cbr:works 4

Compendium

tec:inno GmbH

Sauerwiesen 2
D-67661 Kaiserslautern
Tel: +49 (0) 6301 606 400
Fax: +49 (0) 6301 606 409

cbrworks@tecinno.com
support@tecinno.com
http://www.tecinno.com

Content

Preface

TECINNO's CBR-Works family of products allows for the easy assembling of case-based applications for customer and sales support. CBR-Works presents a line of software tools to be used for the development and maintenance of such applications.

This compendium provides an overview of CBR-Works 4 and the underpinning mechanisms and methodologies it is using.

Beginning with a concise introduction to the abilities of CBR-Works, the following chapters will lead you to the background of Case-based Reasoning, the application of rules and similarities, and strategies for model maintenance for keeping the case base up-to-date.

1 The CBR-Shell

Nowadays, a proper tool for Case-Based Reasoning has to fulfill a wide range of tasks beyond simple retrieval. This chapter gives a brief overview of the abilities and features of the tool CBR-Works which provides support for the design process of a Case-Based application as well as for maintenance and retrieval. CBR-Works also provides the ability to reuse existing data from common database systems and may act as server for distributed access to a case base, including retrieval and case base management.

1.1 Introduction

Case-Based Reasoning (CBR) becomes more and more popular for companies, improving and enhancing their customer and sales support by introducing “intelligent applications”. Using a Case-Based application not only provides stored product catalogs or experience knowledge (the *cases*) to customers of a company. But also, by capturing problems and solutions a corporate memory is built, so the knowledge is no longer distributed in the workers minds but accessible to everyone in a company.

Besides collecting cases, applying Case-Based Reasoning necessitates a CBR-Tool supporting retrieval of matching cases as well as modeling and maintaining of the case base. Companies store information about their products in common database systems. Hence, as the amount of stored data is rather large, the CBR-Tool's ability of easy (re)using those information is important.

Another fundamental characteristic of a CBR-Tool is to cover the complete cycle of Case-Based Reasoning, i.e., retrieving cases similar to a user's specification, reusing a retrieved case as proposed solution, testing a solved case for success during the revisioning process, and retaining a new solution given in form of the revised case by including the experiences (the case) into the existing case base.

CBR-Works is a shell for Case-Based application building. Besides the retrieval of cases, it supports modeling the cases' structure and maintaining the case base. Its consultation mechanism also covers the whole CBR-Cycle from retrieving to revising. Though CBR-Works is designed as a complete environment, it may also act as a CBR-Server for several clients by the use of CQL (Case Query Language). Last but not least, CBR-Works offers an open interface to build a Case-Based application from existing data stored in common database systems.

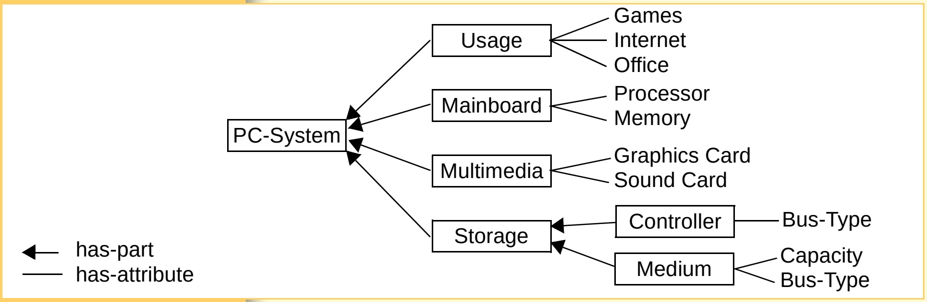

Figure 1-1: Structure of a simplified PC-Domain's case

The following sections give a brief overview of the abilities and features of CBR-Works. It will introduce the tool's elements that are used for building an application. To illustrate the building process, a simplified PC-Domain is used as depicted in figure 1-1. This example will be used throughout this chapter.

The following two sections describe the common elements used for building a case base in CBR-Works. Section 1.3 gives a concise description on maintenance in CBR-Works. In section 1.4 the interface for reusing data is tersely discussed. This is followed by an overview on how to consult a case base in section 1.5.

1.2 Structure Modeling

CBR-Works is suited for intelligent solutions in a variety of domains and environments. Its graphical editors support the user to design complex knowledge models. An object-oriented approach is used in CBR-Works to design the underlying structure of cases. This structure can be edited and maintained in an easy and intuitive way.

1.2.1 Concepts

In CBR-Works, *concepts* define the structure of the cases. They are defined in hierarchy similar to a class-model hierarchy including inheritance. Each *concept* consists of attributes which can be either atomic (defined by a *type)* or complex (has-part relationship to another *concept*).

For retrieval purposes, attributes have three additional, functional properties: one for defining its weight, i.e., its importance in respect to the other attributes of the concept, a property for defining whether an attribute is discriminant for retrieval or will be ignored, and another property defining if an attribute is mandatory for a case to be valid. Moreover, for every attribute a question and an annotation may be given that can be used by clients when asking for the value and to refer to further information about an attribute.

In figure 1-1 each rectangle may be seen as a concept. For example, **Storage** consists of the two complex attributes **Controller** and **Medium**, and again the latter consists of the two atomic attributes **Capacity** and **Bus-Type**.

Concept Similarity

Beside attributes, the type of similarity can be specified for every concept. The concept's similarity consists of two parts: the similarity of a concept's contents (*contents-based similarity*) and the similarity between concepts (*structure-based similarity*).

The contents-based similarity of a concept is computed based on the attributes defined in the concept. It may be one of the following:

- *Average*: All attribute similarities contribute to the contents-based similarity by computing their average.
- *Euclidean*: Geometric interpretation of the contents-based similarity (distance between two concepts, based on its contents).
- *Minimum*: The lowest attribute similarity defines the contents-based similarity.
- *Maximum*: The highest attribute similarity defines the contents-based similarity.

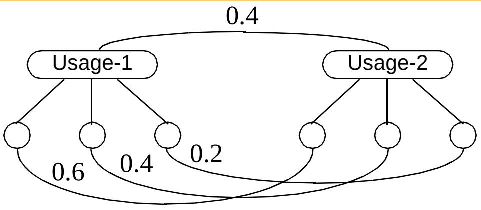

Figure 1-2: Example of contents-based similarity using Average

An example for a contents-based similarity is given in figure 1-2. Here, the similarity between the usage parts of two PC-Domain cases is computed using Average. The numbers are the computed similarities between two objects which are connected by a corresponding arc. The upper similarity computes as average of the lower ones.

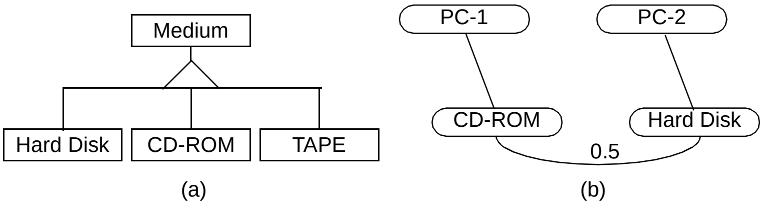

Figure 1-3: Example for structure-based similarity: a) concept-hierarchy for Medium b) structure-based similarity between two PC-Domain cases where Medium is the common father

The structure-based similarity defines similarities between concepts independent of their contents. Inside a concept-hierarchy, the similarity of concepts to each other may be explicitly or implicitly defined by using a taxonomic view of the hierarchy.

In the PC-Domain a concept-hierarchy could be defined like in figure 1-3a. Assuming the initial taxonomic view of the hierarchy as base for the structure-based similarity, it computes to $\frac{\text{level of common father}}{\text{number of levels}}$. An example for a two-level taxonomy is shown in figure 1-3b.

The concept’s similarity computes as a weighted sum of structure-based and contents-based similarities.

Rules

Additionally, rules may be specified for each concept, either being *completion* or *adaptation* rules. Completion rules apply to cases of a case base as well as to a query whenever a new value is given for an attribute. If some attribute values depend on each other, completion rules ease handling by automatically setting appropriate values. Adaptation rules get activated only after retrieval and they are used to combine attribute values of the query and retrieved cases and to apply the result to a target case. That way, slightly modified cases are created which may fit the customers need better than the retrieved case.

Each rule, for adaptation as well as completion, consists of two parts: a *condition* part and a *conclusion* part. The condition part defines a conjunction of conditions. A condition may either be a predicate or a simple calculation over attributes (of the according concept), constants (defined using concepts or types), or local variables (computed by previous conditions). The conclusion part consists of actions being executed if all conditions of the condition part are fulfilled. An action may be an assignment of values to attributes (atomic as well as complex), a command to open a notifier (e.g., to report inconsistencies due to a given value), or changes to retrieval-influencing values (e.g., filters and weights).

For example, to keep consistency for the **Storage** component of a **PC-System**, a completion rule may be defined to ensure that a **Medium** will fit to a specified **Controller.** If a **Medium** gets defined having a **Bus-Type** different to an already specified **Controller**, a notifier will open to inform the customer about this inconsistency. More complex, an adaptation rule may be defined choosing a, e.g, different, fitting **Controller** replacing the previously specified one.

1.2.2 Types

Similar to concepts, types are defined hierarchically. New types are defined by building subtypes of the existing elementary types shown in table 1-1. They differ in their usability: a type may be used *immediate* or *derived*. While *immediate* types cover the whole range of possible values of a type, *derived* types get restricted in their range by defining an enumeration of elements of its elementary type or, in case of numeric types, by specifying an interval.

Table 1-1: Elementary Types in CBR-Works

Type	Usability
Integer	immediate and derived
Real	immediate and derived
Date	immediate and derived
Time	immediate and derived
Boolean	immediate only
String	immediate and derived
Symbol	immediate and derived
Ordered Symbol	derived only
Taxonomy	derived only
Reference	derived only

Additional to the type *Symbol*, *Ordered Symbol* provides a total and *Taxonomy* a partial order over a given enumeration of values. For example, **Hard**

Disk being defined using *Taxonomy* introduces a partial order of the values compatibility regarding Bus-Types as shown in figure 1-4.

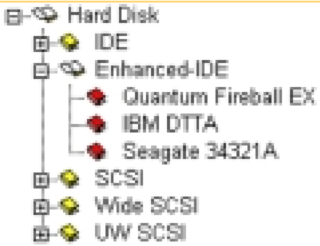

Figure 1-4: Taxonomy over selected Processors

Furthermore, *constructional types* are available for defining intervals and sets using defined, elementary types. Here, intervals are restricted to ordered types where sets may be defined over any elementary type or one of its derivatives (see table 1-2 for restrictions).

Table 1-2: Constructional Types in CBR-Works

Type	Value-Type Restriction
Set	All but Boolean
Interval	Ordered Types (e.g., Ordered Symbol, Integer, Real)

Type Similarity

For each type derived from elementary types, similarities may be defined describing major parts of the experts knowledge which is necessary for intelligent retrieval. The definition ranges from value-to-value specifications in form of a table over special, type-depending similarities (e.g., for string types) to functional specification by graphs. Furthermore, an interface is given to define a programmatic similarity for any derived type. An example of functional similarity is given in figure 1-5 regarding a customers “feeling of an acceptable price” being different in a retrieved case to a specified value in the query.

A higher price only is accepted up to a specific limit quickly dropping the higher it is. The situation is similar offering products with lower prices, as a customer usually thinks of lower quality by a lower price once the negative limit is passed.

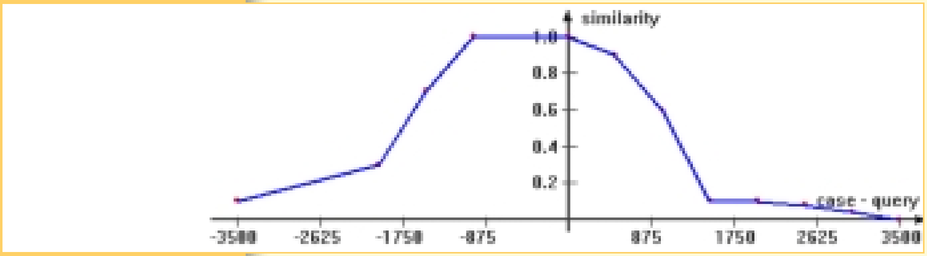

Figure 1-5: Example of a similarity-function for the price of a computer

For derivatives of constructional types, predefined similarity functions are given based on intersection and inclusion of sets or intervals.

In CBR-Works, it is possible to define more than one similarity for each type as the decision which similarity to use may depend on values selected for retrieval. This decision may be formulated using completion rules for concepts.

1.3 Case Base Building and Maintenance

The heart of a CBR-System is the case base containing the active knowledge of the domain to be represented. Each case's structure is defined by the underlying concept and its data represents exactly one information entity.

Cases in the Case Base.

In CBR-Works, the case base consists of a number of virtual case bases each of which is founded on one of the concepts being marked as case-concept, i.e., concepts which are specified for being the structure of cases. These virtual case bases may not be seen stand-alone, but the complete set of virtual case bases is united into the CBR-Works' case base.

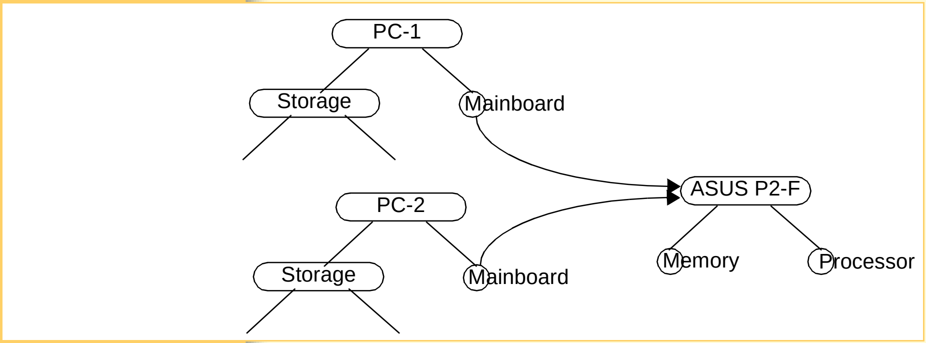

Figure 1-6: Example for cases of multiple, virtual case bases. The Storage of each PC-System is defined as complex attribute belonging to the PC-System case while the Mainboard is defined as reference pointing to the according Mainboard case

In the PC-Domain, several virtual case bases may be useful, e.g., not only storing complete **PC-Systems** as cases but also monitor exchangeable components like **Hard Disk** and **Mainboard** cases. Hence, **PC-System** cases having the same **Mainboard** refer to the same case instead of having the same data twice in the case base (see figure 1-6). As a side-effect, the effort on keeping the consistency of the case base according to changes in the specification of referred information is reduced.

A case in CBR-Works has four possible states: *unconfirmed*, *confirmed*, *protected*, and *obsolete*. Usually, new cases become *unconfirmed* being unrevised or incomplete cases not valid for retrieval. Revised cases become either *confirmed* which allows for retrieval or *protected* which additionally protects the case from changes. Old cases, no longer valid for retrieval but probably useful for further statistics, become *obsolete*.

Case Base Maintenance.

Important for a consistent case base is the maintenance of its cases concerning validity of values and changes to the underlying model of the domain.

Therefore, CBR-Works provides several mechanisms ensuring that each case which is confirmed or protected to be valid regarding modeled type-ranges after inserting or modifying a case in the case base as well as changing the structure of the model, e.g., changing the range of a type. In the latter and similar operations, appropriate actions to the case base are selectable, being necessary to keep consistency and prevent data loss due to changes in the model, e.g., remapping values of cases when changing the type of attributes.

1.4 Reusing Data

Building a CBR-System from scratch is necessary and appropriate for domains that are not available in electronic form. For information being stored in, e.g., a database, a CBR-Tool must be able to reuse such data rather than having the user to remodel the domain and manually add all information to the case base.

CBR-Works supports connections to electronic information via the open database connection (ODBC) system. Hence, any source (e.g., sheet or database) which contains the domain-data can be connected to CBR-Works for import of structure and data to build up the CBR-System. Here, concepts are build from tables or views being defined by the source, and types may be generated from the contents of each column. Relations between tables are modeled by either using references or aggregation. In case of aggregation, the information given in related tables becomes part of a case. Using references necessitates the referenced concept also being marked as concept for building cases.

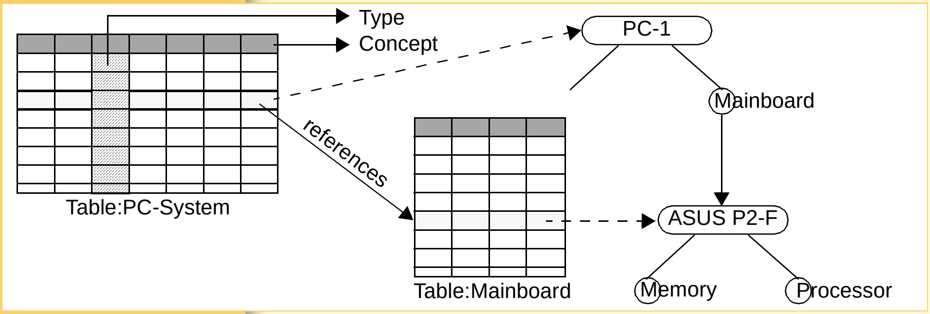

Figure 1-7: Creation of a PC-System case out of a database

After building the domain model that bases on the source information, the cases are imported into the case base using the same interface served by ODBC. Each row of a table becomes one case, including aggregated concepts built of rows from related tables and references to cases built from related table-rows (see figure 1-7).

1.5 Consulting the Case Base

For querying the case base and retrieving cases from it, CBR-Works offers several interfaces for console using as well as for clients using CBR-Works as server. The so called consultation of the case base covers the whole Case-Based Reasoning Cycle. Not only providing retrieval-mechanisms but also the possibility to revise and to retain suggested or confirmed solutions in form of cases. In CBR-Works, the revision step also includes adaptation of cases using the appropriate rules.

1.5.1 Common Consultation

Generally, consultation happens either by using firsthand access to CBR-Works as a CBR-Console or by remotely accessing the case base with CBR-Works acting as a Server.

In addition to requested values, a query consists of further information like filters and weights for attribute-values, being hard constraints in opposition to the rather soft constraints provided by similarity-measures. Other additional information is: a threshold to lay down the minimal similarity a case may have to be valid as solution, options for completion of the query’s values and adaptation of retrieved cases, and options for defining the virtual case bases to be considered.

1.5.2 Strategic Questioning

Besides the common consultation, strategic questioning of attribute-values interactively leads to suggested solutions. Here, algorithmic mechanisms ask for values in order to quickly reduce the number of possible solutions.

The predefined strategy of *information gain* operates on retrieved cases and computes the gain of information for every undefined attribute of the query according to its ability to partition the space of solutions.

A second strategy bases on modeled *importance ranking*, where the modeler determines the order of selected questions. Questions not explicitly ordered by this ranking are handled using the strategy of information gain which is normalized to the range between zero and the lowest ranking given.

1.6 Summary

CBR-Works can be seen as a CBR-Shell providing all necessary tools to model, maintain and consult a case base. Moreover, CBR-Works is able to reuse information already stored in electronic form. For simple representation of the added value and power brought in by CBR, an integrated WWW-Server with adapted generic interface supports online retrieval without additional programming.

The following chapters will lead into more details and partially academic views on mechanisms and methodology of CBR and CBR-Works.

2 Similarity of Taxonomies

2.1 Introduction

In current academic and commercial CBR systems, cases are often represented in an object-oriented fashion. Cases are collections of objects, each of which is described by a set of attribute-value pairs. The structure of an object is described by an object class that defines the set of attributes together with a type (set of possible values) for each attribute. Usually, the similarity between a query and a case from the case base is computed in a bottom up fashion: for each attribute, a local similarity measure determines the similarity between two attribute values and for each object (and the case) a global similarity measure determines the similarity between two objects (or between the case and the query) based on the local similarities of the belonging attributes.

For defining attribute types (sets of possible attribute values), taxonomies are widely used. A taxonomy is an n-ary tree in which the nodes represent symbolic values. The symbols at any node of the tree can be used as attribute values in a case or a query. Unlike a plain symbol type, which only lists possible attribute values, a taxonomy represents an additional relationship between the symbols through their position within the taxonomy-tree. This relationship expresses knowledge about the similarity of the symbols in the taxonomy.

Although taxonomies are widely used, there is currently no clear picture of what knowledge about local similarities is captured in a taxonomy. The impression that similarity measures are usually constructed in an ad hoc manner also holds for local similarity measures for taxonomy type attributes. This chapter analyzes several situations in which taxonomies are used in different ways and proposes a systematic way of specifying local similarity measures for taxonomy types. The proposed similarity measures have a clear semantics and are easy to compute at run-time.

2.2 Different Use of Taxonomies

We now describe four examples in which the taxonomy shown in figure 2-1 is used.

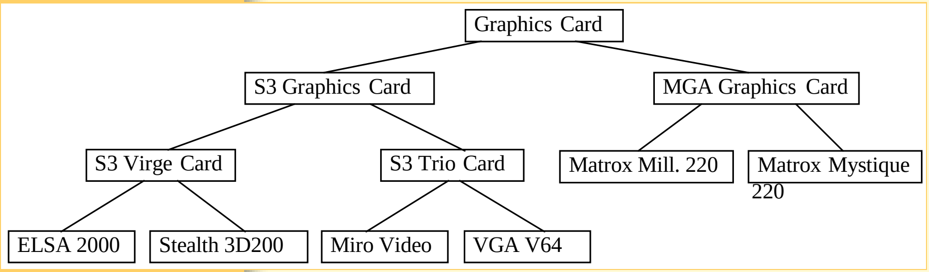

Figure 2-1: Taxonomy of Graphics Cards.

Example 1a

Consider a CBR system for the sales support of Personal Computers. A Case represents an available PC from the stock. The case representation contains an attribute “graphics card”, and the taxonomy from Figure 1 represents the set of possible values. Consider a case c_1 with the *ELSA 2000* card and a case c_2 with *Matrox Mystique* card. If we assume that a customer enters a query to our hypothetical CBR system in which she/he specifies that she/he wants a *Miro Video* graphics card, then c_1 is certainly closer than c_2, because *Miro Video* and *Elsa 2000* have more in common (e.g. the S3 chip) than the *Miro Video* and the *Matrox Mystique*. In general, we could use a similarity measure that assesses similarity based on the distance between case and the query value in the taxonomy tree.

Example 1b

Imagine, the customer states in the query a request for a *S3 Graphics Card*. Then, any of the graphics cards in the S3 sub-tree are perfectly suited. Hence, we expect the local similarity value between this query and case c_1 (from example 1a) to be 1. From this consideration we can conclude that whenever the query value is located above the case value, the similarity measure should be 1.

Example 2a

Consider a trouble-shooting CBR system for PCs in which cases encode diagnostic situations and faults that have occurred previously. The domain expert describes a fault that can occur with any S3 card. Therefore, the respective case contains the attribute value *S3 Graphics Card*. Assume now, a PC user has a problem and she/he states that there is an *Elsa 2000* card in the PC, than the local similarity for the graphics card attribute should be equal to 1 because the case matches exactly w.r.t. this attribute. From this consideration we can conclude that whenever the case value is located above the query value the similarity measure should be 1.

Example 2b

For the same trouble-shooting example, imagine now a different query in which the user does not exactly know what kind of graphics card is installed in the PC, but she/he knows that it is a *S3 Trio* card. She/He therefore enters *S3 Trio* as attribute value in the query. Again, the case about *S3 cards* mentioned in Example 2a matches exactly because, whatever graphics card the user has, we known it is an *S3 card* and the situation described in the case applies. However, if we consider a different case that describes a problem with the *Miro Video* card, then this case does not match exactly. Since we don't know what graphics card the user has (it can be a *Miro Video* but it can also be a *VGA V64*) we expect a local similarity value less than 1. Therefore we cannot conclude that whenever the query value is located above the case value the similarity measure should be 1.

Although we have used the same taxonomy in all four examples, it is obvious that they have to be treated differently for the similarity computation. In the query and cases from example 1a, only leaf nodes from the taxonomy are used. The examples 1b to 2b make use of inner nodes of the taxonomy, but in each example the semantics of the inner nodes is different which lead to different similarity measures.

2.3 Knowledge Contained in Taxonomies

We now analyze the knowledge that is contained in taxonomies. We will show that a taxonomy contains two different kinds of knowledge:

1. Knowledge about classes of objects[1] (represented by inner nodes).
2. Knowledge about the similarity between leaf nodes.

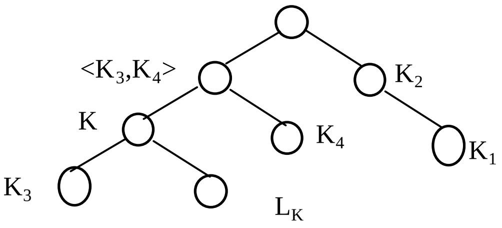

Figure 2-2: Illustration of basic notions

2.3.1 Basic notions

We briefly introduce a few notions (see figure 2-2) that will be further used in this chapter. Let *K* be an inner node of the taxonomy, then L_K denotes the set of all leaf notes from the sub-tree starting at *K*. Further, $K_1 < K_2$ denotes that K_1 is a successor node of K_2, i.e., K_2 is on a path from K_1 to the root node. Moreover, $<K_3,K_4>$ stands for the node that is the nearest common predecessor of K_3 and K_4, i.e., $<K_3,K_4> \geq K_3$ and $<K_3,K_4> \geq K_4$ and it does not exist a node $K' < <K_3,K_4>$ such that $K' \geq K_3$ and $K' \geq K_4$ holds.

2.3.2 Semantic of Taxonomy Nodes

In a taxonomy, we must distinguish between leaf nodes and inner nodes. Leaf nodes represent *con-*

1. Here, the word object is not meant in the sense of the object-oriented paradigm.

crete objects[1] of the real world, e.g., existing graphics cards. Inner nodes, however, describe *classes* of real world objects. An inner node K represents a class with certain properties that all of the concrete objects from the leaf nodes L_K have in common. Unlike classes that occur in the object-oriented paradigm, the classes that are represented by the inner nodes of a taxonomy are not described intentionally by a set of properties, but extensionally through the set of concrete objects L_K that belong to the class. Therefore, an inner node K stands for the set L_K of real world objects.

In the taxonomy shown in figure 2-1, the leaf nodes represent existing graphics cards and the inner nodes represent classes of graphics cards. For instance, *S3 Virge* stands for all graphics cards with the S3 Virge chip on them, i.e, for the set of cards {*Elsa 2000*, *Stealth 3D 2000*}.

When a CBR application developer builds a taxonomy, she/he should introduce useful sets of real-world objects, i.e., sets that are likely to occur in a case or a query. The taxonomy defines unique names (like *S3 Virge*) for these sets which are then used as abbreviations. Since the sets that are represented by these inner nodes are defined by the taxonomy itself, they are the same in all of the examples shown above, e.g., *S3 Virge* always stands for {*Elsa 2000*, *Stealth 3D 2000*}. However, the meaning of this set is quite different in the examples as we will discuss in detail in section 2.3.5.

2.3.3 Similarity Between Leaf Nodes

Besides the definition of classes of objects, a taxonomy also encodes some knowledge about the similarity of the real-world objects, i.e., knowledge about the similarity of the leaf nodes of the taxonomy. The inner nodes cluster real-world objects that have some properties in common. The deeper we decent in the taxonomy, the more features do the objects, that the inner node represents, have in common. For example, all real-world objects (leaf nodes) from the hierarchy in Fig. 1 have in common that they are all graphics cards. The objects below the *S3 graphics card* node have in common that the all use some kind of S3 chip, and the objects below the *S3 Trio* node have in common that they all use the specific S3 Trio chip. We can now define local similarity as a measure of how many features the compared objects have in common. The more features are shared, the higher is the similarity. For example, the similarity between *Elsa 2000* and *Stealth 3D200* is higher than the similarity between *Elsa 2000* and *VGA V64*.

This consideration leads to the following general constraint for defining the local similarity measure for the leaf nodes of a taxonomy:

(1) $$sim(K, K_1) \leq sim(K, K_2) \text{ IF } \langle K, K_1 \rangle > \langle K, K_2 \rangle$$

It states that the similarity between the leaf node K and K_1 is smaller than the similarity between the leaf node K and K_2 if the nearest common predecessor of K and K_1 is located below the nearest common predecessor of K and K_2. It does not state anything about the relationship between $sim(K_1,K_2)$ and $sim(K_3,K_4)$ unless $K_1=K_3$. Please note that this constraint defines an ordinal similarity measure, i.e., if the value K is given in the query, a partial order of all cases is induced.

2.3.4 Assigning Similarity Values for Leaf Nodes

The taxonomy only represents the constraint shown above, but does not define numeric values for the similarity between two leaf node objects. However, several models of similarity computation require a numeric value (e.g. from the interval [0..1]) to express the local similarity, because this value is further used in the computation of a global similarity. For this purpose, we have to add additional knowledge to the taxonomy. Basically, there can be different ways of doing this in a way, that the resulting cardinal similarity measure is compatible with the constraint. We now present a new approach which is quite simple and easy to use, but nevertheless very powerful.

Every inner node K_i of the taxonomy is annotated with a similarity value $S_i \in [0..1]$, such that the following condition holds: if $K_1 > K_2$ then $S_1 \leq S_2$. The deeper the nodes are located in the hierarchy, the larger the similarity value can become. The semantic of the similarity value is as follows:

The value S_i represents a lower bound for the similarity of two arbitrary objects from the set L_{Ki}, or written formally: $\forall x, y \in L_{K_i}\, sim(x, y) \geq S_i$

Any two objects from L_{Ki} are at least similar to each other with the value S_i, but there similarity can be higher. The similarity value that is assigned to a node should be justified by the features that all of the objects that belong to this inner node (class) have in common. The fact that the objects belong to this class and have common properties justifies that we can state a lower bound for there similarity. However, objects belonging to one class can of course also belong to a more narrow class further down in the taxonomy, which means that these objects share even more properties and therefore possibly have a higher similarity. We therefore define the similarity between to objects as follows:

$$(2) \qquad sim(K_1, K_2) = \begin{cases} 1 & \text{if } K_1 = K_2 \\ S_{\langle K_1, K_2 \rangle} & \text{otherwise} \end{cases}$$

$S_{<K1,K2>}$ is the similarity value assigned to the node $<K_1,K_2>$, i.e., the nearest common predecessor of K_1 and K_2.

It can be shown that this similarity definition preserves constraint (1).

Example

If we assign the similarity values from table 2-1 to the taxonomy from figure 2-1, the similarities that are shown in table 2-2 arise.

2.3.5 Semantic and Similarity of Inner Nodes

If we now recall again the examples that we have presented in section 2.2, it is obvious that the ”graphics card” attribute must be treated differently in the different examples, although they all use the same taxonomy. From that it becomes clear, that some additional knowledge which we have not yet discussed, plays a role during similarity assessment. However, this knowledge is not contained in the taxonomy itself.

The knowledge that we are looking for is the knowledge about the semantic of the inner nodes, i.e., the semantic of the set of concrete objects that they represent. In our example, the question is: what does it mean when the case or query contains the statement: ”graphics card: S3 Graphics Card”?

Table 2-1: Similarity between leaf nodes

	Elsa	Stealth	Miro	VGA	M. Mill.	M. Myst.
Elsa	1	0.7	0.5	0.5	0.2	0.2
Stealth	0.7	1	0.5	0.5	0.2	0.2
Miro	0.5	0.5	1	0.9	0.2	0.2
VGA	0.5	0.5	0.9	1	0.2	0.2
M.Mill.	0.2	0.2	0.2	0.2	1	0.8
M.Myst.	0.2	0.2	0.2	0.2	0.8	1

Table 2-2: Similarity Values

Value	Similarity
Graphics Card	0.2
S3 Graphics Card	0.5
S3 Virge	0.7
S3 Trio	0.9
MGA	0.8

In fact, there are different interpretations of this statement that are now discussed.

Any value in the query

The user specifies the value K in the query. This means that she/he is looking for a case that contains one of the values from the set L_K. In the example 1b, the user wants an S3 graphics card, but he does not care whether it is a *Elsa 2000*, *Stealth 3D 200*, *Miro Video*, or an *VGA V64*. It is clear that the local similarity between this query and any of these four leaf nodes is equal to 1. But what about the similarity to any other leaf node? To answer this question more systematically, we can define the required retrieval result indirectly as follows: Instead of submitting a single query to the system that contains an inner node *K*, the user could alternatively submit several queries to the system, one for each concrete value from L_K and merge the retrieval results, i.e., select the case with the highest similarity. The result of using the query with the inner node *K* should yield the same case with the same similarity as the merging of the multiple retrievals. To achieve this, the similarity measure for the inner node must select the maximum similarity that arises from each of the leaf nodes.

Any value in case

The case contains an inner node K, which describes a situation in which the case is valid for all attribute values of the set L_K. This leads to a kind of generalized case. The generalized case (in which the attribute value *K* is used) stands for the set of cases that results by replacing *K* by all of the members of the set L_K. In Example 2a, the case representing a fault for any *S3 graphics cards* stands for a set of four cases, each of which represents a fault for the *Elsa 2000*, *Stealth 3D 200*, *Miro Video*, and the *VGA V64*, respectively. Here, the inner node is used to keep the number of cases in the case base small. However, the retrieval result should of course not be affected. Therefore, the result of having a case in the case-base that contains an inner node K should be the same than having all cases in the case base, one for each concrete value from L_K. Since we are looking for the most similar case, we again have to assess the similarity for the inner node by selecting the maximum similarity that arises from each of the leaf nodes.

Uncertainty

This situation differs significantly from the previous two. Here, the use of an inner node K means that we don’t know the exact value for this attribute, but we know that it is one from the set L_K. In Example 2b, we know that the user has a *S3 Trio* card which means it can be one from the set {*Miro Video, VGA V64*}. This kind of uncertainty can occur in queries as well as in cases. The user can think of this uncertainty in different ways: treating it optimistically, pessimistically, or as an average case.

We can now define the local similarity SIM(Q,C) between a query value Q and the a case value C each of which can be either a leaf node, an inner node with the ”any value” interpretation or an inner node with the ”uncertainty” interpretation. This leads to 9 possible combinations shown in figure 2-3. Seven of the 9 combinations in the table are marked with a roman number that is further used to reference the formulas for computing the similarity. These are the ones that occur most likely. However, the following considerations can easily be extend also to the two missing combinations.

Table 2-3: Combination of different semantics for taxonomy values in query and case

Query / Case	Leaf Node	Any Value	Uncertainty
Leaf Node	I	II	V
Any Value	III	IV	
Uncertainty	VI		VII

In the following, *sim(q,c)* denotes the similarity between two leaf nodes, q from the query and c from the case. It can be computed as shown in section 2.3.4

I: Only the similarity between leaf nodes is computed and hence *SIM(Q,C) = sim(Q,C)* holds.

II: The query contains a leaf node and the case contains an inner node representing a set of values each of which is a correct value for the case. Therefore, the use of this set in the attribute is a shortcut for the use of several cases, one for each value in the set. Since we are looking for the most similar case in the cases base, the similarity between the query and our case containing the inner node is equal to the highest similarity between the query and one of the values from the set. Hence, the following holds:

$$SIM(q,C) = \max\{sim(q,c) | c \in L_C\} = \begin{cases} 1 & \text{if } \ q < C \\ S_{\langle q,C\rangle} & \text{otherwise} \end{cases}$$

This definition ensures, that the similarity is the same as the similarity that arises when each of the cases with leaf node values would have been stored in the case base. This measure is appropriate for example 2a.

III: Here, the specification of this inner node can be viewed as a shortcut for posing several queries to the system, one for each of the values from the set that the node represents. Since we are again interested in the most similar case, we can again select the most similar attribute value from the set. Hence the following holds:

$$SIM(Q,c) = \max\{sim(q,c) | q \in L_Q\} = \begin{cases} 1 & \text{if } \ c < Q \\ S_{\langle Q,c\rangle} & \text{otherwise} \end{cases}$$

This measure is appropriate for example 1b.

IV: This is a combination of II and III. We are looking for the highest possible similarity between two objects from the two sets since both, the query and the case, represent alternatives that are suited equally well. Hence, the following holds:

$$SIM(Q,C) = \max\{sim(q,c) | q \in L_Q, c \in L_C\} = \begin{cases} 1 & \text{if} \quad C < Q \text{ or } Q < C \\ S_{\langle Q,C \rangle} & \text{otherwise} \end{cases}$$

V: The case contains an inner node which represents a set of values from which only one value is actually correct for the case, but we don't know which one. Therefore, our similarity measure has to reflect this lack of information. There are three possible approaches: we can assess the similarity in a pessimistic or optimistic fashion, or we can follow an averaging approach:

Pessimistic approach: We assess the similarity between the known object (in the query) and the partially unknown object (in the case) by computing the lower bound for the similarity as follows:

$$SIM(q,C) = \min\{sim(q,c) | c \in L_C\} = S_{\langle q,C \rangle}$$

Optimistic approach: We assess the similarity between the known object (in the query) and the partially unknown object (in the case) by computing the upper bound for the similarity, which results in the same formula that was already shown in III.

Average approach: We assess the similarity between the known object (in the query) and the partially unknown object (in the case) by computing the expected value of the similarity as follows:

$$SIM(q,C) = \sum_{c \in L_C} P(c) \cdot sim(q,c)$$

where *P(c)* is the probability that the value of the attribute under consideration has the value c given the fact that we know that $c \in L_C$ and given the known information about the current case. Since P(c) is hard to determine, we can, for example, estimate P(c) by $1/|L_C|$, assuming that all attribute values are equally distributed and that all attributes are independent.

VI: The uncertain information is contained in the query; the information in the case is certain. This case is quite similar to the previous case V, i.e., we can again use a pessimistic, an optimistic, or an average approach. The only change in the formulas for similarity computation is the fact that the minimum, maximum, and sum operations are now performed using the elements from the query L_Q and not the elements form the case.

VII: The uncertain information is contained in the query and in the case. The similarity is computed as follows:

Pessimistic approach:

$$SIM(Q,C) = \min\{sim(q,c) | c \in L_c, q \in L_Q\} = S_{\langle Q,C \rangle}$$

Optimistic approach:

$$SIM(Q,C) = \max\{sim(q,c) | q \in L_Q, c \in L_c\} = \begin{cases} 1 & \text{if } \ C < Q \text{ or } Q < C \\ S_{\langle Q,C \rangle} & \text{otherwise} \end{cases}$$

Average approach:

$$SIM(Q,C) = \sum_{c \in L_C, q \in L_Q} P(c) \cdot P(q) \cdot sim(q,c)$$

We see that in all of these cases (except for the average approach to uncertainty), similarity between inner nodes can be computed very easily by determining the position of the query and the case value in the taxonomy and by looking up the similarity value at the appropriate taxonomy node. This enables the use of taxonomies in CBR.

2.4 Summary

We have shown that taxonomies represent two kinds of knowledge: first, knowledge about classes of objects and second, knowledge about the similarity between leaf nodes which represent real-world objects. We have presented a new approach for defining a numeric similarity-value between leaf nodes by assigning similarity values to the inner nodes of the taxonomy. Moreover, we have shown that additional knowledge is required to decide how the similarity between inner nodes of the taxonomy can be computed. This knowledge states how the classes (set of real-world objects) have to be interpreted: as any value from the set or as a kind of uncertainty. However, independent on the kind of interpretation, there is a quite simple way of computing the similarity between two inner nodes, if the proposed approach to determine the similarity between leaf nodes is used.

From these considerations we can see that a taxonomy can be used (and should be used because of the simple computation of similarities) if

- an attribute shall contain a set of values in the query and/or in the case *and*
- these sets represent either uncertainty *or* a list of equally well suited objects *and*
- we can define in advance a hierarchy of disjoint sets of similar objects that can occur in the query or the case.

These three rules of thumb can be used as guidelines within a similarity definition method of a case-based reasoning methodology.

In our discussion, we restricted ourselves to taxonomies of basic objects which don't have an internal structure. However, our considerations also apply to the generalization/specialization hierarchy of the object classes in an object-oriented data (or case) model. This inheritance hierarchy is of the same nature than the taxonomies we have just discussed. The only difference is that the objects, which are instances of classes, have an additional internal structure, i.e., each object is described by a set of attributes.

The global similarity measures (e.g. weighted sum of local similarities from the attributes) used up to now in most CBR systems only allow to compare two objects from the same object class. They do not state anything about how objects of different object classes can be compared.

From our considerations, we suggest to compute the global similarity between to objects (possibly from different classes) based on two components:

- an inter-object similarity stating the similarity of the objects based on those attributes of the nearest common superclass. This superclass contains those attributes that both objects have in common.
- an intra-object similarity stating the similarity between objects on the basis of the class to which they belong. For this purpose, we can directly apply our considerations about taxonomies. We can assign a similarity value to each

object class of the inheritance hierarchy and use them to compute the intra-object similarity.

We combine inter and intra-object similarity multiplicatively to a global object similarity. Thereby, the intra-object similarity states the maximal similarity that two objects of different classes can have.

3 Similarity Measures for Case Representations

Object-oriented case representations require approaches for similarity assessment that allow to compare two differently structured objects, in particular, objects belonging to different object classes. Currently, such similarity measures are developed more or less in an ad-hoc fashion. It is mostly unclear, how the structure of an object-oriented case model, e.g., the class hierarchy, influences similarity assessment. Intuitively, it is obvious that the class hierarchy contains knowledge about the similarity of the objects.

However, how this knowledge relates to the knowledge that could be represented in similarity measures is not obvious at all. This chapter analyzes several situations in which class hierarchies are used in different ways for case modeling and proposes a systematic way of specifying similarity measures for comparing arbitrary objects from the hierarchy. The proposed similarity measures have a clear semantics and are computationally inexpensive to compute at run-time.

3.1 Introduction

Several recent CBR systems apply object-oriented techniques for representing cases. Such representations are particularly suitable for complex domains in which cases with different structures occur. Cases are represented as collections of *objects*, each of which is described by a set of attribute-value pairs. The structure of an object is described by an *object class* that defines the set of attributes (also called slots) together with a *type* (set of possible values or sub-objects) for each attribute. Object classes are arranged in a *class hierarchy*, that is, usually a n-ary tree in which sub-classes inherit attributes as well as their definition from the parent class (predecessor).

Moreover, we distinguish between *simple attributes*, which have a simple type like Integer or Symbol, and so-called *relational attributes*. Relational attributes hold complete objects of some (arbitrary) class from the class hierarchy. They represent a directed binary relation, e.g., a part-of relation, between the object that defines the relational attribute and the object to which it refers. Relational attributes are used to represent complex case structures. The ability to relate an object to another object of an arbitrary class (or an arbitrary sub-class from a specified parent class) enables the representation of cases with different structures in an appropriate way.

Similarity measures for such object-oriented representations are often defined by the following general scheme: The goal is to determine the similarity between two objects, i.e., one object representing the case (or a part of it) and one object representing the query (or a part of it). We call this similarity *object similarity* (or *global similarity*). The object similarity is determined recursively in a bottom up fashion, i.e., for each simple attribute, a *local similarity measure* determines the similarity between the two attribute values, and for each relational slot an object similarity measure recursively compares the two related sub-objects. Then the similarity values from the local similarity measures and the object similarity measures, respectively, are aggregated (e.g., by a weighted sum) to the object similarity between the objects being compared.

Unfortunately, such object similarity measures are currently developed more or less in an ad-hoc fashion. It is mostly unclear, how the structure of the object-oriented case model, e.g., the class hierarchy, influences similarity assessment. Intuitively, it is obvious that the class hierarchy contains knowledge about the similarity of the objects. Objects that are closer in the hierarchy should normally be more similar to each other than objects which are more distant in the hierarchy.

However, how this knowledge relates to knowledge that could be represented in similarity measures that also consider the local similarity of the attributes is not obvious at all. Consequently, there is no clear view about how the similarity between two objects belonging to two different object classes should be determined. Therefore, many existing CBR systems and applications restrict object similarity to the comparing objects of the same object class only, not taking advantage of the high flexibility that object-oriented representations provide.

This chapter provides a framework for object similarities that allow to compare objects of different classes while considering the knowledge contained in the class hierarchy itself. We will show that knowledge about similarity contained in class hierarchies is quite similar to the knowledge contained in taxonomies of symbols, which has been analyzed in the previous chapter. The next section presents four related examples of how class hierarchies can be used and what kind of object similarities are appropriate. Based on these examples a new framework for determining object similarities is developed.

3.2 Example Use of Class Hierarchies and Object Similarities

We now describe possible uses of class hierarchies in different related application examples in which personal computers are represented as part of the case. The class hierarchy (figure 3-1) contains a class for representing a PC as well as different classes for representing components.

The PC class contains attributes like “processor”, “hard-disk”, and “price” (inherited from ”Technical Object”) which are used to describe the properties of a PC in detail. Because a PC consists of a set of components (part-of-relation) which have properties themselves most attributes are relational (printed in italic font). Like for simple attributes, it is necessary to assign a relational attribute a class, for example to express that the relational attribute “hard-disk” can only have an instance of the object class *Hard Disk*. In the example the class *Hard Disk* has no sub-classes and consequently, every object that this attribute refers to has the same structure, i.e., the same set of attributes. In contrast, the relational slot “optional storage” does not have a unique class, because the object-class *Storage Device* has several direct and indirect sub-classes. Hence, the attribute can relate to objects of different structures, but they still have a common super-class (e.g., *Storage Device*) and therefore share at least some common attributes. In our example, one PC can have a second hard-disk as optional storage device, while another PC can have a CD-ROM described by a few different attributes (e.g., type of laser) than a hard-disk.

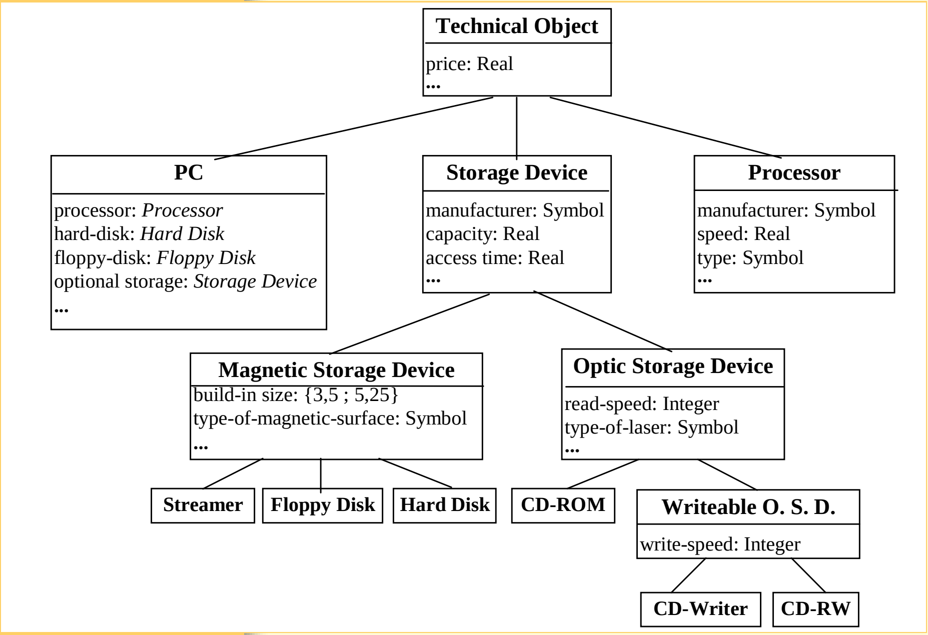

Figure 3-1: A part class hierarchy in an example domain

Consider again the "optional storage" attribute. Now we describe four examples in which this attribute is used differently. We first would like to focus on the knowledge contained in the class hierarchy and therefore don't take different values for simple attributes into account.

Example 1a

Consider a CBR system for the sales support of Personal Computers. A case represents an available PC from the stock. Consider a case c_1 with a second hard-disk as optional storage device and a case c_2 with a CD-Writer. If we assume that a customer enters a query to such a CBR system in which she/he specifies that she/he wants a CD-ROM, then c_2 is certainly closer than c_1, because a CD-ROM and a CD-Writer have obviously more in common than a hard-disk and a CD-ROM. In general, we could use a similarity measure that assesses similarity based on the distance between the class of the case object (of the respective relational attribute) and the class of the query object in the class hierarchy.

Example 1b

Imagine the customer states in the query a request for an optic storage device, i.e., in the query, the relational attribute refers to an instance of the class ”Optic Storage Device”. Then any of the devices in the *Optic Storage Device* sub-tree are perfectly suited. Hence, we expect the similarity value for the relational slot between this query and case c_2 (from Example 1a) to be equal to 1[1]. From this consideration we can conclude that whenever the class of the query object is located above the class of the case object, the similarity should be 1.

Example 2a

Consider now a trouble-shooting CBR system for PCs in which cases encode diagnostic situations and faults that have occurred previously. The domain expert describes a fault that can occur with any optic storage device. Therefore, the respective case contains an instance of the class *Optic Storage Device* in the relational attribute “optional storage device”. Now, assuming a PC user has a problem and she/he states that there is a CD-RW device in the PC, then

1. We assume that similarity measures compute values between 0 and 1.

the similarity for the respective relational slot should be equal to 1 because the case matches exactly w.r.t. this attribute. From this consideration we can conclude that whenever the class of the case object is located above the class of the query object the similarity should be 1.

Example 2b

For the same trouble-shooting example, imagine now a different query in which the user does not exactly know what kind of storage device is installed in the PC, but she/he knows that it is a writeable optic storage device. Therefore, she/he enters an instance of the class *Writeable Optic Storage Device* as attribute value in the query. Again, the case about the *Optic Storage Device* mentioned in Example 2a matches exactly because, whatever storage device the user has, we known it is an *Optic Storage.* Hence, the situation described in the case applies. However, if we consider a different case that describes a problem with a CD-RW device, then this case does not match exactly. Since we don't know what writeable optic storage device the user has (it can be a CD-Writer but it can also be a CD-RW) we expect a similarity value less than 1 to represent this kind of uncertainty. Therefore, we cannot conclude that whenever the class of the query object is located above the class of the case object the similarity should be 1.

Although these four examples are based on the same class hierarchy, it is obvious that they have to be treated differently for the similarity computation. In the query and in the cases from example 1a, only instances of classes without subclasses are used. The examples 1b to 2b make use of abstract classes

(classes with subclasses) of the hierarchy, but in each example the semantics of the use these abstract classes is different, which must lead to different similarity measures.

3.3 Computing Object Similarities

3.3.1 Basic Notions

We briefly introduce a few notions (figure 3-2) that will be further used in this chapter. Let K be an inner node of the class hierarchy, then L_K denotes the set of all leaf nodes (classes) from the sub-tree starting at K. Further, $K_1 < K_2$ denotes that K_1 is a successor node (sub-class) of K_2. Moreover, $<K_3,K_4>$ stands for the most specific common object class of K_3 and K_4, i.e., $<K_3,K_4> \geq K_3$ and $<K_3,K_4> \geq K_4$ and it does not exist a node $K' < <K_3,K_4>$ such that $K' \geq K_3$ and $K' \geq K_4$ holds.

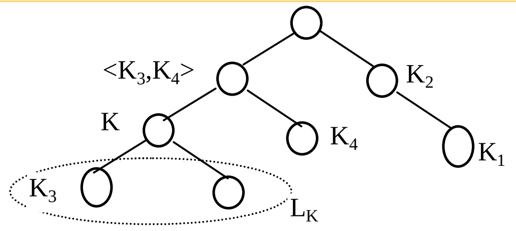

Figure 3-2: Illustration of basic notions

3.3.2 Basic Considerations about Object Similarities

In general, the similarity computation between two objects can be divided into two steps: the computation of an *intra-class similarity* SIM_{intra} and the computation of an *inter-class similarity* SIM_{inter}.

Intra-Class Similarity

The common properties of the two objects can be used to the *intra-class similarity*. For this it is necessary to take the most specific common class of the two objects and to compute the similarity based on the attributes of this class only. By considering only the attributes of the most specific common class, the object similarity computation can be done as usual, since the objects being compared are from the same class. That is, local similarities or object similarities are computed for all attributes and the resulting values are aggregated to the intra-class similarity, e.g., by a weighted sum. Formally written:

$$SIM_{\text{intra}}(q,c) = \Phi(sim_{A_1}(q.A_1, c.A_1), \ldots, sim_{A_n}(q.A_n, c.A_n))$$

where Φ is the aggregation function, $q.A_i$ and $c.A_i$ denote the value of the attribute A_i in the query and case object, respectively, and sim_{Ai} is the local or object similarity of the attribute A_i.

Inter-Class Similarity

The intra-class similarity alone would not be an adequate object similarity for the two objects. For example, in the domain shown in Fig. 1 two instances of *Hard Disk* and *CD-ROM* can have an intra-class similarity of 1, provided that they have the same values in the attributes which they inherit from their common superclass “Storage Device”. But it is obvious that there is a significant difference between a hard disk and a CD-ROM. Hence, the similarity should definitely be less than 1. It is important to note that the difference between two objects is not represented by their shared attributes but by the structure of the class hierarchy. Therefore, it is necessary to use this structure to compute an *inter-class similarity* for the two objects. This inter-class similarity represents the highest possible similarity of two objects, independent of their attribute values, but dependent on the positions of their object classes in the hierarchy. Formally, the inter-class similarity $SIM_{inter}(Q,C)$ is defined over the **classes of the objects** from the query and case being compared.

The final object similarity $sim(q,c)$ between a query object q and a case object c can then be computed, by the product of the inter- and the intra-class similarity, i.e.:

$$sim(q,c) = SIM_{\text{intra}}(q,c) \cdot SIM_{\text{inter}}(class(q), class(c))$$

where class(q) and class(c) denote the object class of the object q and c, respectively.

Next, we analyze how the inter-class similarity should be determined, which is quite similar to the similarity computation between two symbols arranged in a taxonomy.

3.3.3 Different Semantics of Nodes

In a class hierarchy as well as in a taxonomy of symbols, we must distinguish between leaf nodes and inner nodes. In a taxonomy leaf nodes represent *concrete objects* of the real world. Inner nodes, however, describe *classes* of real world objects. An inner node K represents a class with certain properties that all of the concrete objects from the leaf nodes L_K have in common. Unlike classes that occur in the object-oriented paradigm, the classes that are represented by the inner nodes of a taxonomy are not described intentionally by a set of properties, but extensionally through the set of concrete objects L_K that belong to the class. Therefore, an inner node K stands for the set L_K of real world objects.

If we look at the class hierarchy shown in Fig. 1, we can notice a difference in the semantics of its nodes compared to the semantics of taxonomy nodes. While a leaf node of a taxonomy represents a concrete object of the real world, a leaf node of a class hierarchy is naturally a class and therefore represents a set of objects. As shown above, inner nodes of a taxonomy describe classes of real world objects, but if we look at the inner nodes of class-hierarchies, we can see that these nodes represent abstract classes. Because of this, such a node does not represent a set of real world objects, but a set of *abstract objects*. The instances which belong exclusively to the class “Storage Device” or “Optic Storage Device” for example are obviously not objects of the real world. However, abstract objects are sets of real world objects. An instance of “Optic Storage Device”, for example, can be used as abbreviation for the set of all instances of the classes “CD-ROM”, “CD-Writer”, and “CD-RW” that have the same attribute-values in the common attributes as the respective “Optic Storage Device” instance, e.g., the same manufacturer, the same capacity, the same access time, and the same speed.

There is also a difference in the use of the two different structures. A taxonomy tree consists of the symbols that are directly used as values for the attributes. On the other hand, the classes of a class hierarchy are not used as values for the relational slots themselves, but the instances of the classes. If we take this fact into account, we will see that now there is no difference in the semantics of the corresponding values, because the taxonomy symbols must be compared with the instances and not with the classes of the class hierarchy. An instance of a class without subclasses (a leaf node of the hierarchy) represents a concrete object of the real world, and as we have seen before an instance of an abstract class (inner node) can be treated as a set of real world objects. This semantics is equivalent to the semantics of the taxonomy nodes. Therefore, it is possible to apply the similarity measures used to compute similarities between taxonomy symbols for computing of the inter-class similarity between objects.

3.3.4 Inter-Class Similarity Between Concrete Objects

A class hierarchy encodes some knowledge about the inter-class similarity of the real-world objects, i.e., the instances of the leaf nodes. The deeper we descend in the class hierarchy, the more features the instances of the classes will have in common. We can therefore define the inter-class similarity as a measure of how many ”features”[1] the compared objects have in common. The more ”features” are shared, the higher is the inter-class similarity. For example, the inter-class similarity between a CD-Writer and a CD-RW is higher than the inter-class similarity between a CD-Writer and a CD-ROM.

This consideration leads to the following general constraint for defining the inter-class similarity for the instances of the leaf nodes of a class hierarchy:

1. Here, feature does not necessarily mean attribute in the case representations.

$$SIM_{\text{inter}}(K,K_1) \le SIM_{\text{inter}}(K,K_2) \quad \text{IF} \quad \langle K,K_1\rangle > \langle K,K_2\rangle$$

Because the class hierarchy only represents the constraint shown above but does not define numeric values for the similarity between two leaf node objects (that are used for the computation of an object similarity), it is necessary to add additional knowledge to the hierarchy. For this purpose it is possible to annotate every inner node K_i with a similarity value $S_i \in [0..1]$, such that the following condition holds: if $K_1 > K_2$ then $S_1 \le S_2$. The semantics of the similarity value is as follows:

The value S_i represents a lower bound for the inter-class similarity of two arbitrary instances of the classes from the set L_{Ki}, or formally written:

$$\forall X,Y \in L_{Ki}\, SIM_{\text{inter}}(X,Y) \ge S_i$$

With regard to this semantics one may define the inter-class similarity between two objects as follows:

$$SIM_{\text{inter}}(K_1,K_2) = \begin{cases} 1 & \text{if } K_1 = K_2 \\ S_{\langle K_1,K_2\rangle} & \text{otherwise} \end{cases}$$

3.3.5 Semantics and Inter-Class Similarity of Abstract Objects

If we now recall again the examples that we have presented in section 2.2, it is obvious that the ”optional storage” attribute must be treated differently in the different examples, although they all use the same class hierarchy. From that it becomes clear that some additional knowledge which we have not yet discussed plays a role during similarity assessment. However, this knowledge is not contained in the class hierarchy itself.

The knowledge that we are looking for is the knowledge about the semantics of the instances of inner nodes, i.e., the semantics of the abstract objects, which can be treated as sets of real-world-objects (see section 3.3.4). In our example, the question is: what does it mean when the case or query contains the statement:

optional storage: <an Optic Storage Device instance >

In fact, there are different interpretations of this statement that will be further discussed.

Any value in the query

The user specifies an abstract object *k* in the query. This means that she/he is looking for a case that contains a real-world-object that matches with the features of the specified abstract object, i.e., a case that contains an object that belongs to a class of L_K. This was the situation in example 1b.

Any value in case

The case contains an abstract object *k*, which describes a situation in which the case is valid for all objects that are a specialization of *k*. This leads to a kind of generalized case. This occurred in example 2a.

Uncertainty

This situation differs significantly from the previous ones. Here, the use of an abstract object *k* means that we do not know the concrete object for this relational slot, but we know that it is a specialization of *k*. This situation occurred in example 2b.

Depending on the appropriate semantics we can now define an inter-class similarity measure $SIM_{inter}(Q,C)$ which computes a value for the inter-class similarity between two objects Q and C where each can be either a leaf node (concrete object), an inner node (abstract object) with the ”any value” interpretation or an inner node (abstract object) with the ”uncertainty” interpretation. This leads to nine possible combinations shown in Table 1. Seven of the nine combinations in the table are marked with a roman number that is further used to reference the formulas for computing the similarity. These are the ones that occur most likely. However, the following considerations can easily be extended also to the two missing combinations.

Table 3-1: Combinations of different semantics for objects in query and case

Query \ Case	Leaf Node concrete object	Any Value abstract object	Uncertainty abstract object
Leaf Node concrete object	I	II	V
Any Value abstract object	III	IV	
Uncertainty abstract object	VI		VII

I: Only the similarity between concrete objects must be computed as described in section 2.3.4.

II: The query contains a concrete object and the case contains an abstract object (inner node) representing a set of concrete objects each of which is a correct object for the case. Therefore, the use of this abstract object in the attribute is a shortcut for the use of several cases, one for each concrete object belonging to the abstract object. Since we are looking for the most similar case in the case base, the object similarity, and therefore also the inter-class similarity, between the query and our case containing the abstract object is equal to the highest similarity between the query and one of the concrete objects. Hence, the following holds:

$$SIM_{\text{inter}}(Q,C) = \max\{SIM_{\text{inter}}(Q,C') \mid C' \in L_C\} = \begin{cases} 1 & \text{if} \quad Q < C \\ S_{\langle Q,C \rangle} & \text{otherwise} \end{cases}$$

This definition ensures that the similarity is the same as the similarity that arises when each of the concrete objects would have been stored in the case base. This measure is appropriate for example 2a.

III: Here, the specification of this abstract object can be viewed as a shortcut for posing several queries to the system, one for each of the concrete objects from the set that the abstract object represents. Since we are again interested in the most similar case, we should again select the most similar concrete object from the set. Hence, the following holds:

$$SIM_{\text{inter}}(Q,C) = \max\{SIM_{\text{inter}}(Q',C) \mid Q' \in L_Q\} = \begin{cases} 1 & \text{if} \quad C < Q \\ S_{\langle Q,C \rangle} & \text{otherwise} \end{cases}$$

This measure is appropriate for example 1b.

IV: This is a combination of II and III. We are looking for the highest possible similarity between two concrete objects from the two sets represented by the abstract objects since both, the query and the case, represent alternatives that are suited equally well. Hence, the following holds.

$$SIM_{\text{inter}}(Q,C) = \max\{SIM_{\text{inter}}(Q',C') \mid Q' \in L_Q, C' \in L_C\} = \begin{cases} 1 & \text{if} \quad C < Q \text{ or } Q < C \\ S_{\langle Q,C \rangle} & \text{otherwise} \end{cases}$$

V: The case contains an abstract object which represents a set of concrete objects from which only one value is actually correct for the case, but we don't know which one. Therefore, our similarity measure has to reflect this lack of information. There are three possible approaches: we can assess the similarity in a pessimistic or optimistic fashion, or we can follow an averaging approach. We only demonstrate the pessimistic approach; see (Bergmann, 1998c) for more details on the other approaches.

Pessimistic approach: We assess the similarity between the known object (in the query) and the partially unknown object (in the case) by computing the lower bound for the similarity as follows:

$$SIM_{\text{inter}}(Q,C) = \min\{SIM_{\text{inter}}(Q,C') \mid C' \in L_C\} = S_{\langle Q,C \rangle}$$

VI: The uncertain information is contained in the query; the information in the case is certain. This case is quite similar to the previous case V. For the pessimistic approach holds:

$$SIM_{\text{inter}}(Q,C) = \min\{SIM_{\text{inter}}(Q',C) \mid Q' \in L_Q\} = S_{\langle Q,C \rangle}$$

VII: The uncertain information is contained in the query and in the case. The similarity is computed as follows for the pessimistic approach:

$$SIM_{\text{inter}}(Q,C) = \min\{SIM_{\text{inter}}(Q',C') \mid C' \in L_c, Q' \in L_Q\} = S_{\langle Q,C \rangle}$$

In all of these cases, the inter-class similarity can be computed very easily by determining the position of the class of the query object and the case object in the class hierarchy and by looking up the similarity value associated with the most specific common super class.

3.4 Summary

The described object similarities are realized as part of the recent version of CBR-Works 4. The approach was applied for the CBR-Works Support Center (hotline support for troubleshooting Workstations and CAD software).

Currently, there is no other work that proposes similarity measures for object-oriented case representations that make use of the class hierarchy, relational attributes, and flexible local similarity measures for simple attributes. However, similarity measures for different kinds of structured representations are discussed throughout the CBR and instance-based learning literature during recent years.

To some extend, object-oriented representations can be compared to representations in first-order logic where a case is a conjunction of atomic formulas. Each atomic formula $P(id,a_1,...,a_n)$ stands for a single object. The argument *id* of the formula denotes an object identification and the remaining arguments $a_1,...,a_n$ represent the attributes. Relational attributes can be represented by using the object identifications as attribute values.

4 Rules for CBR

When problems are solved through reasoning from cases, the primary kind of knowledge is contained in the specific cases which are stored in the case base. However, in many situations additional general knowledge (we call it *background-knowledge*) is required to cope with the requirements of an application. In CBR-Works, such general knowledge is integrated into the reasoning process in a way that it complements the knowledge contained in the cases. This general knowledge itself is not sufficient to perform any kind of model-based problem solving, but it is required to interpret the available cases appropriately.

Background knowledge is expressed by two different kinds of rules:

- *Completion rules* are formalised by the knowledge engineer during the development of the descriptive model. They describe how to infer additional features out of known features of an old case or the current query case.
- *Adaptation rules* are formalised by the knowledge engineer during the development of the descriptive model. They describe how an old case can be adapted to fit the current query.

4.1 Introduction

General knowledge (rules) is sometimes available and necessary to better explore and interpret the available cases. Such general knowledge may describe constraints which directly lead to the exclusion of a case for reasoning. An other kind of general knowledge may state a strict dependency of one feature of a case on several other features of the same case. This allows to infer additional, previously unknown features from the known ones. Furthermore, some applications require an adaptation of a retrieved case according to the actual problem at hand. Therefore, general knowledge is required to specify such an adaptation. Even if adaptation abilities are much more essential for synthetic tasks such as design or planning, several applications from the field of classification, diagnosis, or decision support as addressed in CBR-Works also require at least some minimal adaptation capabilities.

To use background knowledge should not affect the previously developed methods for integrating induction and case-based reasoning. It is aspired to keep the required inference mechanisms which handle the background knowledge mostly independent from the kind of integration that is chosen for a specific application. Moreover, it is crucial to avoid increased retrieval times as a consequence of a search-intensive inference procedure. The primary type of reasoning is still a combination of case-based and inductive reasoning. The background knowledge is not intended to be a substitution for the knowledge contained in the cases but an addition of general knowledge to the specific knowledge of the cases.

4.2 Representing and Using Background Knowledge

In CBR-Works, background knowledge is expressed in the form of rules of different kinds. This section first introduces the kinds of rules which we have identified useful. Then, the impact of the object-oriented case-representation paradigm followed by CBR-Works on the specific representation and interpretation of the rules is discussed in general. Thereafter, the detailed representation of all kinds of rules is specified and the related semantics are explained. Finally, methods for efficient rule interpretation are presented.

4.2.1 Kinds of Rules

In CBR-Works we have identified two kinds of rules to be essential:

- *Completion rules* infer additional features out of known features of an old case or the query. Thereby, these rules complete description of a case.
- *Adaptation rules* describe how an old case can be adapted to fit to the current query.

In the following we will explain these two kinds of rules informally before going into the details of their representation and processing.

4.2.1a Completion Rules

In several situations, features of a case description are directly dependent on several other features. When the user enters some of the features in the query, she/he should generally not be demanded to enter the values of features which are absolutely determined by the information she/he has already entered. But not having these values in the query case leads to a less informed similarity assessment. Therefore, we propose *completion rules* to extend the description of a case (see figure 4-1). These rules apply to the cases of the case base as well as to the query case which is entered during consultation. Completion rules are used to infer values of attributes of the case description which are directly dependent on some other attributes of the case.

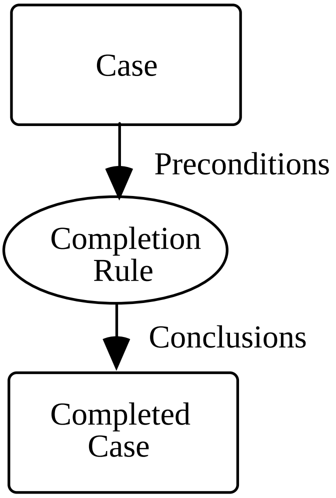

Figure 4-1: Completing case descriptions

Thereby, additional attributes can be assigned a value without asking the user. Furthermore, the occurrence of inconsistent values can be reduced. The attributes which are derived using the completion rules can then be used during the similarity assessment. Such a similarity assessment is based on the knowledge of more attributes and should consequently be more precise. Since the completion rules will be used to derive attributes of a case description which the user might also enter, the rules must be known to be true in all situations. Uncertain, or just probable rules are not considered here. Therefore, an inference drawn using the rules is always considered absolutely correct and cannot be changed by the user.

Figure 4-1 shows these preconditions and conclusions of a completion rule. The rule is based on the values of attributes given in a specific case and as a result of the application the rule may add certain attribute values to this case.

Informal example

As an example, recall the travel agency domain introduced already. Assume a case representation for a journey which includes the specification of the number of adults and the number of children which are involved in the journey. Moreover, assume that the representation also specifies the total number of persons because for several journeys only the total number of people is important (e.g. in an apartment). In this situation, the following general rule is useful:

> The total number of persons is always the sum of the number of children and the number of adults.

This rule avoids to enter the value for the total number of persons when the number of children and the number of adults is already entered by the user. In a similar manner, completion rules may also be used to compute the number of rooms which are required for a certain number of people.

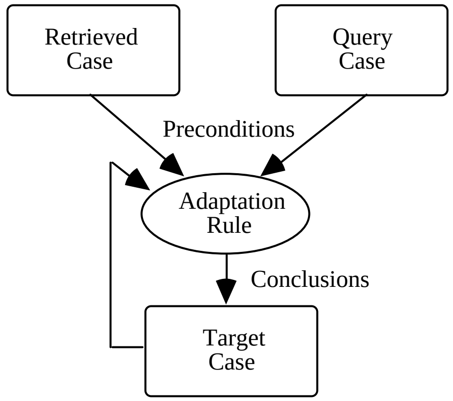

Figure 4-2: Cases used for Adaptation

As shown in figure 4-2, adaptation rules combine attributes of the retrieved case, attributes of the current case, and already derived attributes of the target case in the precondition of the rule. In a rule's conclusion, the attribute values for the target case are derived.

Informal example

Imagine once again a situation from the travel agency domain. Suppose, the user's query specifies a journey with a duration of *two weeks*. Furthermore, assume that the most similar case which satisfies the user best is a journey which usually takes *one week*. Since the price for this journey is calculated on a one week basis, it must be adapted to correctly refer to the two week journey as specified in the query. For this purpose, the following adaptation rule is useful:

> **If** the duration specified in the query case is longer than the duration specified in the retrieved case, **then** the price specified in the target case is computed by adding the price of the retrieved case and the price for accommodation for the additional time period.

Since the transportation costs are already included in the retrieved case adapting the price for a two week vacation by just adding the additional accommodation costs is a useful adaptation strategy in the travel agency business.

4.2.2 Components for Handling Background Knowledge

Figure 4-3 shows how these kinds of rules are used together with the CBR-Works system. Two additional components are required for the processing of the rules. One component is necessary for the completion of case descriptions. This component is used to complete previous cases before they are stored in the case-base and to complete the query case which is entered by the user. The CBR-Works system then works only on completed cases. The task of the second component is the adaptation of one of the retrieved cases to the query case. This component computes a target case (solution) out of the retrieved case and the current completed query case. This target case is then completed and stored again into the case-base for future use.

In the following we want to characterise the two different kinds of rules in general before discussing their detailed representation and processing.

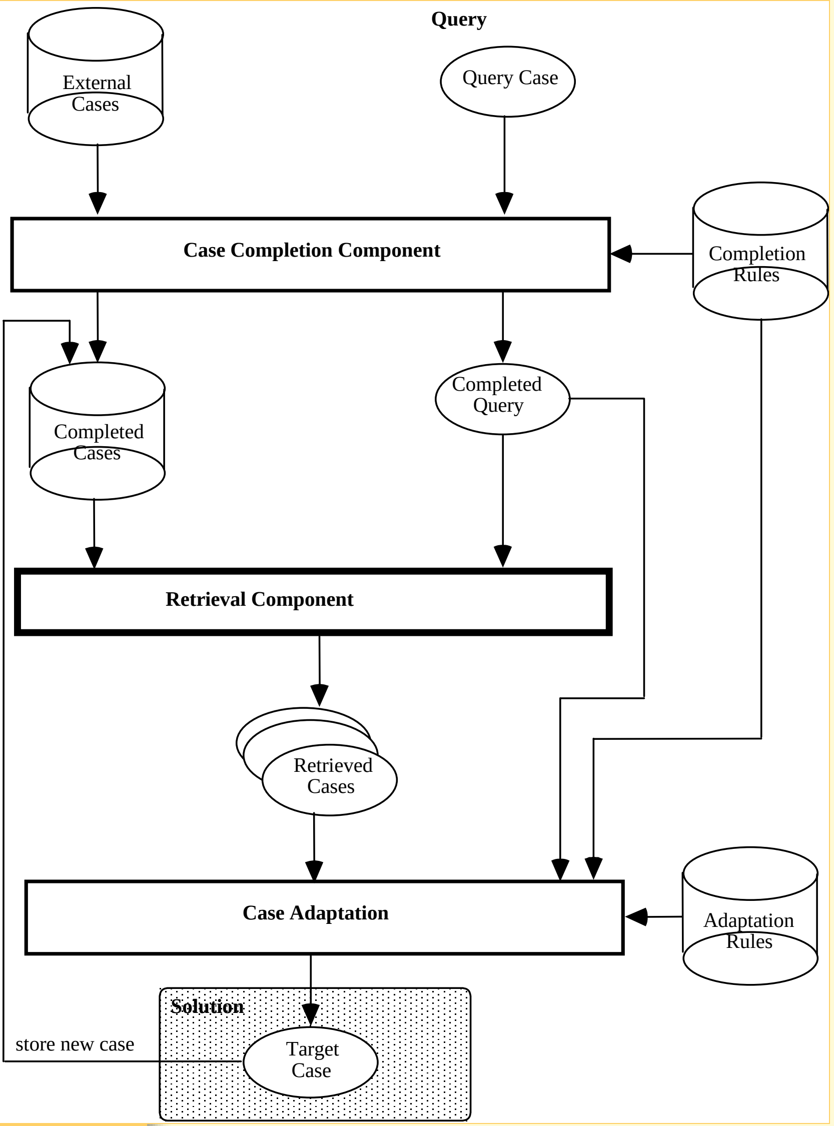

Figure 4-3: Components for processing Background Knowledge

4.2.3 Impact of the Object-Oriented Case Representation

The case representation of the whole CBR-Works system is *object-oriented*. The object structure itself is defined within the descriptive model. It supports inheritance between *classes* as well as arbitrary *relations* between two objects. In the following, we want to make the distinction between the inheritance and arbitrary relations more clear. Figure 4-4 gives an example of an *inheritance hierarchy*.

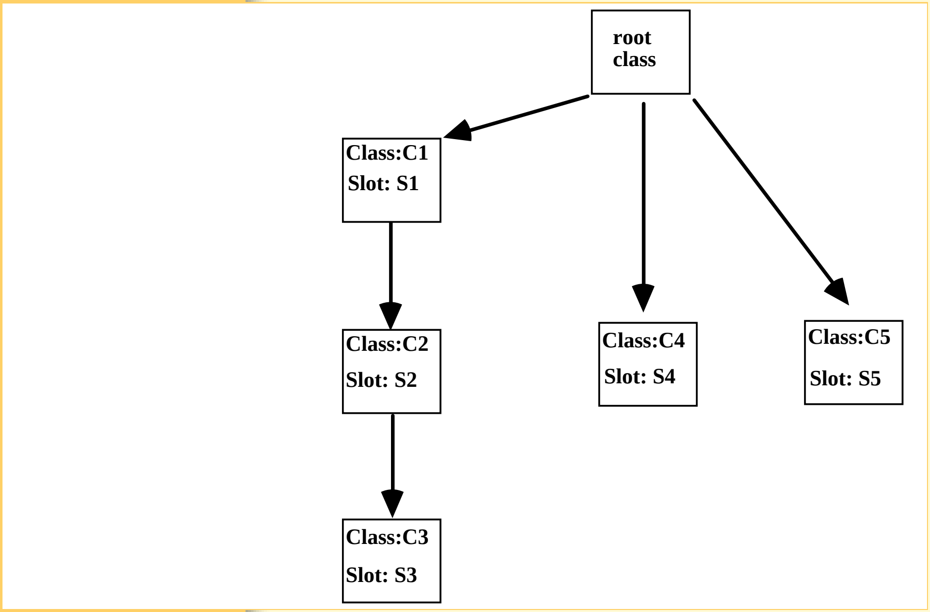

Figure 4-4: Example of an inheritance hierarchy

This figure shows five classes, which are derived from the root class of the hierarchy. The classes C1, C4 and C5 are directly derived from the root class. Objects of these classes only contain the slots which are directly specified in the respective class. No slots are inherited from the root class. Inheritance becomes relevant for the classes C2 and C3. The class C2 inherits the slot S1 from class C1 and class C3 inherits the slots S2 from C2 and S1 from C1.

The mechanism of inheritance has to be clearly distinguished from the ability to specify *relations* between different kinds of objects. In the descriptive model, a relation is declared by a relational slot. A relational slot is a slot which does not hold a basic value but a whole object of some class. For example, Figure 6 shows two classes C2 and C4 with relational slots. The slot R1 of class C2 can hold an object of the class C4 and the slot R2 of class C4 can hold an object of the class C5.

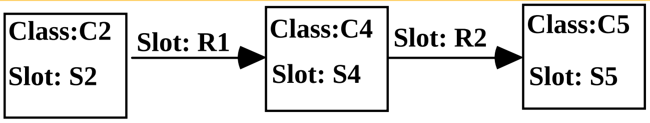

Figure 4-5: Example of Relational Slots

In a descriptive model of a domain, inheritance and relations usually occur simultaneously. So, relational slots are inherited from a superclass in the same manner as slots which hold a basic value are inherited. If the inheritance as shown in figure 4-4 and the relations as shown in figure 4-5 are specified simultaneously, then the class C3 also inherits the relational slot R1 from its superclass C2.

Obviously, this object-oriented representation has a strong impact on the detailed mechanisms which handle the rules. Within this kind of case representation, the classes are the most natural place to attach the rules. Within the scope of a class, a rule has a direct access to the slots which are defined for that class and to those slots which are inherited from its superclasses. Additionally, rules must be given access to slots of those objects which are related to the object the rule belongs to. In the same manner as slots are inherited from the supercass to a class, the rules can also be *inherited*. Rules which are defined for a superclass are always valid for all subclasses.

Figure 4-6 shows an example of the simultaneous occurrence of the inherited and related objects already shown in the figures 4-4 and 4-5. Additionally, the figure shows different rules which are attached to the classes and specifies the slots to which these rules have access to. The figure shows five different classes C1,..., C5 where C2 is a subclass of C1 and C3 is a subclass of C2. Each class has one none-relational slot, i.e. slots which can hold values of basic types, but not objects. These slots are named S1,..., S5 respectively. Moreover, class C2 and class C4 have relational slots R1 and R2, respectively. To illustrate the scope of the rules associated with the five classes, the slots that can be accessed by each of the rules are shown. For example, we can see that Rule-2 has access to the slots of its own class (S2), to the slots of its superclasses (S1), and to the slots which are available in related classes (S4, S5). To make a precise reference to slots of related classes, the relation itself (e.g. R1) must always be noted together with the respective slot (a possible notation would be: R1->S4 or R1->R2->S5). Due to the inheritance of the rules, Rule-1 is also valid for all objects of the classes C2 and C3, but of course not for objects of the classes C4 and C5 since class C1 is not a superclass of C4 and C5.

Exploring the object-oriented representation also for the rules enables an efficient way of expressing background knowledge. Due to the rule inheritance, knowledge which applies to many different objects can be expressed in rules which are attached to the respective superclass these objects belong to. Moreover, the restricted set of slots a rule can access still maintains the principle of information encapsulation of object-oriented representations.

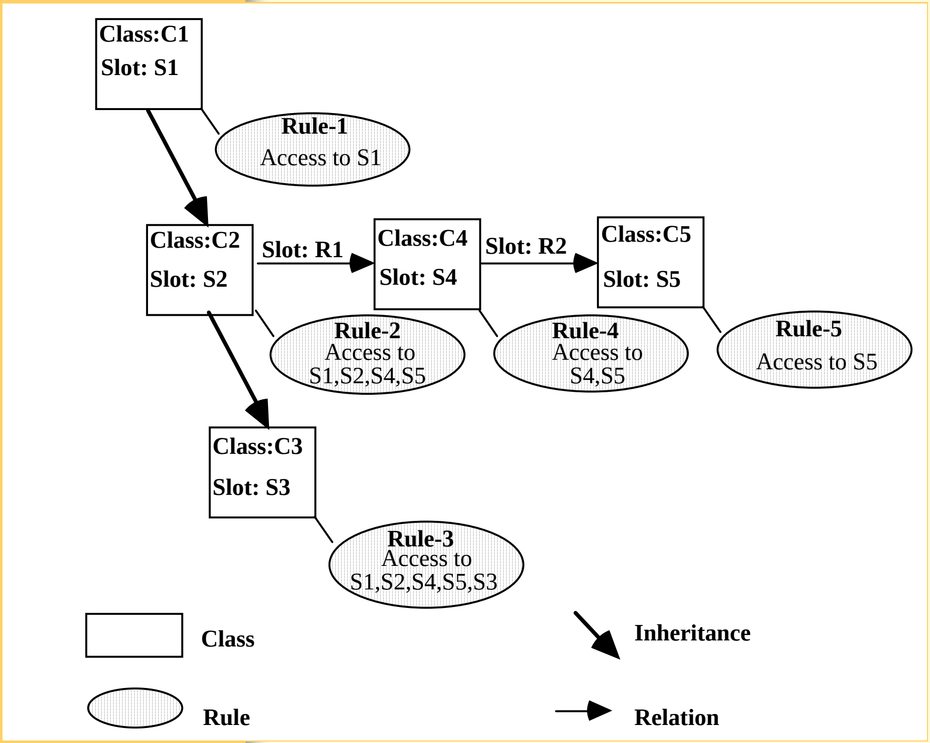

Figure 4-6: Example: Scope of Rules in the Object-Oriented Case Representation

4.3 Detailed Description of Rules

In the following, we want to explain the representation of the two kinds of rules in more detail.

4.3.1 Completion Rules

Sets of rules for classes

Rules for completing the description of an old case or the current case are attached to the classes defined in the descriptive model of the case representation. Each class in the descriptive model can have an arbitrary number of completion rules. All completion rules that are present for a class are always active. Each rule applies to all objects of this class in a case. The set of rules must be consistent in the sense that it is not allowed for two or more rules to infer contradictory values for the same slot in the same case. In general, the consistency of a descriptive model cannot be checked automatically in advance. However, the event of an inconsistency must be checked in the running system and a respective error report must be given to the user.

Components of a rule

Each completion rule consists of two parts: a *precondition part* and a *conclusion part*. The precondition part defines a *conjunction* of *conditions*. Each condition must be expressed in terms of the accessible slots with respect to the class to which the rule belongs to. A condition can compare the value of a slot with respect to values of other slots, constants, or local variables. Moreover, the precondition can also be used to specify an arbitrary function which calculates a new value using the existing slot values. The conclusion part of a rule consists of a set of *ac-*

tions which are executed if the precondition is fulfilled, i.e. all conditions in the precondition are fulfilled. An action in the conclusion of a rule can assign a value to a slot or it can create a new object for a relational slot.

Precondition part of a completion rule

The precondition of a rule consists of a set of conditions. The set of these conditions is treated as conjunction, i.e. all conditions of the precondition of the rule must be fulfilled to fire the rule. Additionally, a condition may occur in negated form. Then, this condition must not be true to fire the rule. Local variables may occur in the precondition of a rule. These variables can become instantiated by a certain condition and can be accessed or tested in conditions evaluated afterwards in the same rule.

A condition can be of three different types:

- *Built-in predicates*:
  A condition can be expressed by using a built-in predicate to compare two values. The two values to be compared can be any slot that lies within the scope of the rule, any constant value, or any local variable (see below). However, one obvious restriction is that the two values to be compared are of the same type or from the same class of objects. Dependent on the type of values, different built-in predicates are available.

- *External functions and predicates*:
  External functions and predicates can be used to define any kind of user-specific predicates which cannot be expressed by the built-in conditions. The external functions must be imple-

mented in the underlying programming language of the CBR-Works-system. In addition to defining a predicate for testing certain conditions, external functions can also be used to compute a new value which can be returned to the rule for an assignment to a slot.

- *A-kind-of test:*
  Using the object-oriented features of the case-representation, a related object can be from different classes. However, by the definition of a relational slot, an object's class is already specified, but objects of all respective subclasses are valid objects for such a slot. Therefore, the a-kind-of test can be used to examine the actual class of a related object.

Built-in predicates

For the definition of the conditions, several built-in predicates are available:

The *equality* (=) and *inequality* (<>) predicate can be used to compare any two values or objects of the same type or class. For *basic values* (integer, real, string, symbol, ordered_symbol, taxonomy, boolean, date, and time), the equality predicate evaluates to *true* if the two values to be compared are identical. For two *objects*, the equality predicate evaluates to *true* if both objects are from the same class and all filled slots of the two objects are themselves equal. The equality of the filled slots is defined by applying this equality definition recursively. If the equality-predicate is used for a slot which does not hold any value, the predicate evaluates to *false*. The *inequality predicate* expresses negation of the equality predicate. It evaluates to true if the equality predicate evaluates to false and vice versa.

The *less-than* (<, <=) and *greater-than* (>, >=) predicates can be used to compare two values from ordered or partially ordered basic types. These types are: integer, real, ordered_symbol, taxonomy, date, and time. The predicates cannot be used to compare symbols, booleans or objects. While the definition of the predicates is obvious for ordered types, the interpretation of the order for taxonomies needs to be explained. We define a value x to be less than y (x < y) if and only if y is below x in the taxonomy.

The *set inclusion* (in) predicates can be used to compare two values from an interval type. The condition: *I1 in I2* holds if and only if the lower bound of I1 is greater or equal than the lower bound of I2 and if the upper bound of I1 is less or equal than the upper bound of I2.

Variables in rules

Variables may also occur in the precondition of a rule as a means of sharing values between different conditions contained in the precondition of the same rule. These variables are always local to the current rule. Variables can hold any kind of basic values, objects, or it can hold the class name of an object only. Variables become instantiated by the first (left-most) condition which is either an equal-predicate (=), an a-kind-of test, or an external function which calculates and returns a new value. An equality predicate can instantiate the variable with the current value of the slot (a basic value or a whole object). The a-kind-of-test assigns the variable with the *name of the class* of the tested relational object. The external function intantiates the variable with the value which is computed by this function. An instantiated variable can then be used in any further conditions. Moreover, a variable can also be used in an action of the conclusion part of the rule. The value of the variable can then be assigned to a new slot. Moreover, a new object of a class contained in a variable can be created and assigned to a slot.

Conclusion part of a completion rule

The conclusion of a rule consists of a set of actions. An action can be either the assignment of a value to a slot or the creation of a new object which is stored in a relational slot:

- *Slot assignment*:
  A slot can be assigned a constant value, the value of another slot, or the value of a variable which was instantiated in the precondition of a rule. If a value is assigned to a slot, then the slot must not have a value before or the value it currently has must be the same value that the value the rule tries to assign to the slot. The situation in which two different rules can fire and assign a different value to the same slot is explicitly forbidden. Such a situation would indicate an inconsistency within the set of rules. Please note that rules which include some kind of "uncertainty" are not the target of this background-knowledge task. All rules are assumed to lead to "certain" consequences and should therefore not lead to contradictory values. However, a contradiction between a rule inference and a wrong value entered by the user may still occur. In this situation, an inconsistency in the users query must be assumed and reported to the user. This property can be employed to implement different

kinds of consistency checks for the current user query.

- *Creation of objects*:
  The second kind of action that may occur in the conclusion of a rule is the creation of new objects for relational slots. This is necessary to be able to extend the object structure of a case itself. With the creation of a new object, the name of the class of the object must be specified. The name of the class can be stated by specifying the name directly or by selecting a variable which is instantiated by an a-kind-of condition in the precondition of the same rule. If the relational slot for which the object should be created is still empty, then the new object is created (with empty slots) and directly linked to the slot. If the relational slot already contains an object, then this object must be of the same class or it must be a super-class of the object which should be created. If the latter is the case, the existing object is replaced by the more specific (sub-class) object which is to be created, but the filled slots of the old object are directly copied into the same slots of the new object. It is not allowed to create an object which is of a completely different class than the object which the slot already contains. Such a situation is also an indication for an inconsistency in the rules or for an inconsistent user query.

4.3.2 Adaptation Rules

The basic difference between completion rules and adaptation rules is that completion rules only refer to one case, while adaptation rules always refer to three cases, namely the query case, the retrieved case, and the target case (see also figure 4-2). These three different cases have to be taken into account when specifying the preconditions and the conclusion of adaptation rules.

Precondition part of an adaptation rule

As for completion rules, the precondition of an adaptation rule also consists of a conjunction of conditions which are specified by predicates over certain slot values, constants, or variables. For adaptation rules, the same three kinds of conditions do exist as for the completion rules, namely built-in predicates, external functions and predicates, as well as a-kind-of tests for relational slots. Each time a slot is referenced in a condition, it must be explicitly stated out of which case this slot has to be taken. Slots can be taken from the retrieved case, the query case, or also from the target case. The slot values which are from the retrieved case are those which are stored in the most similar case that is retrieved. The slot values which are from the query case are those values which are specified by the user as a query, and the values which are from the target case are values which have been already determined by other adaptation rules.

Conclusion part of an adaptation rule

As for completion rules, the conclusion part of an adaptation rule also consists of a set of actions. An action can be either the assignment of a slot or the creation of a new object. Within an action of an adaptation rule, only the target case can be modified, but not the retrieved case or the current case.

Different object structures for the retrieved case and the current case

As already stated, rules are directly connected to the classes of the case description. For the completion rules, each rule attached to a class simply applies to all objects of this class which are contained in the case description. But for adaptation rules, objects out of three different cases must be accessed within the precondition of a single rule. If there is more than one object of the same class in a case, the question arises which of the objects of the retrieved case, the query case, and the target case are addressed within the precondition of a rule.

To cope with this problem, we introduce the additional requirement that the three objects (from the retrieved case, the current case and the solution case) must all occur in the same "context". Two objects from two different classes are in the same context if

- both objects are the root object of the respective case or
- the two objects can be reached by following the same relational slots starting from the root objects of two cases.

This definition is illustrated in figure 4-7, where the object structure of two cases C and C' is shown.

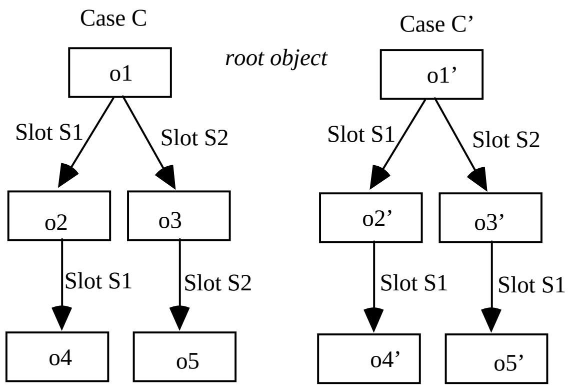

Figure 4-7: Example Object structure for two cases

Each case consists of five objects, and we want to assume that all objects are instances of the same class. In this example, the objects o1 and o1’ are in the same context because both are the root object of their case. Furthermore, the objects o2 and o2’ are in the same context because they can both be reached via the same relational slot S1 from the root object. Following the same argument, o3 and o3’ are also in the same context as well as o4 and o4’. But o5 and o5’ are not in the same context because in case C, o5 must be accessed from o3 via the slot S2 while o5’ must be accessed from o3’ via the slot S1.

5 Maintaining Case–Based Reasoning Systems

This chapter describes an architecture which supports the user of a CBR system during the modelling and maintaining of the used knowledge. Different maintenance operations are described and charac–terised along different dimensions. We give an overview of possible oper– ations with their resulting repair strategies. Exemplary, we describe two operations in detail. We examine the impact of maintenance operations to the overall CBR system which leads to the design of an evaluation component. As a result, we describe our architecture for the maintenance of a CBR system. We close with a short discussion.

5.1 Introduction

Much research and implementation effort has been spent on building Case–Based Reasoning (CBR) systems, but only few on their maintenance. Now, as the systems are commercially used, the need for maintenance is a key issue for overtime success.

Maintenance was completely underestimated during the first commercially usage of expert systems, for example rule-based systems. These systems were working at laboraties in a research environment with small rule bases but were not maintainable in large business applications for “real world” problems. Also recent research projects identified a need for maintenance support for knowledge– based systems.

This chapter presents an approach for the maintenance of CBR systems and a resulting architecture. In a first attempt, we focus on CBR systems for classification tasks without an adaptation component. The goal of such an architecture is the support of maintenance operations and repair operations during changes of the system. Maintenance operations are made by the user in order to modify the system in an intended manner. Repair operations are performed after a maintenance operation has happened to keep the system consistent. Further, an evaluation component is offered which enables the user to estimate the effect of the maintenance operation on the system. As a result, the lifetime of a CBR system can be extended and the modelling and maintenance costs are reduced.

First, we introduce a formal terminology which enables us to formalise possible maintenance operations which occur. The operations are characterised by a set of dimensions in order to specify their impact on the CBR system. Next, we give two detailed examples of maintenance operations and we define our resulting maintenance architecture in detail.

5.2 Knowledge Representation

Maintenance is the execution of operations which change the domain schema and the case base in order to eliminate former modelling errors or to update the system according to changes of facts in reality. The domain schema describes “how” the cases look like and the case base represents the known cases from the past. First, we will define the domain schema and the case base.

5.2.1 The Domain Schema and the Case Base

We assume an object–oriented approach with a domain schema $DS = \{o_1,\ldots, o_m\}$, where the concepts[1] o_i represent concrete entities in the real world and are probably linked by inheritance and has-part relations. The case base $CB = \{c_1,\ldots, c_n\}$ consists of a set of cases c_i . These are instances of a special concept $o_r \in DS$ (root concept), which defines the complete schema of a case.

Each concept $o_i = \{n_{o_i}, SIM_{o_i}, f^i{}_1, \ldots, f^i{}_{l_i}\}$ consists of the name n_{o_i} , a set of possible local similarity measures SIM_{o_i}[2] and a finite set of features f_i for each concept.

The root concept $o_r = \{n_{o_r}, SIM_{o_r}, f^r{}_1, \ldots, f^r{}_{l_r}, d\}$[3] has an additional feature d, which contains the class for each case.

Let $T = \{t_1, \ldots, t_p\}$[4] be a set of atomic types and the above mentioned domain schema DS the set of all possible complex types.

1. Contrary to objects a concept is only a set of attributes without according methods
2. One similarity measure out of SIM_{o_i} is marked specially with a hat. It is actually used for the similarity calculation.
3. $d \in D = \{d_1, \ldots, d_x\}$, D is the set of possible classes. We assume that the class is defined in the root concept which does not limit our approach

If $F = \left\{f^1_1, \ldots, f^1_{l_1}, \ldots, f^m_1, \ldots, f^m_{l_m}\right\}$ is the set of all features, then a feature $f_i \in F$ is defined as a triple $f_i = (n_{f_i}, e_{f_i}, w_{f_i})$ where n_{f_i} is the name of the feature, w_{f_i} specifies a weight which denotes the importance of the feature used for similarity calculation and $e_{f_i} \in T \cup DS$ is an atomic or complex type. Atomic types are all predefined types by the CBR system like Integer, Real, Symbol, Boolean, etc. If a feature has a complex type this represents a has–part relation in the domain schema. Furthermore, inheritance results from an is-a relation between concepts and means that one concept inherits all features from all superconcepts.[1] The domain schema and the case base define the complete domain model.

5.2.2 Similarity and Retrieval

After we have defined our domain schema and the case base, we have to specify how similarity is calculated during the retrieval. A query $q = \{q_1, \ldots, q_{l_1}\}$ is an instance of the root concept

4. $t_i = (n_{t_i}, v_{t_i}, SIM_{t_i})$ where n t i specifies the typename, v t i the value–range for the type and SIM t i a set of possible local similarity measures in which the activated similarity measure is marked specially by a hat.
1. If o_j is-kind-of o_i , that is concept o_j inherits from superconcept o_i , and $o_i = \{n_{o_i}, sim_{o_i}, f^i{}_1, \ldots, f^i{}_{l_i}\}$ then $o_j = \{n_{o_j}, SIM_{o_j}, f^i{}_1, \ldots, f^i{}_{l_i}, f^j{}_1, \ldots, f^{ij}{}_{l_{ij}}\}$

o_r where the class d is unknown. During a retrieval, the most similar cases are searched in the case base. Similarity is defined by local and global similarity measures. The local similarity measures are defined for each type t_i and each concept o_i as a set of possible measures:

$$SIM_{t_i} = \left\{ sim_1^{\ t_i}, \ldots, sim_{m_{t_i}}^{\ \ t_i} \right\} \text{ and}$$

$$SIM_{o_i} = \left\{ sim_1^{\ o_i}, \ldots, sim_{m_{o_i}}^{\ \ o_i} \right\} .$$

Each of these sets contains one specially marked measure $\widehat{sim}^{\ t_i}$ i or $\widehat{sim}^{\ o_i}$ which identifies the measure used for the similarity calculation. The local similarity measures for concepts are combinations of the local similarity measures for the features, e.g. a weighted sum

$$sim_j^{\ o_i}(q, c) = \sum_{a=1}^{l_i} w_a \cdot \widehat{sim}_{e_a}(q_a, c_a)$$

where e a is the type (either complex or atomic) for feature $f_i^{\ a}$ and l_i is the cardinality of features of q and c. The local similarity measures for types are calculated directly. The global similarity measure $sim(q,c)$ between a query q and a case c is defined as $sim(q, c) = sim_{o_r}(q, c)$.

5.3 The Maintenance Operations

After we have defined the knowledge inside a CBR system, we are now able to specify the characteristics of maintenance operations and the operations themselves. The characterisation will help us to determine the kind of changes which result from the operations on the CBR system.

5.3.1 Characteristics of Maintenance Operations

We discovered three major dimension which can be described as follows:

Atomic vs. Composed Operations

The first dimension divides operations into atomic or composed ones. A composed operation can be replaced by a sequence of atomic operations which does the intended changes on the model. An operation is atomic if it can only be replaced by a sequence which contains one or more illegal operations. These are operations which corrupt the integrity of features, concepts, types and cases.

An example for an atomic operation is the deletion of a feature of an concept which can only be handled in one single step, whereas the movement of such a feature from one concept to another is a complex operation. It is composed of two atomic operations, namely delete feature and add feature.

Purpose: If an operation is atomic, the repair is done immediately afterwards. Otherwise, in the complex case, the repair is done sequentially after every atomic operation.

Effects on the Knowledge Containers

Maintenance has an impact on the represented knowledge inside the CBR system. The parts of the system which are affected have to be determined. For a distinction of the different parts we use the knowledge containers vocabulary knowledge, retrieval knowledge, adaptation knowledge and case knowledge. The *vocabulary container* defines the terms for the description of the domain schema *DS* and the case base *CB*, the *retrieval container* includes all knowledge for the retrieval of similar cases, the *adaptation container* is used during the transformation of retrieved solution cases, and the case container holds the cases in the case base.

Which of these containers are touched by a maintenance operation determines the scope of a succeeding repair. Operations can change one or more container(s). If we change the name of a feature, only the vocabulary is changed, the simi– larity measures and the cases remain. On the other hand, adding a feature to a concept concerns the vocabulary, because a new name and relating instances are introduced, as well as the retrieval and the case base, because a similarity measure has to be defined for the new feature and the cases have to be updated with values for the new feature.

Purpose: This dimension determines which containers have to be repaired.

Unambiguousness of Maintenance and Repair Operations

During the execution of maintenance operations and their repair operations several flows are possible. Each flow represents a special way to maintain and repair. This results from the fact that there are various realisations to implement a maintenance and repair operation. Each different kind of such an implementa– tion results in a concrete flow. The different flows are realised by different scripts which combine repair operations. If only one flow exists, we call an operation including its repair *unambiguous*. Otherwise, it is called *ambiguous*. If the system offers different flows the user can choose from a set of scripts to maintain and repair in an intended manner. Beside this kind of user interaction, it can also be necessary to ask the user for certain values in order to update the system.

An unambiguous operation for example, is *rename feature*, whereas *add feature* to a concept is an example for an ambiguous operation. When the feature is inserted into the model, more than one flow is possible. During the update of the case base new values for the new feature have to be assigned to the existing cases. Three different scripts solve this problem: case base update where the user enters a value for each case (CBU_{SV}), case base update where the users enters a default for all cases (CBU_{DV}), and case base update via a given algorithm which defines the values (CBU_{AV}).

As a result, repair operations request user interactions or may work automatically. Consequently, the user has to select one of the repair scripts. If the opera– tion is ambiguous the user has also to interact during the repair of a system. For a detailed description of the possible maintenance operations and repair scripts see the appendix.

Purpose: This characteristic limits the possible repair operations after a maintenance operation and the user's involvement during repair.

Simply spoken, the first dimension defines "when" repair takes place, the second "where" we have to repair and the third the "how-to".

5.3.2 Example Maintenance Operations

Typically, operations are used on several entities of a CBR system. These are the already defined cases, concepts, features of concepts, and types of features. On every entity at least add–, delete– and change–operations are available. In the following, we present two example operations and discuss them according to the dimensions we described in the previous section. The first one is easy to handle because it is atomic, unambiguous and affects only one container. The second, is a composed operation which is ambiguous, affects several containers and is consequently more difficult to manage.

The *change weight of a feature* operation

Description: This operation changes the weight of an existing feature. Name, type and similarity measure of the feature are untouched. Formally, the operation is denoted as:

$$f = (n_f, e_f, w_f) \rightarrow f = (n_f, e_f, \overline{w_f})$$

Atomicity: Changing the weight of a feature is an atomic operation. It cannot be split into further operations and repair takes place after the operation. Container

Affectability: The case base as well as the vocabulary container is untouched because the cases and the vocabulary need not to be updated. The similarity container has changed after the operation and therefore the similarity calculation, too.

Unambiguousness: The result of this operation is unambiguous. No user interaction for repair is required. This operation is easy to handle and requires no further repair operations of the system by a repair script. Such scripts are used to keep a system consistent by a sequence of suggested operations.

The *move concept* operation

Description: The operation moves a concept or a set of concepts from one location to another one in the concept hierarchy. The operation excludes or includes all subconcepts of the moved concept.

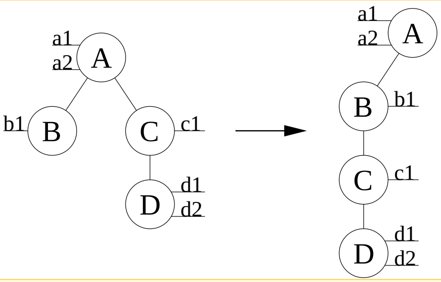

Figure 5-1: move concept Operation

This means that the operation moves only the concept or the concept with all derived subconcepts. The scope of several features may change: features inherited by the previous superconcepts are not visible anymore. Instead, the features of the new superconcepts get visible after the movement. This is illustrated in figure 5-1. As a consequence naming conflicts may occur.

Atomicity: This complex operation can be split into several atomic operations. If only one concept is moved these are: delete all features of the concept, delete concept, add concept and add all features to the new concept. If the whole subtree is moved, this operation sequence has to be perormed recursively on all concepts and subconcepts.

Container Affectability: In general, the movement of a concept affects all containers, the vocabulary, the retrieval and the case base. The vocabulary changes by the changed inheritance of the features. The affectability on the similarity calculation is caused by possible new features which are inherited from superconcepts. Also the case base has to be changed on the basis of schema modification. New values have to be acquired for the new appearing features and values for obsolete features have to be deleted. The structure of the cases itself has been changed.

Unambiguousness: There are two possible user interactions. The first one is the decision if the concept only or the whole subtree should be moved. Second, new values for the new inherited features have to be questioned from the user. This can be realised by system-stored default values, by asking the user for default values, which are used for all cases, or by asking him for each case's value. All these repair operations are realised by different scripts which perform the desired operations. We have identified many maintenance operations and have discussed them like the both above.

5.4 Quality Changes During Maintenance

Maintenance operations and the succeeding repair affect the overall outcome of a CBR system. This holds for analytical as well as for synthetical systems. Because of our restriction to CBR systems for classification tasks, we are able to use either cross–validation or a leave-one-out test to estimate the systems' quality. We know that this restriction is considerable because the quality estimation in other analytical systems like decision support systems is much more difficult. This holds especially for synthetical CBR systems because of their novel resulting solutions.

Because of the unpredictability of the effects of the maintenance operations and their repair to the system in general, it is necessary to offer an evaluation component to the user. This component measures the effects and visualises them so that the user can estimate if the system has changed in the intended manner.

In a first attempt, we use the leave-one-out test to measure the classification results because of its high degree of visualisation combined with its simple usage. As already mentioned, this should be seen as a starting point.

5.4.1 Evaluation Matrix

The class of a query q is predicted by retrieving the k–nearest neighbours $R = \{r_1,\ldots,r_k\}$ of the query and applying a majority vote method. Let $p_{q,\alpha}$ denote the probability that the query is a member of the class $\alpha \in A$. It is defined as:

$$p_{q,\alpha} = \frac{\sum_{r \in \Re} \delta_{r,\alpha} \cdot sim(q,r)^2}{\sum_{r \in \Re} sim(q,r)^2} \quad \text{and}$$

$$\delta_{r,\alpha} = \begin{cases} 1 & \text{if } \alpha_r = \alpha \\ 0 & \text{otherwise} \end{cases}$$

where $\alpha_r \in K$ denotes the class of case r.[1] The prediction of the CBR system is the class with the highest probability calculated from the set of the *k* nearest neighbours. To visualise the outcome of the system, we take a matrix built by a leave-one-out test based on the k–nearest neighbour. This looks as given in Figure 2.

$$\mathrm{LOO} = \left(\begin{array}{c|ccc} c_1 & c_{11} & \cdots & c_{1k} \\ c_2 & c_{21} & \cdots & c_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ c_n & c_{n1} & \cdots & c_{nk} \end{array}\right)$$

Figure 5-2: Quality Matrix LOO

If $|CB| = n$ this test provides a $(k + 1) \times n$ - matrix. Row *i* is formed by excluding case $c_i = \{f_{i_1}, \ldots, f_{i_l}, \alpha_{c_i}\}$ from the case base, retrieving the *k*–nearest neighbours with query c_i and sorting the remaining cases according to their similarity. This will result in a row $r_i = (c_i \mid c_{i1}, \ldots, c_{ik})$. This is done for all cases in the case base and leads to the $(k + 1) \times n$–matrix *LOO*.

5.4.2 Quality Changes

Maintenance operations have effects on the above mentioned matrix with differ– ent impacts. We distinguish five different levels. The goal is to present the user the changes after a maintenance operation for evaluation purposes. The levels themselves are distinguished by the impact of the changes. The five levels are:

1. The operation does not affect the matrix *LOO* at all. All rows and even the similarity values of each entry remain.

1. If the numerator in the definition of $p_{q,\alpha}$ is 0, then $p_{q,\alpha}$ is set to 0.

2. The similarity values of one or more entries of the matrix *LOO* are changed by the operation but the order of the matrix entries still remains the same.
3. Cases permute in one or more rows of the matrix *LOO*. The retrieval provides the same *k*–nearest neighbours as before but in another order.
4. One or more rows of the matrix *LOO* differ from the rows before applying the operation. The retrieval now provides other cases from the case base.
5. The size of the case base has changed which has an effect to the size and contents of the whole matrix *LOO*. This results from adding or removing cases from the case base.

Further, it should be mentioned if the classification of a case has changed or not by the evaluation component. This may happen at all levels, except the first one.

It is not possible to predict the exact level of quality changes for maintenance operations in general. However, we offer an opportunity which presents the im– pact of the maintenance operations to the system. This can be seen as a source of information for the user to evaluate the performed maintenance operations.

However, until now the visualisation of such evaluation results is an un– touched topic in CBR research. Effective visualisation tools are missing, espe– cially for large case bases. Such a toolset would be a challenge for further research and would also improve a lot of other CBR techniques.

5.5 The Overall Architecture

After we have described the different parts of our maintenance component, we present the integration of this component into a CBR system. As already men– tioned, a CBR system consists of the knowledge containers and a modelling tool, which is used for manipulating the containers. A change of the contents of one container may have an effect to the other ones.

The goal of our architecture is to preserve the system's consistency and to enable the user to deal with the above mentioned effects. An overview of the architecture is shown in figure 5-3.

The maintenance component, is placed between the user and the modelling tool. The maintenance component consists of three parts, the maintenance in–terface, the history/undo tool and the quality evaluator. A typical maintenance step goes as follows:

The user sends a *model change request* to the maintenance interface, which includes a *lookup table* with all maintenance operations, possible repair scripts and *inverse operations*. An inverse operation restores the system to the state before the execution of the maintenance operation. The maintenance operation is enacted with the modelling tool and possible repairs are performed. Additionally, the maintenance interface logs the executed model changes in the history/undo tool.

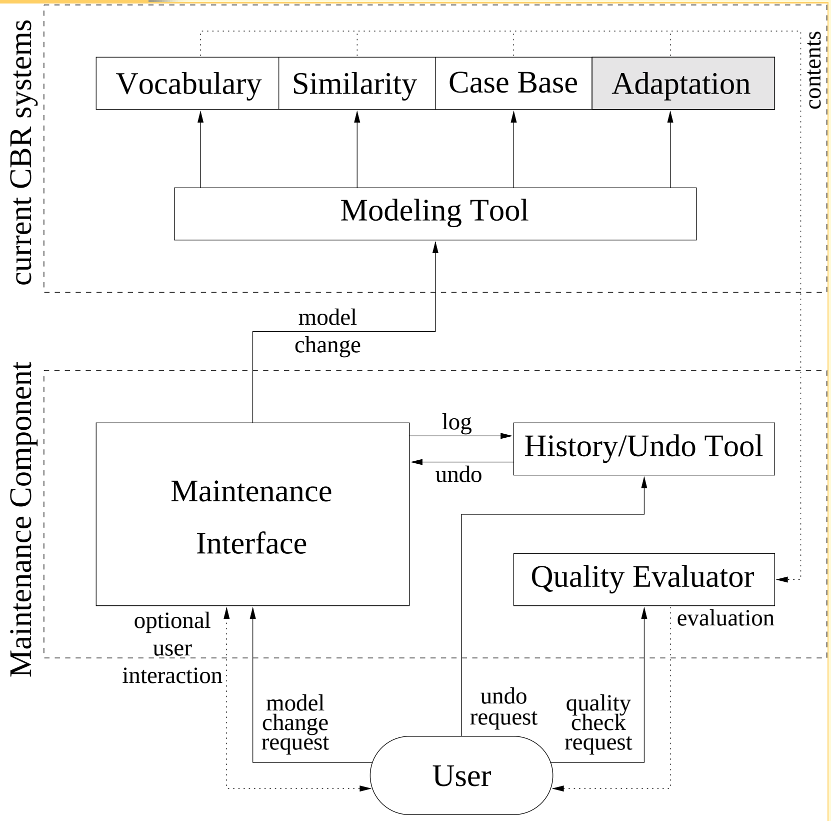

Figure 5-3: Layout of the Architecture

After the maintenance operation, the user has the possibility to analyse the resulting system with the quality evaluator. So, a comparison between the system's quality before and after the maintenance operation is possible. The quality evaluator visualises the differences between the systems according to the differ– ent levels defined in section 5.4.2. If the result of the maintenance operation is unsatisfactory, the user can perform an undo operation. The undo tool looks up the last operation and sends an undo request to the maintenance interface. The interface fetches the inverse operation from the lookup table and executes it. User Maintenance Interface History/Undo Tool Quality Evaluator log undo Maintenance Component Vocabulary Similarity Case Base Adaptation Modeling Tool current CBR systems request undo quality check request evaluation request change model model change contents optional user interaction.

In general, inverse operations exist for all maintenance opera– tions. Some require the storage of nearly the whole system, for ex– ample, if the complete case base has been affected. Due to the stor– age amount, this is often not fea– sible. We found two work–arounds for this problem. The whole sys– tem (or case base) can be dumped to disk before such an operation in order to reload it. Second, the user can be warned, that no undo for this and all former operations is possible. The system can offer both methods and the user can specify the method he prefers in the system preferences.

6 Methodology for Building and Maintaining CBR Applications

This chapter presents a brief overview of the INRECA-II methodology for building and maintaining CBR applications. It is based on the experience factory and the software process modeling approach from software engineering. CBR development and maintenance experience is documented using software process models and stored in a three-layered experience packet.

6.1 Introduction

Today, there are already a few companies which are specialized in developing CBR applications. Their problem is that they mostly develop their applications in an ad-hoc manner: They do not have guidelines or methods which could help their developers in performing a new project and there are no ways to preserve experience made in performed projects for future use. This can cause serious problems when members of the staff leave, taking their experience with them, and new staff has to be trained. The result is an inefficient or ineffective system development, which cannot be sustained by contemporary organizations. From these problems, the need for a *methodology* to support the development and maintenance of CBR applications arson a few years ago and several approaches in that direction have been proposed. A methodology describes the development of a software system using a systematic and disciplined approach. It gives guidelines about the activities that need to be performed in order to successfully develop a certain kind of product, e.g., any kind of software system, as in our case, a CBR application. A methodology shall use a well-defined terminology, which makes it also possible to collect experiences made in past projects in a structured and reusable way to improve future projects. One of the main driving forces behind the development and the use of a methodology relates to the need for quality in both the products and processes of the development of computer-based systems. The use of an appropriate methodology will provide significant quantifiable benefits in terms of *productivity* (e.g. reduce the risk of wasted efforts), *quality* (e.g. inclusion of quality deliverables), and *communication* (a reference for both formal and informal communication between members of the development team and between the developer and the client) and it will provide a solid base for *management decision making* (e.g. planning, resource allocation, and monitoring).

This chapter describes the methodology approach which is based on two relatively new areas in software engineering (SE): *experience factory* and *software process modeling*. We developed a methodology based on recent SE techniques which is enriched by up-to-date experience on building and maintaining CBR applications.

6.2 Methodology Approach

Our approach to a CBR development methodology is itself very ”CBR-like”. In a nutshell, it captures previous experience from CBR development and stores it in a so-called experience packet (a term from the experience factory approach). The entities being stored in the experience packet are software process models, or fragments of it such as processes, products, or methods. The experience packet is organized on three levels of abstraction: a *common generic level* at the top, a *cookbook-level* in the middle, and a *specific project level* at the bottom.

6.2.1 Experience Factory

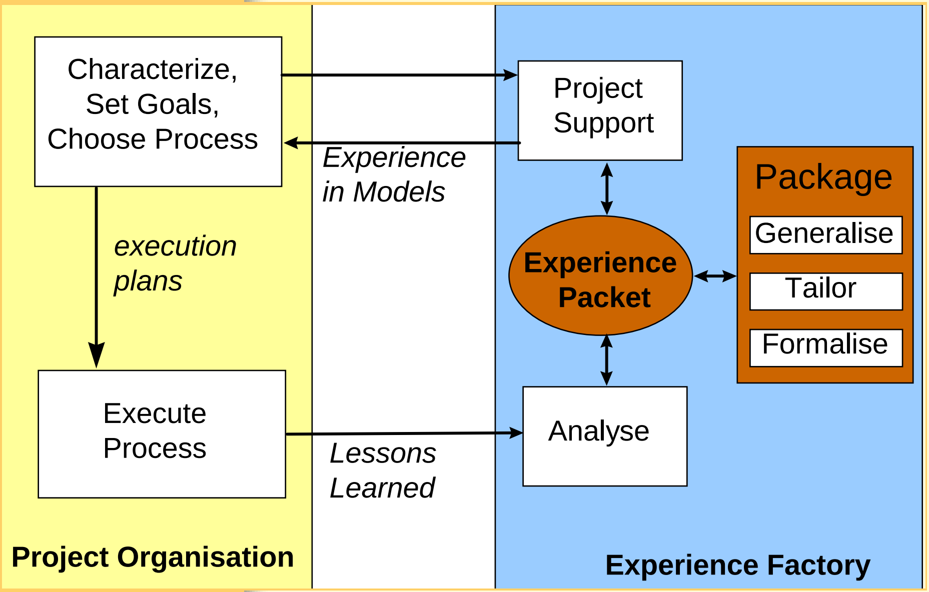

Figure 6-1: The Experience Factory Approach (Basili, Caldiera, & Rombach, 1994)

The experience factory idea is motivated by the observation that any successful business requires a combination of technical and managerial solutions which includes a well-defined set of product needs to satisfy the customer, assist the developer in accomplishing those needs and create competencies for future business; a well-defined set of processes to accomplish what needs to be accomplished, to control development, and to improve overall business; a closed-loop process that supports learning and feedback.

The key technologies for supporting these requirements include: modeling, measurement, the reuse of processes, products and other forms of knowledge relevant to the (software) business. An experience factory is a logical and/or physical organization that supports project developments by analyzing and synthesizing all kinds of experience, acting as a repository for such experience, and supplying that experience to various projects on demand (see figure 6-1). An experience factory packages experience by building informal, formal or schematized models and measures of various software processes, products, and other forms of knowledge via people, documents, and automated support. The main product of an experience factory is an *experience packet*. The content and the structure of an experience packet vary based upon the kind of experience clustered in the packet.

6.2.2 Software Process Models

Software process modeling is an approach that is highly important in the context of the experience factory approach. Software process models describe the engineering of a product, e.g., the software that has to be produced. Unlike early approaches in SE, the software development is not considered to follow a single fixed process model with a closed set of predefined steps. A tailored process model particularly suited for the current project must be developed in advance. Software process models include *technical SE processes* (like requirements engineering, design of the system to be built, coding, etc.), *managerial SE processes* (like management of product related documentation, project management, quality assurance, etc.), and *organizational processes* (covering those parts of the business process in which the software system will be embedded and that need to be changed in order to make best use of the new software system). From time to time, such a model has to be refined or changed during the execution of the project if the real world software development process and the model do not match any longer.

Several representation formalisms for process models have been already developed. Although the particular names that are used vary from one representation to another, all representations have a notation of *processes*, *methods*, *products*, *goals*, and *resources*. A *process* is a single step that has to be carried out in a software development project. Each process has a defined goal and it consumes, produces, or modifies certain products. Usually, the goal of a process is to create or modify the *products*. Products include the executable software system as well as the documentation like design documents or user manuals. For enacting a process, there can be several alternative *methods* that describe how to actually enact the process. When the process is enacted, an appropriate method must be chosen. We distinguish between simple and complex methods. A *simple method* can be a textual description like a guideline of what has to be done to reach the goal of the process. A *complex method* decomposes a process into a set of sub-processes that exchange certain by-products in the course of achieving the goal of the main process.

In this methodology, software process models are used to represent the CBR development experience that is stored in the experience packet. Software processes being represented can be either very abstract, i.e., they can just represent some very coarse development steps such as: domain model definition, similarity measure definition, case acquisition. But they can also be very detailed and specific for a particular project, such as: analyze data from Analog Device Inc. operational amplifier (OpAmp) product database, select relevant OpAmp specification parameters, etc. The software process modeling approach allows to construct such a hierarchically organized set of process models. Abstract processes can be described by complex methods which are themselves a set of more detailed processes. We make use of this property to structure the experience packet.

6.2.3 Structure of the Experience Packet

The experience packet is organized on three levels of abstraction: a *common generic level* at the top, a *cookbook-level* in the middle, and a *specific project level* at the bottom (figure 6-2).

Common Generic Descriptions

At this level, processes, products, and methods are collected that are common for a large spectrum of different CBR applications. These descriptions are the basic building blocks of the methodology. The documented processes usually appear during the development of most CBR applications. The documented methods are very general and widely applicable and give general guidance of how the respective processes can be enacted. At this common level, processes are not necessarily connected to a complete product flow that describes the development of a complete CBR application. They can be isolated entities that can be combined in the context of a particular application or application class.

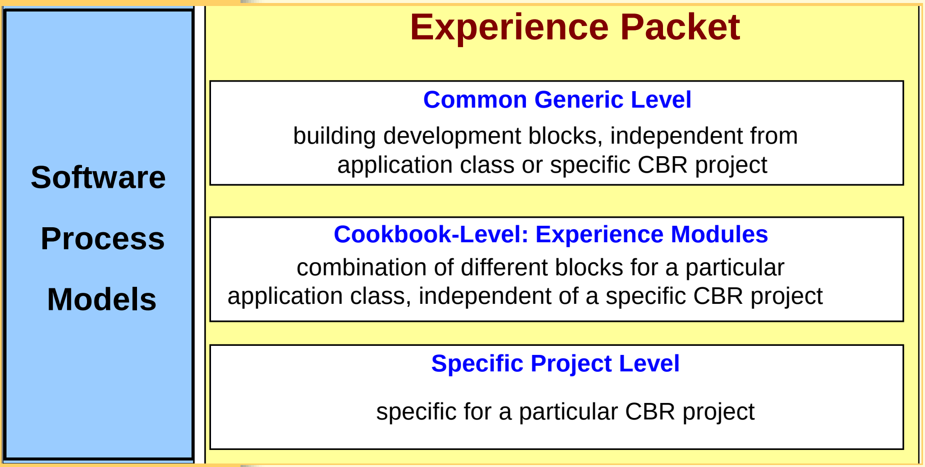
Software Process Models
Experience Packet
Common Generic Level
building development blocks, independent from application class or specific CBR project
Cookbook-Level: Experience Modules
combination of different blocks for a particular application class, independent of a specific CBR project
Specific Project Level
specific for a particular CBR project

Figure 6-2: Structure of the Experience Packet

Cookbook-Level: Experience Modules

At this level, processes, products, and methods are tailored for a particular class of applications (e.g., help desk, technical maintenance, product catalog). For each application class, the cookbook-level contains an *experience module*. Such an experience module is a kind of recipe describing how an application of that kind should be developed and/or maintained. Thereby, the items (processes, methods, and products) contained in such a module provide specific guidance for the development of a CBR application of this application class. Usually, these items are more concrete versions of items described at the common level. Unlike processes at the common level, all processes which are relevant for an application class are connected and build a product flow from which a specific project plan can be developed.

Specific Project Level

The specific project level describes experience in the context of a single particular project that had already been carried out in the past. It contains project specific information such as the particular processes that were carried out, the effort that was spent for these processes, the products (e.g. domain model) that have been produced and methods that have been selected to actually perform the processes and people that had been involved in executing the particular processes.

6.2.4 Documentation of the Experience Packet

Processes, products, methods, agents, and tools being stored in the experience packet are documented using a set of different types of sheets. A sheet is a particular form that is designed to document one of the items. It contains several predefined fields to be filled as well as links to other sheets (see example in the Appendix). We have developed four types of sheets (for products, processes, simple methods, and complex methods) for documenting generic processes that occur on the top and the middle layer of the experience packet and six types of sheets (four sheets for products, processes, simple methods, and complex methods, and two additional sheets for tool and agent descriptions) for documenting specific processes for the specific project level of the experience packet. Figure 6-3 shows the four generic description sheets. One kind of sheet is used to describe generic processes. Generic process sheets contain references to the respective input, output, and modified products of the process. Each product is documented by a separate generic product description sheet. Each process description sheet also contains links to one or several generic methods. A generic method can either be a generic *simple* method (which is elementary and does not contain any references to other description sheets) or it can be a generic *complex* method. Such a generic complex method connects several sub-processes (each of which is again documented as a separate generic process description) which may exchange some by-products (documented as separate generic product descriptions).

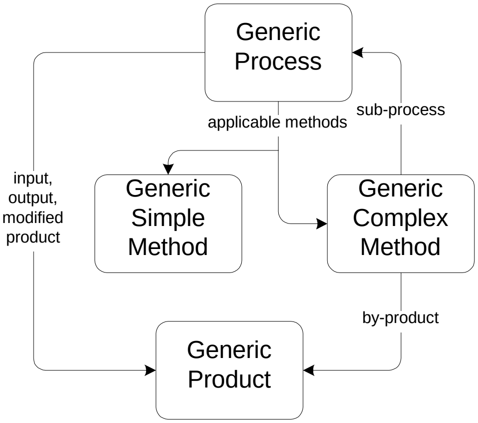

Figure 6-3: Overview of generic description sheets

A particular methodology tool was implemented which supports the management of the experience packet and the different modules it consists of. It supports the filling of the different sheets, checks consistency, and creates the required links. It exports the experience packet as an HTML network in which each sheet becomes a separate HTML page that includes links to the related pages. Therefore, it is possible to investigate the experience packet via Intranet/Internet using a standard Web browser.

6.2.5 Using and Maintaining the Experience Packet

When a new CBR project is being planned, the relevant experience from the experience packet must be selected and reused. The experience modules of the cookbook-level are particularly useful for building a new application that directly falls into one of the covered application classes. We consider the experience modules to be the most valuable knowledge of the methodology. Therefore, we suggest to start the “retrieval”[1] by investigating the cookbook-level and only using the common generic level as fall-back. Furthermore, it is important to maintain the experience packet, i.e., to make sure that new experience is entered if required. For using and maintaining the experience packet we propose the following procedure:

1. Identify whether the new application to be realized falls into an application class that is covered by an experience module of the cookbook. If this is the case then goto step 2a; else goto step 3.
2. a) Analyze the generic processes, products and methods that are proposed for this application class.
   b) Select the most similar particular application from the specific project level related to this module and analyze the specific description sheets in the context of the current application.

1. Up to now, this retrieval is not supported by a tool, but through an index schema. However, support for retrieval (e.g. a CBR approach) is considered important for the future.

c) Develop a new project plan and workflow for the new application based on the information selected in steps 2a and 2b. Goto step 4.

3. Develop a new project plan and workflow for the new application by selecting and combining some of the generic processes, products and methods from the common generic level; make these descriptions more concrete and modify them if necessary.
4. Execute the project by enacting the project plan. Record the experience during the enactment of this project.
5. Decide whether the new project contains new valuable information that should be stored in the experience packet. If this is the case, goto step 6, else stop.
6. Document the project using the specific description sheets and enter them into the specific project level of the experience packet (supported by the methodology tool).
7. If possible, create a new experience module by generalizing the particular application (together with other similar applications) to an application class and generalize the specific descriptions into generic descriptions. Add the new to the current cookbook (supported by the methodology tool).
8. If new generic processes, methods, or products could be identified that are of a more general interest, i.e., relevant for more than the application class identified in step 7, then add them to the common generic level (supported by the methodology tool).

Bibliography

[Aam91] Aamodt, A.: A knowledge-intensive approach to problem solving and sustained learning. Ph.D. Dissertation, University of Trondheim, Norwegian Institute of Technology (1991)

[AaP94] Aamodt, A., Plaza, E.: Case-Based Reasoning: Foundational Issues, Methodological Variations and System Approaches, AICom - Artificial Intelligence Communications, IOS Press, Vol. 7: 1 (March 1994) 39-59

[AlW97] Althoff, K.-D., Wilke, W.: Potential uses of case-based reasoning in the experience-based construction of software systems. In: R. Bergmann & W. Wilke (eds.), *Proceedings of the 5th German Workshop in Case-Based Reasoning (GWCBR'97)*, LSA-97-01E, Centre for Learning Systems and Applications (LSA), University of Kaiserslautern (1997)

[BBF98] Bergmann, R., Breen, S., Fayol, E., Göker, M., Manago, M., Schumacher, J., Schmitt, S., Stahl, A., Wess, S. & Wilke, W.: Collecting experience on the systematic development of CBR applications using the INRECA-II Methodology (1998)

[BBG97a] Bergmann, R., Breen, S., Göker, M., Johnston, R., Schumacher, J., Stahl, A., Traphöner, R., Wilke, W.: Initial methodology for building and maintaining a CBR application. INRECA-Deliverable (1997)

[BBG97b] Bergmann, R., Breen, S., Göker, M., Johnston, R., Schumacher, J., Traphöner, R., Wilke, W.: Cookbook for building and maintaining a CBR application. INRECA-Deliverable (1997)

[BCR94] Basili, Caldiera, Rombach: The Experience Factory. In J. Marciniak (Ed.) *Encyclopedia of Software Engineering - Vol 1*. New York: Wiley (1994)

[BeA98] Bergmann, R. & Althoff, K.-D.: Methodology for building CBR applications. Chapter 12 of Lenz, Bartsch-Spörl, Burkhard, Wess (Eds). *Case-Based Reasoning Technology*. LNAI 1400, Springer (1998)

[BeE95] Bergmann, R. & Eisenecker, U.: Fallbasiertes Schließen zur Unterstützung der Wiederverwendung objektorientierter Software: Eine Fallstudie. Proceedings der 3. Deutschen Expertensystemtagung, XPS-95, pp. 152-169, Infix-Verlag (1995)

[Ber98] Bergmann, R.: On the use of taxonomies for representing case features and local similarity measures. In Gierl & Lenz (Eds.) *6th German Workshop on CBR* (1998)

[BeS98] Bergmann, R., Stahl, A.: Similarity Measures for Object-Oriented Case Representations, Proceedings of the European Workshop on Case-Based Reasoning, EWCBR'98

[Boo91] Booch, G.: Object–Oriented Design with Applications. Benjamin/Cummings (1991)

[BPW94] Bergmann, R. Pews. G., Wilke, W.: Explanation-based similarity: A unifying approach for integrating domain knowledge into case-based reasoning for diagnosis and planning tasks. In Wess, S., Althoff, K.-D., and Richter M.M. (Eds.) *Topics in Case-Based Reasoning*, Lecture Notes in AI, pp. 182-197 (1994)

[BuM94] Bunke, H. & Messmer, B.: Similarity measures for structured representations. In Wess, Althoff & Richter (Eds.) *Topics in Case-Based Reasoning*, pp. 106-118, LNAI 837, Springer (1994)

[BWA97] Bergmann, R., Wilke, W., Althoff, K.-D., Breen, S., Johnston, R.: Ingredients for Developing a Case-Based Reasoning Methodology. In: R. Bergmann & W. Wilke (eds.), *Proceedings of the 5th German Workshop in Case-Based Reasoning (GWCBR'97)*, LSA-97-01E, University of Kaiserslautern, pp. 49-58.

[BWS97] Bergmann, R., Wilke, W., Schumacher, J.: Using software process modeling for building a case-based reasoning methodology: Basic approach and case study. In: D. Leake & E. Plaza (eds.) Case-Based Reasoning Research and Development (ICCBR'97). Lecture Notes in AI. Springer, pp. 509-518.

[BWT94] Bergmann, R., Wess, S., Traphöner, R., Breen, S.: Using Background Knowledge in the Integrated System: Specification and Approach, ESPRIT project 6322, Deliverable, Kaiserslautern (1994)

[BWW94] Bergmann, R., Wess, S., Wilke, W.: Using Rules to Represent Background Knowledge for CBR, Deliverable of the INRECA Esprit Project (1994)

[For82] Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, **19**, pp. 17-37 (1982)

[FBF77] Friedman, J.H., Bentley, J.L., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. *ACM Trans. math. Software*, 3, pp. 209-226 (1977)

[GRB98] Göker, M., Roth-Berghofer, T. Bergmann, R., Pantleon, T., Traphöner, R., Wess, S., & Wilke, W.: The development of HOMER: A case-based CAD/CAM help-desk support tool (1998)

[HeW98] Heister, F. & Wilke, W.: An Architecture for Maintaining Case-Based Reasoning Systems. Proceedings of the European Workshop on Case-Based Reasoning, EWCBR'98 (1998)

[Kol93] Kolodner, J.: Case-Based Reasoning, Morgan Kaufmann Publishers, San Mateo (1993)

[Lal87] Lalonde, W.R.: A novel rule base facility for Smalltalk. *Proceedings of the ECOOP*'87, pp. 193-198 (1987)

[LBB98] Lenz, M., Bartsch-Spörl, B., Burkhard, H.-D., Wess, S. (eds.): Case-Based Reasoning Technology, From Foundations to Applications, Springer-Verlag, Berlin/Heidelberg (1998)

[MaB94] Manago, M. Bergmann, R. et al.: CASUEL: A common case representation language. *Deliverable of the INRECA Esprit Project* (1994)

[OsB96] Osborne H., Bridge, D.: A case base similarity framework. In Smith & Faltings (Eds.) *Advances in Case-Based Reasoning, pp. 309-325, LNAI 1168,.* Springer (1996)

[Pla95] Plaza, E.: Cases as terms: A feature term approach to the structured representation of cases. In Veloso & Aamodt (Eds.) *Case-Based Reasoning Research and Development*, pp. 265-276, LNAI 1010, Springer (1995)

[RoV95] Rombach, Verlage: Directions in Software Process Research. *Advances in Computers*, Vol. 41, Academic Press (1995)

[Ric95] Richter, M. M.: The knowledge contained in similarity measures. Invited Talk on the ICCBR–95. http://wwwagr.informatik.uni–kl.de/~lsa/CBR/Richtericcbr95remarks.html (1995)

[Sul99] Schulz S.: CBR-Works - A State-of-the-Art Shell for Case-Based Application Building, Proceedings of the German Workshop on Case-Based Reasoning, GWCBR'99 (1999)

[SKH97] Sanders, K., Kettler, B., Hendler, J.: The case for graph-structured representations. In Leake & Plaza (Eds.) *Case-Based Reasoning Research and Development*, LNAI 1266, Springer (1997)

[VBH93] Voß, A., Batsch-Spörl, B., Hovestadt, L., Jantke, K.P., Peterson, U., Strube, G.: FABEL: Projektstatus, Perspektiven und Potentiale, Fabel Report No. 16, Gesellschaft für Mathematik und Datenverarbeitung mbH, Sankt Augustin, Germany (1993)

[WAD94] Wess, S., Althoff, K.-D., Derwand, G.: Using k-d trees to improve the retrieval step in case-based reasoning. In Wess, S., Althoff, K.-D., and Richter M.M. (Eds.) *Topics in Case-Based Reasoning*, Lecture Notes in AI, Springer-Verlag, pp. 167-181 (1994)

[WeK91] Weiss, S. M., Kulikowski, C. A.: Computer Systems That Learn -- Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems. Morgan Kaufmann (1991)

[Wes95] Wess, S.: Fallbasiertes Problemlösen in wissensbasierten Systemen zur Entscheidungsunterstützung und Diagnostik, Ph.D. Dissertation, University of Kaiserslautern (1995)

[Wil98] Wilke, W.: Knowledge Management for Intelligent Sales Support in Electronic Commerce, Ph.D. Dissertation, University of Kaiserslautern (1998)

[TEC99a] CBR-Works 3 - Reference Manual, TecInno GmbH, Kaiserslautern (1999)

[TEC99b] Introduction to the Case-Query-Language, TecInno GmbH, Kaiserslautern (1998)

[Tve77] Tversky, A.: Features of Similarity. *Psychological Review*, 84, pp. 327-352 (1977)