedWell;

        if (s.equals(expectedSolution))
            solvedWell = true;
        else
            solvedWell = false;

        // The rating is 1 if the solutions are equal, 0 otherwise.
        return new SolutionQuality(solvedWell, solvedWell
                                                ? 1
                                                : 0);
    }
}
```

B.3 Realtor Example Source Code

The realtor example source code includes the following:

- RealtorTestClass.java - contains the main method
- RealtorSystem.java - defines the domain-dependent subclass of `CBRSystem`
- RealtorAdapter.java - specifies a simple domain-dependent adaptation scheme
- RealtorMaintenance.java - specifies some basic maintenance policies
- RealtorCaseBaseListener.java - reacts to case base events

```java
/******************************************************************************
 *                       Initial Version By: Steven Bogaerts                  *
 *                       Copyright (c) 2001 IUCBRF Group                      *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                   *
 * sbogaert@cs.indiana.edu                                                     *
 ******************************************************************************/

/*
 * RealtorTestClass.java
 *
 * Created on March 30, 2001, 3:22 PM
 */
package edu.indiana.iucbrf.examples.realtor;

import edu.indiana.iucbrf.solution.Solution;
import edu.indiana.iucbrf.casepackage.CaseGenerator;
import edu.indiana.iucbrf.domain.Domain;
import edu.indiana.iucbrf.domain.componentfactory.ComponentFactory;
import edu.indiana.util.distribution.*;
import edu.indiana.util.doublecollection.DoubleMap;
import edu.indiana.iucbrf.problem.Problem;
import edu.indiana.iucbrf.problem.ProblemGenerator;
import edu.indiana.iucbrf.systemtest.SystemTest;
import edu.indiana.iucbrf.systemtest.RandomProblemTest;
import edu.indiana.iucbrf.systemtest.LeaveOneOutTest;
import edu.indiana.iucbrf.domain.ProblemEquivalenceClass;
import edu.indiana.iucbrf.feature.FeatureKey;
import edu.indiana.iucbrf.cbrsystem.CBRSystem;
import edu.indiana.iucbrf.casepackage.Case;
import edu.indiana.iucbrf.feature.format.FeatureValueFormat;
import edu.indiana.iucbrf.feature.format.NumberFeatureValueFormat;
import java.util.LinkedList;
import java.util.HashMap;
import edu.indiana.util.clustering.ArrayListPartition;
import edu.indiana.util.clustering.Partition;
import edu.indiana.util.clustering.KMedoidClustering;

/**
 * This class can be run to test the functionality of this CBR House Appraisal system.
 * This is a simple example to demonstrate IUCBRF functionality; it is not intended to
 * be a fully realistic realtor application.
 *
 * The "solution" to a case describes the market value of a house, based on five problem
 * features: size, age, local traffic congestion, tax rate, and crime rate. When a problem
 * is presented to the system, the single nearest case is retrieved, and a simple adaptation
 * based on house size is performed.  This result is compared to the reference solution.
 * See Domain class for details of the reference solution.
 *
 * There are several options in running this application, selectable by a simple edit of the
 * main method.  This approach was taken in preference to cluttering this IUCBRF demo with
 * extra GUI code.
 *
 * @author  Steven Bogaerts
 * @version 1.0
 */
public class RealtorTestClass
    extends Object {

    // The number of problem features
    private static final int NUM_PROBLEM_FEATURES = 5;

    // The number of initial cases in the case base
    private static final int NUM_CASES = 200;

    // The keys used to refer to features of any problem
    private static FeatureKey[] problemFeatureKeys;
```

```
// A key of particular interest to RealtorAdapter, pointing to the "house size" feature
public static FeatureKey sizeFeatureKey;

/** Creates new RealtorTestClass */
public RealtorTestClass() {
}

/** Test the functionality of this CBR system
 * @param args the command line arguments
 */
public static void main(String[] args) {
    CBRSystem sys = setup();

    // Set whether or not to provide status information in a GUI format
    sys.setGUIMode(true);

    // Set whether or not to provide status information as textual output to stdout
    sys.setDebugMode(true);

    // Test the system on 1 randomly generated problem
    randomTester(sys, 1);

    // For the following options, it may be wise to set GUI mode (and perhaps DebugMode)
    // to false, to avoid a lot of output.
    // sys.setGUIMode(false);
    // leaveOneOutTester(sys); // Do a leave-one-out test on the case base
    // testCBEvents(sys); // View a demonstration of basic event handling in IUCBRF
    System.exit(0);
}

/** Prepare the system for operation. */
private static CBRSystem setup() {
    ///////////////////////////////////////////////////////////////////////
    // Set up the features that describe a house
    // The types (classes) of each problem feature
    String[] problemFeatureTypes = new String[NUM_PROBLEM_FEATURES];

    // The textual descriptions of each problem feature
    String[] problemFeatureDescriptions = new String[NUM_PROBLEM_FEATURES];

    // The weight to be used in similarity measures, for each problem feature
    double[] problemSimilarityWeights = new double[NUM_PROBLEM_FEATURES];
    boolean[] problemFeatureIsIndex = new boolean[NUM_PROBLEM_FEATURES];

    // Optionally specified feature formatting instructions
    // (for GUIs and terminal output)
    FeatureValueFormat[] problemFeatureValueFormat =
        new FeatureValueFormat[NUM_PROBLEM_FEATURES];

    // Define the relevant data for each problem feature
    // The size of the house is prompted by "Size"
    problemFeatureDescriptions[0] = "Size";
    // This feature is a DoubleFeature (its value is of type double)
    problemFeatureTypes[0] = "edu.indiana.iucbrf.feature.DoubleFeature";
    // Define the similarity weight for this feature
    problemSimilarityWeights[0] = 5.0; // house size is quite important for similarity
    // (Optional) Define a formatting specification
    problemFeatureValueFormat[0] =
        new NumberFeatureValueFormat(new java.text.DecimalFormat("#,###.## ft^2"));
    problemFeatureIsIndex[0] = true;

    // etc, for the remaining problem features
    problemFeatureDescriptions[1] = "Age";
    problemFeatureTypes[1] = "edu.indiana.iucbrf.feature.IntegerFeature";
    problemSimilarityWeights[1] = 3.0; // house age is pretty important too
    problemFeatureValueFormat[1] =
    new NumberFeatureValueFormat(new java.text.DecimalFormat("#,### years"));
problemFeatureIsIndex[1] = true;

problemFeatureDescriptions[2] = "Local traffic congestion";
problemFeatureTypes[2] = "edu.indiana.iucbrf.feature.finiteset.FiveSeverityFeature";
// traffic is less important (but higher weight due to values 0...4)
problemSimilarityWeights[2] = 400.0;
problemFeatureValueFormat[2] = null; // no special formatting for this feature
problemFeatureIsIndex[2] = true;

problemFeatureDescriptions[3] = "Property tax rate";
problemFeatureTypes[3] = "edu.indiana.iucbrf.feature.DoubleFeature";
problemSimilarityWeights[3] = 50.0;
problemFeatureValueFormat[3] =
    new NumberFeatureValueFormat(java.text.NumberFormat.getPercentInstance());
problemFeatureIsIndex[3] = true;

problemFeatureDescriptions[4] = "Violent crime rate, per 1000 people";
problemFeatureTypes[4] = "edu.indiana.iucbrf.feature.DoubleFeature";
problemSimilarityWeights[4] = 2.0;
problemFeatureValueFormat[4] =
    new NumberFeatureValueFormat(new java.text.DecimalFormat("#,###.##"));
problemFeatureIsIndex[4] = true;

// Define the relevant data for the solution features
// The solution is simply the estimated value of the house
String[] solutionFeatureTypes = new String[1];
String[] solutionFeatureDescriptions = new String[1];
FeatureValueFormat[] solutionFeatureValueFormat = new FeatureValueFormat[1];
solutionFeatureDescriptions[0] = "Market value";
solutionFeatureTypes[0] = "edu.indiana.iucbrf.feature.DoubleFeature";
solutionFeatureValueFormat[0] =
    new NumberFeatureValueFormat(
        java.text.NumberFormat.getCurrencyInstance(java.util.Locale.US));

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// Build the domain description
Domain domain = new Domain(0.9);

// Customize names of various objects for this domain
domain.setProblemTitle("House Description");
domain.setSolutionTitle("Result");
domain.setRetrievedCasesTitle("Similar Houses");
domain.setCaseTitleBase("House");

// Add the previously defined features to the domain
// Store the keys the domain assigns
problemFeatureKeys = new FeatureKey[NUM_PROBLEM_FEATURES];

for (int i = 0; i < NUM_PROBLEM_FEATURES; i++)
    problemFeatureKeys[i] =
        domain.addProblemFeature(problemFeatureDescriptions[i],
                                 problemFeatureTypes[i],
                                 problemSimilarityWeights[i],
                                 problemFeatureIsIndex[i],
                                 problemFeatureValueFormat[i]);

// store this key separately, for RealtorAdapter
sizeFeatureKey = problemFeatureKeys[0];

// Add solution features in a manner similar to problem features
FeatureKey[] solutionFeatureKeys = new FeatureKey[1];
solutionFeatureKeys[0] = domain.addSolutionFeature(solutionFeatureDescriptions[0],
                                                   solutionFeatureTypes[0],
                                                   solutionFeatureValueFormat[0]);
        // Set up the equivalence classes for the reference solution for the domain
        // See Domain.java for details on the reference solution.
        LinkedList pecs = new LinkedList();
        ComponentFactory componentFactory = domain.getComponentFactory();

        /* Basically, high values decrease the price of the house, except for the
         * house size (where a high value increases the price of the house)
         * Differences in magnitudes between feature weights reflect the differences
         * in typical magnitudes of those feature values, as well as
         * the difference in importance of the various features in
         * determining the final price of the house.
         */
        pecs.add(new ProblemEquivalenceClass(
            componentFactory.constructProblem(new double[]{13000, 17, 2, 0.2, 40 }),
            new DoubleMap(problemFeatureKeys,
                          new double[]{200, -500, -400, -3000, -100 }),
            domain));
        pecs.add(new ProblemEquivalenceClass(
            componentFactory.constructProblem(new double[]{ 12500, 10, 4, 0.3, 20 }),
            new DoubleMap(problemFeatureKeys,
                          new double[]{ 210, -300, -450, -2500, -150 }),
            domain));
        pecs.add(new ProblemEquivalenceClass(
            componentFactory.constructProblem(new double[]{ 10000, 12, 0, 0.5, 10 }),
            new DoubleMap(problemFeatureKeys,
                          new double[]{ 190, -400, -375, -2700, -125 }),
            domain));

        // Register the equivalence classes with the domain
        domain.setEquivalenceClasses(pecs);

        ////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////
        // Build the system itself
        // Construct a CaseGenerator, to randomly create cases to fill the case base
        CaseGenerator caseGenerator = constructCaseGenerator(domain);

        // Construct the system with the given domain, with NUM_CASES in the case base,
        // to be generated by the CaseGenerator.
        RealtorSystem sys = new RealtorSystem(caseGenerator, domain, NUM_CASES);
        sys.setTitle("House Appraisal");

        return sys;
    }

    /** Demonstrate the leave-one-out functionality. */
    private static void leaveOneOutTester(CBRSystem sys) {
        // Build the system to be tested
        CaseGenerator caseGenerator = constructCaseGenerator(sys.getDomain());

        // Build the tester
        SystemTest tester = new LeaveOneOutTest();

        // Test the system
        tester.test(sys, false);
    }

    /** Demonstrate the handling of events on the case base. */
    private static void testCBEvents(CBRSystem sys) {
        int numTests = 100;
        int iterations = 5;
        Case newCase;
        Problem newProblem;

        // Build the problem generator, to randomly generate problems
        ProblemGenerator tpg = constructTestProblemGenerator(sys.getDomain());
        // Build the system tester
        SystemTest tester = new RandomProblemTest(tpg, numTests);

        // Run the tests
        for (int i = 0; i < iterations; i++) {
            System.out.println("\n==================================================");
            System.out.println("Running the system on 100 problems...");

            // test the system
            tester.test(sys, true);

            // Simulate action that initiates a case base event
            System.out.println("\nA realtor decides to add another case from the real world...");

            // Generate a case to add to the case base
            newProblem = tpg.generateProblem();

            Solution fbsol = sys.getDomain().getReferenceSolution(newProblem);
            newCase = new Case(newProblem, fbsol, sys.getPerformanceMonitor().getSystemAge());

            // Add the case
            sys.getCB().addCase(newCase);
        }
    }

    /** Test the system on some randomly generated problems. */
    private static void randomTester(CBRSystem sys,
                                     int numTests) {

        // Build the problem generator
        ProblemGenerator tpg = constructTestProblemGenerator(sys.getDomain());

        // Build the system tester
        SystemTest tester = new RandomProblemTest(tpg, numTests);

        // Test the system
        tester.test(sys, true);
    }

    /** Construct a CaseGenerator, by specifying the arguments to send to a constructor
     * for each feature.  The constructor to use can be specified in the corresponding
     * FeatureSpec.  Typically (as is the case in all these features), the argument
     * to send is a distribution over which to generate the feature values.  Of course
     * this assumes that there is a reasonable mapping from a double to the feature
     * value required.  (That is, that each feature class in use implements
     * DoubleRepresentable.)
     */
    private static CaseGenerator constructCaseGenerator(Domain domain) {

        // Map arguments to the corresponding features, via the feature keys.
        HashMap argMap = new HashMap(NUM_PROBLEM_FEATURES);
        argMap.put(problemFeatureKeys[0], new UniformDistribution(1450, 5000));
        argMap.put(problemFeatureKeys[1], new UniformDistribution(3, 26));
        argMap.put(problemFeatureKeys[2], new UniformDistribution(0, 4));
        argMap.put(problemFeatureKeys[3], new UniformDistribution(0.1, 0.40));
        argMap.put(problemFeatureKeys[4], new UniformDistribution(13, 37));

        // Build a problem generator based on these arguments
        ProblemGenerator problemGenerator = new ProblemGenerator(argMap, domain);

        // Define any additional noise to be placed in the feature generation.
        // (None in this example.)
        NormalDistribution dist = new NormalDistribution(0, 0);
        DistributionCollection featureNoiseDistribution =
            new DistributionMap(NUM_PROBLEM_FEATURES);

        for (int i = 0; i < NUM_PROBLEM_FEATURES; i++)
```

```
            featureNoiseDistribution.putDistribution(problemFeatureKeys[i], dist);

        Distribution solutionNoiseDistribution = dist;
        double problemCount = 0;
        double noiseProbability = 0;

        // Build the CaseGenerator based on these specifications
        return new CaseGenerator(problemGenerator,
                                 featureNoiseDistribution,
                                 solutionNoiseDistribution,
                                 problemCount,
                                 noiseProbability,
                                 domain);
    }

    /** Construct another ProblemGenerator, distinct from that used to populate the case base. */
    private static ProblemGenerator constructTestProblemGenerator(Domain domain) {
        HashMap argMap = new HashMap(NUM_PROBLEM_FEATURES);
        argMap.put(problemFeatureKeys[0], new NormalDistribution(3500, 1000));
        argMap.put(problemFeatureKeys[1], new NormalDistribution(19, 6));
        argMap.put(problemFeatureKeys[2], new UniformDistribution(0, 4));
        argMap.put(problemFeatureKeys[3], new NormalDistribution(0.25, 0.07));
        argMap.put(problemFeatureKeys[4], new NormalDistribution(29, 8));

        return new ProblemGenerator(argMap, domain);
    }
}
```

```
/*******************************************************************************
 *                      Initial Version By: Steven Bogaerts                    *
 *                      Copyright (c) 2001 IUCBRF Group                        *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                   *
 * sbogaert@cs.indiana.edu                                                      *
 *******************************************************************************/

/*
 * RealtorSystem.java
 *
 * Created on March 30, 2001, 2:59 PM
 */
package edu.indiana.iucbrf.examples.realtor;

import edu.indiana.iucbrf.solution.*;
import edu.indiana.iucbrf.problem.*;
import edu.indiana.iucbrf.casebase.*;
import edu.indiana.iucbrf.casepackage.*;
import edu.indiana.iucbrf.cbrsystem.*;
import edu.indiana.iucbrf.retrieval.*;
import edu.indiana.iucbrf.adaptation.*;
import edu.indiana.iucbrf.maintenance.*;
import edu.indiana.iucbrf.performancemonitor.PerformanceMonitor;
import edu.indiana.iucbrf.domain.Domain;
import java.util.Vector;

/**
 * This class describes what components will be used for this house appraisal domain.
 * @author  Steven Bogaerts
 * @version 1.0
 */
public class RealtorSystem
    extends CBRSystem {

    /** This constructor exists for serialization purposes only. */
    protected RealtorSystem() {
    }

    /** Construct a RealtorSystem for the given domain, with a case base
     *  containing numCases case generated by caseGenerator.
     */
    public RealtorSystem(CaseGenerator caseGenerator,
                         Domain domain,
                         int numCases) {
        super();
        setDomain(domain);
        nonDataInit();

        // Use a simple, flat case base organization
        setCB(new FlatCaseBase(caseGenerator, numCases));

        // Set up the maintenance policy of the system.
        RealtorMaintenance maintenance = new RealtorMaintenance(this.getCB());
        setMaintenance(maintenance);

        // Set up a listener for case base events.
        RealtorCaseBaseListener listener = new RealtorCaseBaseListener(maintenance);
        CB.addCaseBaseListener(listener);
    }

    /** Initialize the components of the system that are never loaded from a file.
     */
    private void nonDataInit() {
        // Use 5-nearest-neighbor retrieval
        kNN retrieval = new kNN(getDomain().getProblemDifferentiator(), 5);
        // Use 1-nearest-neighbor retrieval
        // kNN retrieval = new kNN(getDomain().getProblemDifferentiator(), 1);

        // Do not provide debugging messages during retrieval
        retrieval.setDebugMode(false);

        // Do not track the differences between cases retrieved and the current problem
        // - only track the cases themselves
        retrieval.setTrackDifferences(false);
        setRetrieval(retrieval);
        setAdaptation(new RealtorAdapter(getDomain(), RealtorTestClass.sizeFeatureKey));
        setPerformanceMonitor(new PerformanceMonitor(false)); // send debugMode boolean value
    }
}
```

```
/*****************************************************************************
 *                       Initial Version By: Steven Bogaerts                 *
 *                       Copyright (c) 2001 IUCBRF Group                     *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                 *
 * sbogaert@cs.indiana.edu                                                   *
 *****************************************************************************/

/*
 * RealtorAdapter.java
 *
 * Created on March 30, 2001, 3:03 PM
 */
package edu.indiana.iucbrf.examples.realtor;

import edu.indiana.iucbrf.adaptation.*;
import edu.indiana.iucbrf.solution.*;
import edu.indiana.iucbrf.problem.*;
import edu.indiana.iucbrf.casepackage.*;
import edu.indiana.iucbrf.domain.Domain;
import edu.indiana.iucbrf.feature.DoubleFeature;
import edu.indiana.iucbrf.feature.DoubleRepresentable;
import edu.indiana.iucbrf.feature.FeatureKey;
import java.util.ArrayList;

/**
 * This is the domain-dependent implementation of the adaptation policy for this domain.
 * @author  Steven Bogaerts
 * @version 1.0
 */
public class RealtorAdapter
    extends Adaptation {

    /** The feature key for the house size problem feature, used in this simple
     * adaptation approach. */
    public FeatureKey houseSizeKey;

    /** Creates new RealtorAdapter */
    public RealtorAdapter(Domain domain,
                          FeatureKey sizeFeatureKey) {
        houseSizeKey = sizeFeatureKey;
    }

    /** Domain-specific adaptation of house prices.  This is a simple approach that
     *  only considers house size.  It is assumed that the domain sent here is the
     *  same as the one sent in the constructor.
     *  @param retrieved Contains only a single solution, because Realtor uses 1NN
     */
    public Solution adaptTo(final ArrayList retrieved,
                            final Problem currentProblem,
                            final Domain domain) {

        // Get the first (and only) retrieved case,
        // and its corresponding problem and solution.
        Case rc = (Case)retrieved.get(0);
        Problem rp = rc.getProblem();
        Solution rs = rc.getSolution();

        // Adjust price according to relative house size:
        // currentProblemPrice = (retrievedSize * retrievedPrice) / currentProblemSize
        return domain.getComponentFactory().constructSolution(
            (((DoubleRepresentable)currentProblem.getFeature(houseSizeKey)).getValueAsDouble() *
             rs.getValueAsDouble()) /
            ((DoubleRepresentable)rp.getFeature(houseSizeKey)).getValueAsDouble());
    }
}
```

```
/*****************************************************************************
 *                      Initial Version By: Steven Bogaerts                    *
 *                      Copyright (c) 2001 IUCBRF Group                        *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                   *
 * sbogaert@cs.indiana.edu                                                     *
 *****************************************************************************/

/*
 * RealtorMaintenance.java
 *
 * Created on August 5, 2002, 9:06 AM
 */
package edu.indiana.iucbrf.examples.realtor;

import edu.indiana.iucbrf.maintenance.NullMaintenance;
import edu.indiana.iucbrf.casepackage.Case;
import edu.indiana.iucbrf.casebase.CaseBase;
import java.util.Iterator;

/**
 * This maintenance policy does none of the standard maintenance approaches (hence extending
 * NullMaintenance) except that it provides a method to remove the "worst" case.
 * @author  Steven Bogaerts
 * @version 1.0
 */
public class RealtorMaintenance
    extends NullMaintenance {
    private CaseBase CB;

    /** This constructor exists for serialization purposes only. */
    protected RealtorMaintenance() {
    }

    /** Creates new RealtorMaintenance */
    public RealtorMaintenance(CaseBase CB) {
        this.CB = CB;
    }

    /** Remove the "worst" case: the lowest percentage of successful uses - where
     * the case contributed to a solution that was deemed successful.
     * Expensive (going through entire case base), but is just used to demonstrate a reaction
     * to CaseBaseListener events.  This method is called in reaction to case base
     * event received by RealtorCaseBaseListener.
     */
    public Case removeWorstCase() {
        Case currentCase;
        Case worstCase;
        double currentRating;
        double worstRating;
        double ratingSum = 0;
        Iterator CBiter = CB.iterator();

        // Guess that the first case is the worst
        worstCase = (Case)CBiter.next();
        worstRating = (double)worstCase.getSuccessfulUseCount() / worstCase.getUseCount();

        // Go through the remaining cases, searching for the worst
        while (CBiter.hasNext()) {
            currentCase = (Case)CBiter.next();
            currentRating =
                (double)currentCase.getSuccessfulUseCount() / currentCase.getUseCount();
            ratingSum += currentRating;

            if (currentRating < worstRating) {
                worstRating = currentRating;
                worstCase = currentCase;
            }
        }

        // Remove the worst case
        CB.removeCase(worstCase);

        //        System.out.println("Removed case with rating: " + worstRating);
        return worstCase;
    }
}
```

```java
/*****************************************************************************
 *                        Initial Version By: Steven Bogaerts                  *
 *                        Copyright (c) 2001 IUCBRF Group                      *
 * The licence for this code is included in IUCBRF_LICENSE.txt.  If you did not *
 * get this file with this distribution, request a copy from                   *
 * sbogaert@cs.indiana.edu                                                     *
 *****************************************************************************/

/*
 * RealtorCaseBaseListener.java
 *
 * Created on August 5, 2002, 8:59 AM
 */
package edu.indiana.iucbrf.examples.realtor;

import edu.indiana.iucbrf.casebase.event.CaseBaseListener;
import edu.indiana.iucbrf.casebase.event.CaseBaseEvent;
import edu.indiana.iucbrf.casepackage.Case;

/**
 * Listen for case base events and react accordingly.
 * @author  Steven Bogaerts
 * @version 1.0
 */
public class RealtorCaseBaseListener
    implements CaseBaseListener {

    // Needs the maintenance object for this system, since it uses it to call removeWorstCase().
    private RealtorMaintenance maintenance;

    protected RealtorCaseBaseListener() {
    }

    /** Creates new RealtorCaseBaseListener */
    public RealtorCaseBaseListener(RealtorMaintenance maintenance) {
        this.maintenance = maintenance;
    }

    /** A Case was added to the case base. */
    public void caseAdded(CaseBaseEvent cle) {
        System.out.println("Case added: " + cle.toString()); // + ", case added: " + caseAdded);
        System.out.println("\nTriggering maintenance to remove the least useful case...");

        Case caseRemoved = maintenance.removeWorstCase();
        System.out.println("Case removed: " + caseRemoved);
    }

    /** A Case was removed from the case base. */
    public void caseRemoved(CaseBaseEvent cle) {
        System.out.println("Case removed:" + cle.toString());
    }

    /** A Case was replaced in the case base. */
    public void caseReplaced(CaseBaseEvent cle) {
        System.out.println("Case replaced:" + cle.toString());
    }

    /** All cases were removed from the case base. */
    public void casesCleared(CaseBaseEvent cle) {
        System.out.println("Case cleared:" + cle.toString());
    }

    /** The standard class used to represent cases in the case base was changed. */
    public void caseClassChanged(CaseBaseEvent cle) {
        System.out.println("Case class changed:" + cle.toString());
    }
}
```

References

Aamodt, A., and Plaza, E. 1994. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications* 7(1):39–52. http://www.iiia.csic.es/People/enric/AICom.pdf.

Aha, D.; Breslow, L.; and Munoz-Avila, H. 2001. Conversational case-based reasoning. *Applied Intelligence* 14:9–32.

Aha, D. W. 1992. Generalizing from case studies: A case study. In *Proceedings of the Ninth International Conference on Machine Learning*.

Aktas, M.; Pierce, M.; Fox, G.; and Leake, D. 2004. A web based conversational case-based recommender system for ontology aided metadata discovery. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID 2004)*. IEEE Computer Society Press.

Bello-Tomás, J. J.; González-Calero, P. A.; and Dı́az-Agudo, B. 2004. Jcolibri: An object-oriented framework for building cbr systems. In Funk, P., and González-Calero, P. A., eds., *Proceedings of the Seventh European Conference On Case-Based Reasoning*, 32–46. Berlin: Springer.

Bogaerts, S., and Leake, D. 2005. Increasing ai project effectiveness with reusable code frameworks: A case study using IUCBRF. In *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference*. AAAI Press.

Breslow, L., and Aha, D. 1998. NaCoDAE: Navy conversational decision aids environment. Technical Report AIC-97-018, NCARAI, Naval Research Laboratory, Washington, DC.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2001. *Introduction to Algorithms*. Cambridge, MA: MIT Press.

Fayyad, U. M., and Irani, K. B. 1993. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the 13th International Joint Conference in Artificial Intelligence*, 1022–1027. Morgan Kaufmann.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley.

Jaczynski, M., and Trousse, B. 1998. An object-oriented framework for the design and the implementation of case-based reasoners. In *Proceedings of the Sixth German Workshop on Case-Based Reasoning, Berlin, Germany.*

Kolodner, J. 1993. *Case-Based Reasoning*. San Mateo, CA: Morgan Kaufmann.

Leake, D., and Sooriamurthi, R. 2004. Case dispatching versus case-base merging: When MCBR matters. *International Journal of Artificial Intelligence Tools* 13(1):237–254.

Leake, D., and Wilson, D. 1998. Categorizing case-base maintenance: Dimensions and directions. In Cunningham, P.; Smyth, B.; and Keane, M., eds., *Proceedings of the Fourth European Workshop on Case-Based Reasoning*, 196–207. Berlin: Springer Verlag.

Leake, D.; Bogaerts, S.; Evans, M.; McMullen, R.; Oder, M.; and Valerio, A. 2005. Using cases to support divergent roles in distributed collaboration. In *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference*. AAAI Press.

Leake, D., ed. 1996a. *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. Menlo Park, CA: AAAI Press/MIT Press.

Leake, D. 1996b. CBR in context: The present and future. In Leake, D., ed., *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. Menlo Park, CA: AAAI Press. 3–30. http://www.cs.indiana.edu/ leake/papers/a-96-01.html.

Leake, D. 2002. CBR/CIP microprograms archive. Accessed at www.cs.indiana.edu/˜leake/cbr/code/ August 25, 2004.

Manago, M.; Bergmann, R.; Wess, S.; and Traph oner, R. 1994. CASUEL: A common case representation language - version 2.0 esprit-project inreca, deliverable d1. Technical report, University of Kaiserslautern. ftp://ftpagr.informatik.uni-kl.de/pub/CASUEL/README.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Riesbeck, C., and Schank, R. 1989. *Inside Case-Based Reasoning*. Hillsdale, NJ: Lawrence Erlbaum.

Schank, R.; Riesbeck, C.; and Kass, A., eds. 1994. *Inside Case-Based Explanation*. Hillsdale, NJ: Lawrence Erlbaum.

Trousse, B. 2001. Introduction to CBR Tools. Accessed at http://www-sop.inria.fr/axis/cbrtools/usermanual-eng/Introduction.html August 25, 2004.

Watson, I. 1997. *Applying Case-Based Reasoning: Techniques for Enterprise Systems*. San Mateo, CA: Morgan Kaufmann.