Experiences from Application of a Faceted Classification Scheme

Lars Sivert Sørungård; Guttorm Sindre and Frode Stokke.

Division of Computer Systems and Telematics
Norwegian Institute of Technology (NTH)

Abstract

In this article we discuss various aspects of the classification and retrieval mechanisms which have been applied in the ESPRIT-2 project REBOOT. This classification scheme is now being slightly modified due to requirements identified through practical application.

This article first discusses different ways of classification and retrieval in general, then the classification scheme originally proposed by REBOOT is examined. With this as the background, we continue by discussing the experiences we have gained throughout the development and application of this scheme. Then, a proposal for a refined version of the scheme is presented in order to help out some of the shortcomings of the original scheme. Finally we draw some conclusions from our work, and point out problems which are of primary concern for our forthcoming research.

Keywords: software reuse, faceted classification, retrieval, relaxation, structured term spaces

1 Introduction

Different approaches to reuse have been proposed, many of them have considered pieces of source code as the only interesting reusable “artifact” from the software development process. The REBOOT process is promoting reuse of practically all output from the software life-cycle, thus a reusable entity in the REBOOT context is called a component ([5]) in order to emphasize its general nature.

Supporting reuse of various kinds of components implicitly calls for REBOOT to apply reuse by composition which simply means that the reuse builds new systems by composing it from the more or less atomic building blocks ([4]) provides a survey of different approaches to reuse). Another implication of the diversity of the components is that a large number must and will be present. We are talking about components from many domains, in many languages and design notations. Due to this large amount of components, we think that a component management system is needed in order to keep track of the properties of all the components which are available.

A crucial part of a reuse tool is the mechanisms of insertion and retrieval. Through insertion, information is attached, manually or automatically, to the component. The retrieval mechanism is based on this information, thus its relevance is of utmost importance in order to support “intelligent” retrieval.

REBOOT applies a classification scheme in order to support categorization, search and retrieval. The rest of this article will concentrate on the classification scheme we originally proposed and the application experiences we have. A more detailed description of the classification scheme we use is provided in [3, 2]. Based on these experiences, we will point out how the shortcomings and weaknesses of the scheme may be corrected.

The rest of this article is organized as follows:

Section 2 will provide a brief introduction to classification, in particular faceted classification will be examined. Then the classification scheme we originally developed and used will be described.

Section 3 is a summary of the experiences we have gained from applying our classification scheme on various kinds of components.
Section 4 proposes some changes to the existing scheme in order to improve the expressibility according to our experience.

Section 5 will draw some conclusions from our experiences, and also explain what problems that need further elaboration.

2 Classification

As we pointed out in the introduction, our scenario is a development process with component based reuse and a large repository of reusable components available throughout the entire software development process. If the number of components is small, less than a hundred, we will probably quickly get so familiar with them that we instantly know which of them is most suitable for reuse in a given situation. But when the number of components is very large, more than one thousand, our brain is not capable of handling all the necessary information simultaneously. Thus we need a component management system for keeping track of all the relevant information about the components. Such a system will be able to store a lot of detailed information, and it will be able to compare information in order to select the most suitable component.

Another aspect of reuse is that some modifications often are necessary in order to make the reused component work in its new context. Of course, the benefit of reuse is biggest when the required modification of a component is small. However, it definitely is interesting to do reuse even if some modifications are necessary. Thus, a reuse support system should provide a mechanism which is capable of retrieving the component which requires the smallest modifications on order to be reused.

As we mentioned in the introduction, it must be easy to search for and retrieve components, otherwise reuse will of course not happen. And further, the retrieval mechanism is dependent on the information which is available. Different kinds of simple existing search mechanisms are e.g.:

**Associative search** Retrieval of objects in relational databases are based on associative search on the value of attributes. This search requires a perfect match between the search word and the attribute. One adaptation of associative search is the use of keywords to characterize the objects.

**Textual search** By textual search we can retrieve objects where certain text patterns appear. These patterns are usually restricted to regular expressions, and textual search is mainly restricted to free text. One interesting variant of this approach is to analyze sentences in free text for (noun verb) pairs, and distinguishing pairs are associated with each component [1]. Queries are natural language sentences where a similar analysis is made, and matching is performed. The problem with this approach is the necessity of an available free-text description of each reusable component.

We think that these two approaches in their pure forms are too limited for application in a reuse support system. Associative search is too restrictive since we need to know the possible keywords to search for, and we will only get the components with exactly these words as attributes. Textual search, on the other hand, has limited capabilities of expressing different categories of information, i.e. data with different interpretation.

Requirements to a classification scheme

Since search and retrieval are based on the available information about the components, a natural step is to ensure that the necessary information is provided, and that the provided information is relevant. Information may be attached to a component manually, as user input, or automatically, i.e. extracted from already present information, e.g. documentation. With respect to reuse, the interesting information is primarily about the functionality of the component. Due to the diversity, in quality as well as in quantity, of the documentation which is usually available for a component, we feel that manual classification is the most flexible solution currently. A more sophisticated automatic solution of this process may perfectly well be applied later.

If we consider the two search strategies previously mentioned, we see that the keyword approach to associative search very much resembles classification. Also, in the textual search case, classification will help extracting information which is known to be relevant and which describes the component. However, the problem with textual search will still be that it is hard to express different categories of information. Thus, the classification scheme must be able to structure the provided information in a way that permits expression of different aspects of information. Based on this, we think some common requirements to a classification scheme are:

1. The classification information must express relevant information about the component (from the
Faceted vs. enumerated classification

Different sorts of classification schemes have been used for different purposes, each of them with their strong and weak sides. In addition to the traditional schemes, where keywords are entered to describe the component, there are other approaches as e.g. [13] where different kinds of components are characterized by component descriptor frames. In this article, we concentrate on the keyword-based approach.

Traditional keyword-based classification schemes can be mainly be divided into two categories:

1. Enumerated schemes
2. Faceted schemes

The main difference between these two categories is that enumerated schemes\(^2\) is a breakdown of knowledge in only one dimension (e.g. the subject of a book), thus it provides a way of linearizing components classified with such a scheme. This is an important property of classification schemes made for e.g. literature, where books on the same subject are supposed to be closely located in the library.

Faceted classification was proposed by S. R. Ranganathan, see e.g. [9]. The main difference from enumerated schemes is that faceted schemes do a breakdown of information in different categories, this makes it possible to consider information about several different aspects or properties of the components. Such an aspect is denoted a facet.

Rubén Prieto-Díaz [7, 8] has made a faceted classification scheme for reuse of traditional function oriented software. An important part of his work is the association of a structured set of legal terms, a termspace, with each facet. Terms from this termspace are used for classification and searching. The structure of the termspace reflects the semantical relations between the terms, so the termspace is a kind of semantic network. The relation between the termspace, the facets and different components is shown in figure 1. Here, each component is described with an arbitrary number of terms for each facet. The terms are chosen from the termspace, which is here structured as a hierarchy with increasing specialization downwards.

In our opinion, a faceted scheme is a favorable choice for classification of reusable software components. The main reason for this opinion is that the question of whether to reuse or not relies on information from several different categories. This implies that we need a new requirement to the classification scheme: the information categories (facets) should be well defined and as orthogonal to the other facets as possible.

In this context, it is important to note that faceted classification is not restricted to have facets only, ordinary attributes may perfectly well be applied too. The difference between a facet and an attribute is that:

- Facets represent the information most important with respect to reuse.
- Each facet has an associated structured term space.

Since we do not consider attributes to be as important as facets, and also more straightforward to use, we will limit our following discussion to facets.

2.1 The original four facets of REBOOT

The critical question when developing a faceted classification scheme is of course what facets to choose.

---

\(^2\)An example of this is the Universal Decimal Classification, consider [16] for details.
Our intention has been to develop a classification scheme applicable for components from several phases in the development life cycle. Another important concern of REBOOT is that the classification scheme shall be made particularly for object oriented components. Thus, we have a point of view slightly different from Prieto-Díaz, because the scheme developed by him was intended used for functionally oriented source code components.

Figure 2: The original four REBOOT facets

We have chosen a classification scheme built around four facets, which are depicted in figure 2.

**Abstraction:** Usually, an object-oriented component can be characterized by a noun, e.g. stack, resource manager etc.

**Operations:** Components have operations, and these are characterized in the Operation facet, e.g. push and pop.

**Operates On:** This facet describes what kind of objects this component is able to act on, e.g. integer, set, list, resource.

**Dependencies:** This facet lists dependencies to such as compilers, operating systems etc. which must be present for the component to work, e.g. AT&T-C++, UNIX, etc.

We thought that these four facets would be applicable to most reusable components, we also considered the expressibility to be appropriate. We tried to reduce the complexity so only the absolutely most interesting information would be available through the faceted scheme. Note however that an arbitrary selection of ordinary attributes could be available in addition to the facets, as we mentioned earlier.

**Terms, relations and relaxed search**

As discussed above, the different information categories of interest are denoted *facets*. The *values* for each facet are denoted *terms*. When the reuser is going to do a search for a component, he must fill in terms for each facet according to the idea he has of what component he wants to reuse. Much of the same is to be done when a programmer is classifying a reusable component; he must fill in proper terms for each facet. In fact, as we show in figure 3, the processes of retrieval and classification are rather similar. This similarity is partly a side effect of faceted classification in which the component is viewed "through" its facets. Thus, whether we consider a real component or merely an idea of a component does not matter as long as it is viewed from the outside.

If we reconsider the four facets with classification of object oriented components in mind, we see that the terms for the *abstraction* facet often resemble the *class name*, i.e. the abstract concept which is implemented by the component. The *operations* facet will denote what *methods* the component offers, while the *operates on* facet describes what other components that cooperate with the classified one (i.e. the classified component sends messages to components specified in the *operates on* facet). Ideally, the terms for the *operates on* facet should be the same as the *abstraction* terms (with the addition of simple data types).

Figure 3: The retrieval and classification processes

The possible terms for one facet are kept in a particular data structure called a *term space*. An example of a term space is shown in figure 4. The term space is a kind of graph where the vertices are the terms and the edges are particular *relations*. These relations reflect the relationship between related terms, and may be of different types. Currently we apply generalization, specialization and synonym. These relations are present for two purposes:

1. Simplify navigation
2. Relax the search terms

The purpose of support for navigation should be obvious. When searching (or classifying), the user can only use "legal" terms, i.e. terms present in the term space. In order to find such terms, the user usually must navigate in the term space. Thus, following relations should lead quickly to the desired term (possibly combined with some ordinary kind of search such as "common substrings" etc.).

The other reason for the use of relations is the requirement of retrieving the component which requires least changes. Thus, if not a component with classification terms exactly matching the specified search terms is found, then the components with classification terms close to the search terms are inspected. The closeness we mention here is the closeness between the terms in the term space. In order to express such closeness, the relations should have a weight reflecting similarity. For a further discussion of weights, consider [2]. This mechanism of finding similar components is called relaxation of the search. A similar approach to this problem is proposed by [6].

3 Experiences from software classification

The purpose of this section is to describe the problems we encountered when classifying object oriented software. The experience we have is limited to classification of two object-oriented class libraries (NIH and COOL) implemented in C++. In addition, we have been classifying a complete special application. In the following, we will try to avoid mentioning particular C++ mechanisms which have caused trouble and rather keep the discussion at a general and language-independent level.

Our experience, even if somewhat limited, covers three different aspects of classification:

Restructuring the term space and classification.

Classification is an iterative process, thus there must be some kind of semantic "synchronization" between the term space and the classification. As the classification goes by, new terms are added to the term space. This may require other components being reclassified according to the new terms. Also, the term space may need some restructuring.

Classification of another class library. In this case we classified another class library (COOL) with the termspace developed for NIH. This is a good test of whether the term space is generally applicable, or if it is too much tied to the particular set of components for which it was first developed.

Classifying a special application. This is important in order to explore the extendibility of the term space. It is also important to consider particular problems which occur when classifying specialized software. Probably this is an example of the most typical classification situation.

We will try to emphasize and discuss problems with the original REBOOT classification scheme with respect to expressibility of the facets, ease of use and maintenance of the termspace. Thus we will discuss each of the four facets in turn, and finally discuss the problems of maintaining the termspace.

3.1 Abstraction

The main problem with this facet is simply to determine the right abstraction. In particular, this is a bit different for a specialized application compared to a class library. E.g. specialized data structure classes can be classified with terms reflecting both the pure data structure from an ADT point of view, as well as terms reflecting the specialized behavior from the application domain point of view. An example of this is the class TermSpaceTerm which may be considered a vertex of a general graph as well as a class representing the behavior and properties of a term. Classifying pure data-structure classes, such as those usually present in most common class hierarchies, is generally easy since these classes tend to be well defined as an ADT.

3.2 Operations

This facet has some similarities to the abstraction facet, selecting an Operations term is the same as finding the abstraction of a particular method. However, methods are more fine-grained than classes, this makes the selection easier. But still there are a few problems with different kinds of methods:

Virtual functions may cause some difficulties with respect to classification. The purely virtual functions can only be classified according to what the classifier thinks they are supposed to do. Inspection of implementations in subclasses may help this. If different subclasses implements the function in different ways requiring different terms;
the purely virtual method should be classified with a term high up in the term hierarchy, i.e. a general term.

**Methods with non-atomic functionality** can be classified in two ways:

1. With several terms describing each atomic functionality
2. With one term describing the entire operation

The problem with the first approach is that it gives the user the impression that each atomic function is available from the component. Another problem is: **What is atomic functionality?** We may break functionality down into machine code, or even further, but classifying at that level of detail will make the classification more complex than the component. We have applied a slightly modified version of the second approach where we classify a function according to what operations it may perform from an external point of view.

### 3.3 Operates On

Problems with the **Operates On** facet are particularly likely to occur in specialized and complex components with many collaborators. One problem is to limit the number of terms in the facet; there is not really much point in classifying absolutely all parameters, data members, local variables etc. as being operated on. We think that only the most important ones should be classified. The limit for this importance is hard to define, it is up to the classifier to judge this.

But yet there is one problem left, namely how to specify the role of the component being operated on. Consider e.g. a dictionary defining a mapping from strings to real numbers, e.g. “one quarter = 0.25”. This dictionary would be classified as **Operates On** strings and reals. But there is no way to tell, from the classification, whether strings or reals are the keys or the values of this dictionary.

3.4 **Dependencies**

This facet is probably the most straightforward to use, we have not really encountered any particular problems when applying it. However it is important to remember that we want to express dependencies to items outside the software system; internal dependencies are captured by the **Operates On** facet.

### 3.5 General problems

Classifying macros and non-member functions is necessary e.g. in the COOL case, since this library contains several such constructions. This can be done in two ways:

1. If the function can be considered a method of a class, then a term describing it may be included in the **operations** facet for that class.
2. If the function is not related to any particular class, it can be classified as a single function where
the *abstraction* facet is not taken into consideration, or perhaps a term like *function* can be used as abstraction. Another possible way is to classify a *group* of related functions as a kind of *module* with one abstraction and several operations.

Another problem of general nature is how to classify parameterized classes; this is a bit different from classifying "ordinary" classes. The *abstraction, operations* and *dependencies* facets are used as before, while the term *user defined type* is used for the *operates on* facet. One problem here is that it is difficult to express what *operations* the parameterized class requires from the particular user defined type. This is related to the problem of specifying roles as mentioned above.

Classifying public data members should ideally be a non-existing problem since strict encapsulation should be applied to object oriented software. However, public data members do exist rather often, by the way this is also a question of language mechanisms. The two possible ways of classifying such data members with the current facets are:

1. Classify only the methods which are explicitly available, i.e. the methods offered by and implemented by the containing class.

2. Classify both explicitly available and *indirectly* available methods.

The problem with the first solution is that the methods indirectly available do not seem to exist at all when only the classification is inspected. The second approach is also insufficient since all methods appear to be offered by the containing class. This approach could be applied in cases where the containing class could have inherited its public data member instead of containing it. But in most cases, we have applied the first solution.

We feel that the perhaps most important weakness of this classification scheme in general is the inability to express different aspects or variants of the terms explicitly. Currently, variants must be represented by adding a new term completely representing the new variant. A better solution would be to use *qualifying terms* representing different variants of a main term. An example is a set of terms representing different variants of insertion: *insert first, insert last, insert by index* etc. This could rather be modeled as a set of *qualifying terms: first, last, by index* etc. and one main term *insert*. In this way, the variants are constructed by combining the different qualifying term with the main term. This will be discussed in section 4.

3.6 Termspace maintenance

The process of maintaining a term space could be improved by tool support providing guidance and help for semi-automatic updating. Several aspects of this is discussed in [11].

One of the most important rules for maintenance of a term space is to *keep the classification consistent*. One aspect of consistence is to use the same terms for the same concepts even if the implementation of the concepts is different. This is particularly important when classifying several similar components. In order to avoid a term space with terms directly reflecting e.g. method names, one should try to use existing terms as far as possible. But, of course, sometimes a new term must be added. In such cases it is important to add a term describing the general concept, not the particular implementation. Thus, usually one should avoid classifying e.g. the methods with the methods names as terms directly.

Consistence can also be interpreted as keeping the *structure of the term space* consistent. I.e., the structure should reflect the terms and the relations between them in a logical way. A consistent structure is important for several purposes:

- It will be easier to decide where new terms should be inserted.
- The grouping of terms will simplify the interpretation of ambiguous terms by inspection of the surrounding ones.
- The structure will decide how the *relaxation* of the search will work.
- Navigation in the term space will be easier.
- Incremental changes to the termspace will not confuse users who have already become familiar with the existing termspace.

Insertion of new terms is likely to occur rather often when maintaining a term space. Sometimes it may be necessary to remove terms too, this task is a bit more complicated. The problem is how to handle the components which are classified with the term which is about to be removed. The most open and flexible solution would be to have a tool which semi-automatically replaced the removed term with another term close (in the term space) to the removed one.
3.7 Granularity of the termspace

If the term space is to coarse-grained it will be difficult to find accurate terms for classification, and a large number of components are likely to be classified with the term. A term so general that it can be applied to almost any component, is clearly not very useful. This will only cause a large number of uninteresting components to be found during a search. Still a small term space is easier to get familiar with, and hence easier to maintain and use.

The domain of the term space also affects the granularity. A term space with a narrow domain should be more fine-grained than a term space containing a large variety of components.

Generally we think that it is better to have a too fine-grained term space than a too coarse-grained one. When using a fine-grained term space in retrieval it is not necessary to know the exact terms used for classification. If one supplies a general term, the relaxed search will still find the components. If the number of retrieved components is too large, one can just use more specialized terms in the search, or set some more restrictive limits for the expansion.

4 Possible improvements of the classification scheme

The three most important shortcomings of the current scheme are in our opinion:

1. The inability to express roles of collaborators
2. Representing variants by using qualifying terms
3. The problem of specifying data members

In fact, both the two first problems will be eliminated if the opportunity of applying qualifying terms is present. In this way, different roles or protocols may be specified to explicitly denote the required operational interface of collaborators. This also calls for a change of the name of the facet, Operates On, should be changed to Collaborators to emphasize the commitment to object-oriented techniques and terminology.

The qualifying terms should be structured in a termspace in the same way as the main terms. Thus, each facet will have two different termspaces: One for the qualifying terms and one for the main terms. Thus the Collaborators facet will have qualifying terms representing different protocols. Another useful feature here would be to have a special kind of relation from these protocol-terms into the main-termspace of the Operations facet to explicitly show what methods are available through the various protocols. This approach will also support classification of frameworks, which we currently have no experience with.

The problem of specifying data members can not be easily solved without adding a new facet representing exactly this property. An appropriate name of this facet would in our opinion be Parts. This facet will syntactically be similar to Collaborators with respect to qualifying- and main-terms. The difference is that Parts are internal to the component while Collaborators are external.

Thus, we are left with five facets with corresponding termspaces:

Abstraction
- **Main terms** are used for denoting what real-world concept the component represents (as in the original proposal).
- **Qualifying terms** describe various functional as well as non-functional properties of the component such as e.g. life-cycle phase etc.

Operations
- **Main terms** are chosen to represent the overall functionality of the method (as in the original proposal).
- **Qualifying terms** reflect aspects of each method, e.g. with respect to structure manipulation (first, last etc.), kinds of algorithms applied (quick-sort, bubble-sort) etc.

Parts
- **Main terms** are the same as for Abstraction.
- **Qualifying terms** are protocol names representing the set of methods which the part must be able to respond to. These terms have relations into the Operations termspace to reflect what methods are actually required to satisfy each protocol.

Collaborators
- **Main terms** are the same as for Abstraction.
- **Qualifying terms** are the same as for Parts.
5 Conclusion

In addition to the concrete problems we have discussed, one of the most important experiences from the classification work is that the amount of work required to classify a component system strongly varies, but is in general time consuming. Good documentation of the software components is extremely important for the classifier.

In general, we feel quite satisfied with the basic mechanisms of the classification scheme. We think that the ideal classification scheme is almost impossible to find, there will always be a trade-off between expressibility and complexity. Anyway, the treatment of term spaces and the mechanisms of search and relaxation seem to be convenient. However, we must emphasize that our experience so far is mainly in the area of classification.

The possible improvements to the classification scheme need to be examined more carefully, this will be the focus for our further research. In particular, we must determine in detail how the qualifying terms should be applied. This implies a lot of new work on how the relaxation of the search should be performed. Also, the treatment of weights under the influence of qualifying terms will need some changes.

In fact, much more classification work must done before a classification scheme can be proven effective. The classifications will have to cover a large number of domains, and the number of components should be large. Evaluation of the scheme should not only involve classification, but also search and retrieval. Otherwise, the quality of the classification scheme will be impossible to judge.

References


