Matti Äijänen

Component Search in a MetaCASE Environment

Master's thesis

19.11.2001

University of Jyväskylä
Department of Computer Science and Information Systems
Jyväskylä

Abstract

Äijänen, Matti Ville
Component Search in MetaCASE Environment/ Matti Äijänen
Jyväskylä: University of Jyväskylä, 2001.
107 p.
Master's thesis

The full utilization of the potential of software components requires that the components be stored into a library. When these libraries grow large enough, finding an appropriate component merely by browsing through the library becomes too laborious. Therefore, some kind of retrieval tool is needed.

The goal of this study is to evaluate different information retrieval possibilities and to find the ones most applicable to component search. The study is largely based on existing literature on different retrieval possibilities. The focus is further narrowed to CASE tools and particularly metaCASE tools. Studies in this area are relatively few although the concept is important in modern software development.

We will propose a practical example of a retrieval tool based on our literature review and provide advice on how it should be implemented. This retrieval tool is designed primarily for an existing commercial metaCASE environment called MetaEdit+ and it utilizes its concepts and features. The nature of the metaCASE environment raises some challenges on the implementation, and the main contribution of this thesis is to provide guidelines on how to solve these unique problems. This will benefit future efforts of building component retrieval tools in these environments.

KEYWORDS: reuse, component, software component, information retrieval, retrieval model, search representation, CASE tools, metaCASE tools, component-based development

1 Introduction

1.1 Background

It should be clear to everybody by now that the software industry is as important to the modern way of life as the more traditional areas of industry. It is quite alarming that there still remain some remarkable insufficiencies regarding the design, maintenance and quality control of software projects. Information systems are still often built very arbitrarily, without consistent utilization of the modern design methods and quality standards. Also the reuse of existing elements is often minimal. The maintenance of the resulting systems is difficult and very time-consuming, and therefore the negative effects of bad design and insufficient documentation may be visible for years. Therefore, the software industry is far behind traditional engineering areas such as road- or shipbuilding, where expense estimates and schedules are, if not perfect, at least much closer to reality. Ships or roads also usually function relatively robustly after being released to the public and do not suffer from mystical "blue screens" or annoying bugs.

The remedy to the low quality of software has been searched for since the late 1960s, and some proposals for potential solution have been made. According to Mili et al. (1995), it has become clear that the only realistic, feasible solution to increase productivity and quality is systematic reuse. During the last decade, one of the most promising solutions and also one of the most hyped concepts in the software industry has been the use of software components that allow software reuse in a coherent way. Software components promise to improve the quality and productivity of software projects by enabling the reuse of existing, well-tested pieces of software. This practice will make the software industry more like traditional industries, in which engineering products from the existing parts has been a common practice for a long time. (Persson 1998) For

example, a shipyard building a new cruiser will not fabricate every screw or steel plate needed in the construction of a ship. Instead, these small parts (or components) are bought from different vendors and are just brought together at the shipyard. In this way, the producers of different parts can specialize in the area to which they are most accustomed and of which they have the best knowledge. Also, these parts will be tested in many different environments, because the same screw or bolt may be used as well in a computer as in a Caribbean cruiser. This practice leads to well tested parts of good quality, which in turn – at least in theory – leads to end products of better quality.

1.1.1 Reuse in Software Development

Due to the nature of software industry, the reuse of components would be even more beneficial than in the traditional industry. This is because once a software component has been designed and implemented it can be duplicated infinitely practically without any cost. (Persson, 1998) Despite that a major portion of new software systems are typically constructed from scratch. This is very unfortunate, because studies show that much of the code in a system is functionally identical to the previously written code. (Jones, T. 1984). Nowadays, due to the hegemonic role of certain software companies the diversity in the software market is lesser.

Whereas in traditional engineering industry reuse means using parts designed and constructed beforehand, the basic idea of software components lies in reusing the existing code as much as possible instead of writing everything from scratch. (Persson, 1998) In theory, reusing code may refer to virtually anything that enables the use of old code in a new application: From copying and pasting code from a module to another to integrating a whole application to another application. Unfortunately, reuse that is based only on duplicating

the source code – especially the cut-and-paste approach – is very inefficient in the long run, because it is extremely hard to identify reusable parts in poorly documented code libraries. Also, this kind of reuse benefits only the implementation phase, whereas the modern view is that any lifecycle product could be reused (Frakes & Gandel 1989). In practice, research on reuse in other phases has not emerged until the last few years, and little is known about reuse of software that is not code. For example, reuse of requirements or software designs is mostly an unexplored area, although it might benefit the software development process drastically.

1.1.2 The Emergence of Software Reuse

Modular programming was the first invention that enabled user-definable reuse between programs (as opposed to internal reuse, which is reuse built inside the programming language). It allowed functions related to each other to be packed in the same module apart from the actual program. This module could be imported into other programs. A good example of this approach is the standard library for C language, where, for example, mathematic-related functions are packed in one module and file-manipulation-related in another. These modules can be integrated to programs relatively effortlessly, providing a possibility for reuse to at least some degree. The emergence of object-oriented programming (Smalltalk, Simula and Ada in 1970s and lately languages such as C++ and Java) and techniques (methods such as UML), have made reuse even easier to accomplish by introducing classes and objects. Classes encapsulate similar services in one, cohesive package. Besides the services (functions), classes also contain the necessary data for these services. These services may be accessed through the interface that the class defines.

The term software component is commonly used very loosely; for example, in UML a component may be any "distributable piece of implementation of a system, including software code" (OMG, 1999). However, we take a more strict approach to the component concept. Software components borrow the ideology of object-oriented programming and take it one step further: they encapsulate services and data, which can be accessed through a defined interface, **and** they are designed exclusively for reuse purposes (Korhonen, 2000). Components are also larger entities than classes. The interface is a fundamental part of a component because it represents the component to a developer. The interface should also include some kind of documentation to provide guidelines in usage and integration of the component (Tracz, 1991). By this definition, where interface and reuse are the crucial features of a software component, many object-oriented class libraries like Java's Swing classes or Microsoft's MFC can be seen as component libraries or collections (Korhonen, 2000). In practice, the use of software components is a constantly growing trend in the software industry. Still, there are some common problems that complicate accepting components as a standard means of information system development (ISD).

1.1.3 Problems in Software Reuse

The reason for the lack of reuse in many organizations is that reuse is a very complex issue to handle, and it requires large initial investments. It assumes quite radical changes on the organizational and technical levels of a company (Prieto-Diaz, 1991). In many companies the main reasons are the following. First, the companies considering the introduction of reuse have to face the economical and financial challenges. Second, the companies considering the introduction of reuse have to face a change in the organizational culture that impacts the software development lifecycle. Third, the companies considering the introduction of reuse have to face assessment and evaluation of the actual

benefits of reuse in terms of time and cost savings, improvement in the quality of products, and decreased effort. (Rine & Nada 2000) According to Frakes and Fox (1995), developers believe that reuse is beneficial to their work and efficiency, and they claim that they would rather reuse a piece of code than create it from a scratch. However, the reality suggests the opposite.

According to Prieto-Diaz, one of the fundamental technical problems in software reuse is organizing collections of the reusable components for efficient search and retrieval (Prieto-Diaz 1991). Frakes and Gandel (1989) phrase this by stating "A fundamental problem in software reuse is the lack of tools for representing, indexing, storing, and retrieving reusable components." Frakes and Gandel (1989) have defined this problem as a storage and retrieval problem: How reusable components should be stored and retrieved efficiently? This problem is very closely coupled with a representation problem: How the reusable components should be represented so that they can be found and understood. These two problems are the main focus of our study, and we will discuss them both thoroughly in their respective sections.

1.1.4 Component Distribution

A common solution to component storing and distribution is to store the components in a software repository that is available for all the software developers in the organization or company. This kind of repository is usually a multi-user database that stores the components as binary objects. The repository is typically integrated with maintenance functionality such as documentation tools, integrity checking tools etc.

According to Isakowitz and Kauffman (1996. p. 408), "a key ingredient for promoting software reuse in repository-based environments is providing

support to software developers who wish to search the repository to locate suitable software objects for reuse." When the number of components in the library is large, the developers can no longer afford to examine and inspect each component individually to check whether it is appropriate or not. We need an automated method that, at least, can reduce the number of potential components by some degree. Such a method would match an encoded description of requirements against encoded descriptions of the components in the repository (Mili et al, 1995). Unfortunately, repositories very often lack this kind of versatile querying methods that are available for traditional database systems (SQL for relational databases, for example). The lack of decent search tools may reduce the usability of the repository very drastically if the amount of components is high because developers cannot find the component they are looking within a reasonable time. To make the repository an acceptable distribution method and to make efficient reuse possible it is important to find decent solutions for querying the repository. (Zhang, 2000)

Although components and component search from repositories are a relatively new research area, there has been a lot of research on different information search and retrieval methods during the last few decades. Also, some methods for searching and representing reusable software have proliferated in recent years. These methods are drawn from three major areas: library and information science, artificial intelligence (AI) and hypertext (Frakes & Gandel, 1990). To date, AI and hypertext have been used only experimentally. Existing fielded reuse library systems use library and information science methods (Frakes & Poole, 1994). In this study we will concentrate on library and information science methods because we want to concentrate on methods that are known to be applicable to our own domain.

1.1.5 CASE-tools and Reuse

Since the early 1970's, one remarkable direction in the search for the cure to software crisis has been the emphasis and focus on analysis of the early phases of the ISD process rather than just the implementation. This has resulted in a huge number of different ISD methods, maybe most notably – at least from the success point of view – Rational Unified Process (RUP) with Unified Modelling Language (UML) by Jacobson, Rumbaugh and Booch. Along with the new methods came also plenty of software that allowed utilizing those methods with exclusive tools – *CASE tools* (Computer-Aided Software Engineering). A CASE tool is "a design aid tool that implements automated support for one prominent task of systems development." (Koskinen, 2000, p. 44) A closely related concept is a *CASE environment*, which is "a collection of CASE tools that cover several parts of the system development life-cycle. (Koskinen, 2000, p. 44) A CASE environment enables the definition of different aspects of the target system with tools that make use of graphical notations, diagrams, rules and constraints. With these features, the CASE tools should provide task related support for analysing, designing, versioning, change management and implementing an information system or their components according to a method. (Kelly et. al., 1996)

The traditional view of reuse and software components is that only the code and the executables can be reused efficiently (Frakes & Poole 1994). Despite that this is still the truth in practice, it would greatly benefit the software development if components and reuse could also be exploited in the earlier phases of an ISD process. This can be achieved with the use of the CASE tools. If the software being developed is designed to be component-based, there should be no reason to use other than component-based methods also in the design phases, and possibly even in analysis phase. Therefore, there should exist decent component and reuse support for the CASE tools as well as the

implementation environments. This development is still in its early phase and lacks standards and efficient methods.

Although decent commercial CASE tools with good support for popular methods are already available (for example, Cool:SPEX, Together HOW, Rational Rose) (Hänninen et al., 2000), the practice of using them has not yet grown very popular among IS community and most companies still utilize them only marginally or not at all, and even fewer utilise their full potential (Lending & Chervany, 1998). The reasons for this are not completely clear, but it seems that the managers do not always recognise the potential benefits of the methods. Moreover, the process of adopting new techniques is often quite complicated and requires commitment on the operational and management level. A study by Lending and Chervany also shows that system developers do not appear to be motivated to use the tools, and as Lending and Chervany (1998, p. 57) state, "Neither intrinsic motivation (the tool is fun to use) nor extrinsic motivation (the tool is perceived to be useful) is high." This is a remarkable problem, because the widespread adoption of CASE tools would also bring the software industry closer to traditional engineering industries where careful and thorough design before constructing anything is a generally accepted and practically applied convention of working.

1.1.6 MetaCASE Tools

The first CASE tools were simple text-based command line tools that allowed the user to define simple rules and dependencies. Newer methods adopted graphical representations and interfaces, and CASE tools followed by increasing the usability of the methods and making them more acceptable to the wide audience. Soon the CASE tools were outnumbered by methods, forcing users to adopt their built-in methods rather than to allow them to use the

method fitting best their organization and the development process. (Kelly, 1997)

In (Kelly et. al, 1996), a solution for this "methodology flood" - problem is proposed in the form of a flexible, fully configurable CASE tool – a *metaCASE tool*. The basic idea underlying a metaCASE tool is that instead of being bound to pre-defined methods provided by a CASE tool, the user can create a method best applicable to him or her. Methods are implemented as high-level conceptual models, which define the rules, objects, relationships and other aspects used in the method. This definition of a method is called *metamodel*, and it is a formal description of the characteristics of an ISD method. The activity of defining and creating metamodels is called *metamodeling*, whereas the activity of engineering existing methods to suit particular needs is known as *method engineering* (Heym & Österle, 1993). To be able to formally define this metamodel, some kind of data model of a higher level is needed to provide *metatypes*, of which the actual metamodels are constructed. Method engineering is carried out by constructing new methods from these metatypes. Logically, the model that defines concepts of metamodels is called metameta model. Interestingly, UML starts the meta-hierarchy at the application level and therefore considers the level that defines elements used in models as a metameta model. For example, classes are derived from metameta model object called metaclass. This is probably due to UML's lack of possibility to define your own modelling languages, which is understandable due to the fact that UML is a model level language itself. (OMG, 1999)

1.2 Goals

As discussed in the previous sections, there are major problems in adopting software components and component-based software development (CBD) in organizations. One of the reasons complicating and reducing the amount of reuse in organizations is that without the proper search functions finding the reusable part or component quickly becomes too difficult and time-consuming when the amount of components in the library increases. This problem is complicated by the fact that the search situations may vary greatly. In many occasions, the developer knows that the component he or she is looking for exists, but does not necessarily know how to look for it in a large repository. Even more challenging a situation emerges when the developer faces a problem situation and tries to find a suitable component from the organization's component repository. In this kind of situation, there might be no relevant components in the repository. (Hänninen & Äijänen, 2000) In this thesis, we will provide some solutions to these kinds of problem situations by proposing requirements for component search tool that would enable complex queries in component repositories.

1.2.1 Search Process

The search process can be roughly divided into two main phases. In the first phase, the query is created using the features provided by the search tool. In the second phase, the query is executed and a list of results is created.

In the first phase, the user creates the query using the search tool. Query creation methods vary much: a method may require knowledge of some formal

language such as SQL, it may be some higher-level query language or it may be purely visual. Different problem situations may (and probably will) require different kinds of search capabilities. For example, if the developer is aware of the component's existence, he or she probably wants to search for it by free text such as the component's name, creator or some words in its documentation. If the problem alone is known, the search task is more complex. In these situations it is practical to narrow down the number of components by some other method than free text search – for example, keyword search or search based on a classification (enumerated search, for example). However, the main goal in the first phase of the search process is to express the requirements for the component as accurately as possible. The expression capabilities are totally dependent on the search tool.

When a query is formed, it is translated into an internal query language expression. This expression is then tested with each of the components in the repository by the matching algorithm, which is the core of the retrieval process. Implementation possibilities for this algorithm are numerous, from simple binary Boolean matching to complex fuzzy algorithms that allow also partial matches to be retrieved. If the algorithm finds a matching component, it adds it to the *candidate component* list. When all components in the library are tested with the algorithm, the candidate component list is presented to the user. (Baeza-Yates et. al, 1999)

1.2.2 Major Issues in Component Search

According to Frakes and Gandel (1989), the major issues to be faced in designing a storage and retrieval system for reusable components are,

- How to classify and index the components,

- How to search for the components,
- How to evaluate the effectiveness of the system.

The main objective of this study is to answer the first two of these questions, and therefore solve how the searching of software components from a large component library/repository should be implemented. The third problem involves developing a universal metrics on component reuse, and therefore it is out of the scope of this study. We refer to these two problems as *classification and indexing problem* and *retrieval problem* accordingly to Damiani and Fungini (1995). In this study, we will use the component library of the MetaEdit+ metaCASE environment. It uses the GOPRR model as its meta-metamodel. This tool currently lacks decent component search capabilities, and this study will contribute to MetaEdit+'s development. Because of the nature of the metaCASE tools and the user-definable component model, the search tool should be flexible rather than fixed to a certain component model. This forms one of the most significant challenges in this study. Fortunately, MetaEdit+ uses a flexible, generic component model, which will be introduced in Chapter 4.

1.2.3 Research Problems

Our research problems are formed to reflect the major problems of component indexing and classifying described in the previous chapter. We will study how the representation method and retrieval model should be chosen and ultimately implemented in MetaEdit+ metaCASE tool. These problems and research goals are expressed with three research problems:

- What are the different possibilities in representing component search to the user? It is also possible – and even probable – that a single search

representation cannot satisfy all the querying needs, and therefore several different search presentations must be considered.

- What possibilities exist in realizing a component search in a GOPRR based repository? This should take into account the structure of the repository, how the component model is defined on the meta-metalevel along with the search representation and the retrieval model.
- How can the component search functionality be implemented in a metaCASE environment? This problem combines the questions above and adds the implementation-centric view, including the user interface design. The user interface design should be based on the chosen representation. Solving this problem includes the actual implementation of a search tool prototype in the GOPRR based metaCASE environment (MetaEdit+) although this part is not discussed in this thesis.

By answering the first two questions, we will create a theoretical basis for a flexible and efficient component search in a GOPRR based software repository. In practice this means finding the search representations most suitable for component search and solving other retrieval-related problems. We will find answers to the third question by implementing the prototype of the search tool into the metaCASE environment. However, extensive testing of the prototype does not fit into the scope of this study, although it is necessary to get some relevant and credible results on the usability of search tool.

A reasonable component search is naturally impossible if no descriptive data is stored with the components (Frakes & Gandel 1989). However, because we are particularly interested in a metaCASE environment instead of conventional CASE tools with predefined methods, we have to make the definition of the component model somewhat generic. Therefore, we have to define the component model generally so that the users can make their own component

models that are inherited from the generic model. Because the users of a metaCASE tool probably wish to build their own more specific component model (inherited from the generic component model) along with their own method, it is not necessary to fix the information stored with components – although we can provide some advisory guidelines for it.

1.3 Methods

According to Nunamaker, Chen and Purdin (1991), information system research should be carried out by using multi-methodological approach that integrates four strategies: theory building, system development, observation and experimentation. These strategies form a research framework where different research strategies are mutually supportive and therefore may utilise the outcomes of each other and exchange the data.

From the perspective of Nunamaker's et al. framework (1991), this study is mostly placed on the theory building area. As Nunamaker et al. (1991) state, a theory building strategy includes the development of new ideas and concepts, construction of conceptual framework, methods and models. Our approach is mostly theoretical, based on literature and building new theories on the pieces of existing knowledge. The constructive part is carried out by implementing a prototype of the query tool described in this study. This study does not involve experimentation or observation strategies, although they are a fundamental part in constructive research.

This study is conducted using two main methods. First, we create a theoretical background for the querying functionality by examining current literature on components, repositories and information retrieval. We will study the modern outlook of components, component storages and repositories, reusable software representation methods such as faceted and enumerated classification and

information retrieval related techniques such as indexing. Here the research method is reviewing the existing theories and empirical studies.

At University of Jyväskylä, the RAMSES (Reuse in Advanced Method Support Environments) project has studied components and has proposed component requirements for metaCASE environments (Korhonen 2000). These requirements also include the minimum requirements for a component search tool, and we will use these requirements as a resource in our study. However, we do not see these requirements as the end of the evolution, but more as a basis and a guideline to our extended study on the subject.

After building the theoretical foundation we will implement a prototype of a querying tool for MetaEdit+ metaCASE tool. The prototype will be implemented with Smalltalk programming language using theoretical and practical information gathered in the first and the second parts as a basis. This prototype will include basic functionality and an interface for simple queries, and provide all necessary encoding layers for the query types we found most important. We will exclude more advanced and complex queries such as recursive and visual queries because they do not fit in the scope of this study. Some significant querying possibilities such as hypertext will also be left out of the study. The querying tool will work on the top of MetaEdit+'s GOPRR based repository. The research method used in the prototype implementation is making deductions based on the theories and personal experiences.

1.4 Structure

Structurally, the thesis is divided into five chapters and four major structural parts. Each of these parts is dedicated to one major area of the research. Table 1

presents how these chapters and structural parts correspond to the research parts introduced in 1.3.

Table 1. Structure of the thesis

<table>
<tr><th>Chapter</th><th>Structural part</th><th>Research part</th></tr>
<tr><td>2</td><td>Concepts</td><td rowspan="4">Literature review</td></tr>
<tr><td>3</td><td rowspan="2">Component search representation & retrieval</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td><td>MetaCASE domain</td></tr>
<tr><td>6</td><td>Contribution</td><td>Prototype construction</td></tr>
</table>

The first part consists of the second chapter (the first chapter being the introduction) and its purpose is to introduce the concepts used in the thesis. We discuss about components and repositories commonly, and different definitions for software components and differences between classes and components. Some organizational aspects related to component-based development are also discussed.

The second part consists of two chapters, Chapter 3 and Chapter 4. This part deals with the two main aspects of component search: component representation and component retrieval. Chapter 2 deals with component representation, along with indexing and classifying that are closely related to representation. Here we take a look at the common classifying and indexing vocabularies used in different fields, including controlled and uncontrolled vocabularies, and review empirical studies conducted on their appropriateness on component or reuse representation. In Chapter 3 different retrieval methods are discussed, and some algorithms are introduced. The idea of binary versus non-binary search and its implementation possibilities are explained.

The third part consists of Chapter 5, and it introduces the domain of this research, metaCASE environment, and concepts associated with it. These include the GOPRR metameta model, which is the data model used in the MetaEdit+ environment and an integral part of this research. We will discuss the nature of model components and how they differ from their binary counterparts. We will also introduce component model designed exclusively for GOPRR data model. This component model will be the basis for the component query requirements represented in Chapter 6.

In the final (fourth) part, which consists of Chapter 6, we present the contributions of the research. We discuss the implementation of the search tool based on the theoretical aspects presented in the former parts. Different aspects of the search tool implementation, the search representation in a metaCASE environment, retrieval in a metaCASE environment, component browsing and query result representation. All these aspects are discussed.

Testing the prototype described in this thesis is naturally crucial, but the scope of this study does not include empirical testing; rather it is a subject for further research. This kind of testing is already being carried out by the RAMSES project.

2 Components And Repositories

In this section, we will discuss the basic concepts of component based software development. First we will define the concept of *component* for this study using references from former component literature. We will examine the differences between components and OOP (object-oriented programming) classes to address the fact that the components are more coherent and independent entities. We will discuss repositories as component libraries and what features we should expect of one. In the last part of this chapter, we will take a short look at the organizational aspects involved in component-based development.

2.1 Different Views of Software Components

Generally, when term software component is used it refers to some kind of binary object that can be integrated into a software project under development. For example, in UML (Unified Modelling Language) a component is defined as a "distributable piece of implementation of a system, including source code and business documents in a human system" (OMG, 1999). These distributable pieces include modules, class libraries or external executables. Another common approach (and maybe somewhat more old-fashioned) is the one used by Frakes and Gandel (1989), who define a component as any piece of reusable software: examples of this category include OOP classes, function libraries and UNIX tools like *grep* or *fopen*.

How the integration of components is accomplished depends on the type of the component and the programming environment, in which software is developed

– different software development environments provide different levels of abstraction. In the case of traditional programming language, such as ANSI C, the reusable libraries are included in the program with external declarations – in ANSI C these declarations are "include"-statements defined in the source code. This statement allows programmers to use functions and structures from the libraries defined in these declarations. In modern visual development environments (also known as *integrated development environment*s or *IDE*s) such as Visual Basic by Microsoft or Delphi by Borland the component integration process is abstracted to a particularly high "no coding required" -level: usually the components may be integrated in the project by dragging them with the mouse from some kind of visual component collection and then their services can be used in program code or sometimes even visually. In an object-oriented language like Java, the JavaBeans (or similar components) are also imported by some declaration, and then they can be explicitly created in the program code. It is important that in all these different kinds of software reuse situations the modifications made to original code reflect to every occurrence of that code in different software projects. Under this criterion, the cut and paste – approach is not component reuse, and probably not reuse at all in its truest sense. (Korhonen, 2000)

Numerous exact definitions for software components have been proposed in the literature, many of them somewhat contradictory and emphasizing different aspects. Allen and Frost (1998) emphasize the service aspect of the component by defining a component as "an executable unit of code that provides physical black-box encapsulation of related services. Its services can only be accessed through a consistent, published interface that includes an interaction standard. A component must be capable of being connected to other component (through a communication interface) to form a larger group." Some definitions do not explicitly bind the interface concept to the concept of a component. For example, Rational's Unified Process uses a broader and more general component definition by proposing that a component is "a piece of software

code (source, binary or executable), or a file containing information, (for example, a start-up file or a reassume file); a component can also be an aggregate of components." (Jacobson et al., 1999)

In this study we will take an *interface*-centric approach to binary software components. The interface can be seen as a joint or contract between a component and its clients. It is a formal definition that describes the services the component provides (or requires) but not how these services are implemented. One of the key aspects of the interface is to hide the inner implementation of the component from the user and to show only the information that is necessary to fully utilize the component. One of the interface-centric component definitions is proposed by D'Souza (1999, p. 15): "a component is a coherent package of software that can be independently developed and delivered as a unit, with defined interfaces by which it can be connected to other components to both provide and make use of services." Another good definition that emphasizes also the replaceable nature of components is made by Booch et al (1999): "A component is the physical packaging of model elements, such as design classes in the design model. A component assumes an architectural context defined by its interfaces. It is also replaceable, meaning that developers can replace one component with another, maybe better, one, as long as the new one provides and requests the same interface." As these definitions clearly demonstrate, in terms of coherence, encapsulation and ease of integration, function libraries and software modules (as C programming language understands them) are not coherent enough. As D'Souza (1999) and Allen and Frost (1998) state, software components should encapsulate their implementation and offer their services through a well-defined interface, which complies with some appointed standard. This is also Persson's (1998) view, as he states that standardized interfaces are one of the critical concerns when evaluating component technology's possibility of success.

A component's interface should be the only visible part of the component, and therefore it should contain all necessary documentation and information for users to understand how the component works and how it should be used (Korhonen, 2000). This kind of component reuse is called black-box reuse because the developer cannot inspect how the component is implemented internally. This is the way most current component technologies and frameworks such as Sun's JavaBeans and Microsoft's ActiveX/COM objects function. These frameworks define the standardized interfaces that each component must abide at a high level, and therefore they make the components easily attached to each other. Standardized and universally accepted frameworks are crucial to the success of component based software implementation because they make the efficient and fast integration of different components possible.

Model components used in earlier phases of ISD process are slightly different than components used in implementation phases, although most guidelines apply also to them. Special characteristics of model components are discussed in detail in Section 5.

2.2 Components vs. Classes

While technological solutions underlying the current component models are new, the basic ideas are not: they can be traced back to when objects originated, or even to earlier ideas of modules. Therefore, to fully understand the concept of components, it is necessary to address some distinctions between components and OOP objects (D'Souza, 1999, p. 15):

- A component is described by a specification of services it provides and requires from others, one interface at a time, whereas objects have

traditionally focused only on services provided. The description of an object does not include a specification of any of the calls coming out of that object.

- Component designs are purely based on interfaces; a component provides interfaces and it is implemented in terms of the interfaces of others. Objects may expose their data to other objects using them, although this is not considered a good practice.
- A component is typically larger grained than an object in OOP. It may be implemented with several classes, and its interface may be provided by a single "façade" class or by exposed internal instances. Components do not have to be implemented with OOP, and very often they are implemented with some lower-level language such as C, especially if they are performance-critical.
- Components should be connected at a higher level than API calls, e.g. *pipes, events* and *replication*; just as a component provides a higher-level part, so a connector provides a higher-level way to connect components.
- Components are units of packaging, and a packaged component can include the executable, the interface specification, test, default property, and "plug-ins". The form of that packaging differs across different technologies: Sun MicroSystem's JavaBeans relies on self-describing compiled representations based on Java's reflection API and standardized function calls; MicroSoft's COM requires separate type-library information to be explicitly registered; and COM+ moves towards a self-describing version.

Besides these, there are other less significant differences. Conclusively, it is clear that there are so many differences between components and classes that the research results gained in the studies of OOP classes cannot be applied to components as such. However, in many cases classes may be turned into components by encapsulating them in a coherent package and creating well-defined interfaces with some component standard.

2.3 Component Repositories

The adoption of software components does not benefit an organization appreciably if the components are not easily available to all the developers that may want to use them. Even small inconveniences in retrieving components can reduce the utilization of components very drastically (Banker et al, 1993). Therefore, it is extremely important to have adequate support for the use of components inside the organization to make full use and reuse of components possible. Reuse itself is a complex issue, and needs full support not only from the technical department but also from the organization's managing directors and infrastructure. Some of these infrastructural elements are practices for the creation, management and maintenance of components, decent usability of the component library and appointing the persons in charge of component management. There are positive experiences from the use of component libraries: when the library is well implemented it can provide remarkable financial benefits to the organization, and most organizations have fairly good technical conditions for the implementation of component library. (Prieto-Diaz, 1991) In this section, we will discuss mostly the technical issues of component libraries, and pay less attention to the organizational aspects.

As stated before, one of the most important conditions for component reuse is common component storage, where components are stored and indexed and where developers can retrieve them with relatively low effort. (Jeng et al. 1993) According to Maarek, Berry and Kaiser (1991), an adequate component library must meet the following basic requirements. First, the library has to provide sufficient amount of components from different domains, which can be reused either as-is (black-box reuse) or after relatively small modifications (white-box reuse). Second, the component library or storage has to be organized so that the

user can find the best component for his or her needs with relatively low effort. Especially the library should help in finding similar components, which may be not exactly what the developer was looking for but which can satisfy at least part of the developer's needs. (Maarek, Berry & Kaiser 1991)

When components in a library are code components (as opposed to model components, discussed in Chapter 5), they are stored in binary form – possibly with their source code. This is not sufficient for reasonable retrieval, because it is very hard to find right components from the library if queries are based only on the source code and practically impossible if only binary code is available. Therefore, some kind of "metadata", a formal description of a component's features and characteristics should be stored in the component's interface. Queries carried out in the component library are mostly based on this metadata rather than the component's implementation or source code. In a typical problem situation, the developer does not have accurate knowledge about the components in the library, and therefore components must be described accurately enough to make finding the applicable component possible. (Henninger, 1994) We will use the term repository or component repository when referring to component storage.

Because of the complex data structures in a component repository, the typical implementation of the library is object database, which can easily store complex data such as binary or model components with their corresponding metadata. Object database is also much more flexible than traditional relational database when representing complex relationships between components which are typical for GOPRR[1] data model and other data models used in CASE tools. For example, with GOPRR data model it is possible to create recursive data structures that would be very hard or impossible to represent with a relational database schema.

---

[1] The GOPRR model will be discussed thoroughly in chapter 5.

Handling data and its structures in a repository is more complex than in traditional relational database or file-based system. This is due to several factors, such as hierarchical structure of the repository, complex objects and different representations available in CASE tools. (Liu, 1996) Therefore, repositories rarely provide a formal query language (as opposed to SQL in relational database systems) and querying the repository is done via a programming language. Complexity issues and how they affect the queries carried out in the repository are discussed in Chapter 5.

Besides technical aspects, other fundamental concerns when realizing a component library are: organizing the library, maintenance and describing the components in adequate accuracy. Some aspects of consideration in a library's organization and maintenance are publishing the components, version control, access control and informing users about changes in the library. An approach proposed by Hadjami and Ghezala (1995) is to delegate responsibility of most of these tasks to a person (or people) specialized in the management of the component library. This person should be responsible for the publishing of the components. For example, he should take care that the library will not contain defective, malfunctioning, unfinished or insufficiently documented components. It is important that only the well-tested and semantically valid components are stored in the public component library. (Hadjami & Ghezala, 1995) Version control should also be arranged and organized thoroughly, because arbitrary updates of components may affect projects, in which those components are used. User access control is naturally needed for controlling access of certain groups or persons to the components.

Regarding the organizational aspects, one of the most important things is close interaction between the person in charge for the component library, component developers and the components' end users, the software developers utilizing the components. The person in charge should be aware of the activities and

operations of the development units so he or she could sensibly control component publishing and version updates. (Prieto-Diaz, 1991) The component library should also offer systematic support for informing the library users about changes in the library structure, new versions of components and new components published in the library. One possible way to achieve this is an automatic notifying system, which could inform users about updates and new components in which they are interested.

2.4 Summary

The most important contribution of this chapter is the definition of a component and its interfaces because a component definition is a fundamental part of this study. It is impossible to discuss aspects of component search if we do not know exactly what we mean when we talk about components, especially nowadays when the term component is used very loosely. If we want to find answers to our research problems, we need a solid foundation to build our view of representations and retrieval models on. The component definition is elemental part of this foundation.

Besides defining the component, we discussed some important concepts related to components, component based software development and component repositories. We briefly presented some history of software reuse, taking a little peek at reuse in structural and object oriented languages. We compared components and objects in OOP and addressed some distinctive differences between them. Finally, we discussed external requirements for successful component reuse such as component repositories and organizational aspects of component reuse. Examples of essential organizational aspects are component publishing, version control and access control of the component library.

3 Component Search Representation

In this chapter, we will concentrate on information retrieval technologies mostly from a user-centred view. We will discuss different ways to represent components and queries to the users and analyse different search situations that may call for different kinds of component representation. For this representation we will use term *component search representation* as proposed by Frakes and Gandel (1989). Search representation is dependent on the classification and indexing of the component library, and we will discuss different classification and indexing methods. It is notable that our approach to search representation is based on component search rather than component storing – these can be seen as the two sides of search representation. For convenience, we use shorter *search representation* referring to component search representation throughout the study.

As a main resource we will use literature and articles about information retrieval. Ph. D. William Frakes has conducted a lot of research on representing reusable software, and his work will provide a large part of the basis of this section. (Frakes & Gandel, 1989, Frakes & Poole, 1994) It is notable that we will not use the original terminology proposed by Frakes and Gandel (1989) because their terminology is somewhat confusing and, more importantly, includes terms overlapping with terms already defined in the metaCASE domain.

3.1 What Is a Search Representation?

The terminology concerning component search representation is quite diverse and unestablished. For example, Zhang (2000) discusses "search techniques",

Henninger (1997) sees a representation as a feature of repository's structure and Prieto-Diaz uses term "classification scheme". We will, however use term search representation proposed by Frakes and Gandel (1991). The reason for this is the fact that their study is by far the most complete one, and they have a long chain of articles on the subject, and their classification has the strongest theoretical ground. Furthermore, the term search technique is somewhat misleading because it may indicate any aspect of a search, while we are interested only in how the search should be represented to the user. However, we have made some modifications to the terminology proposed by Frakes and Gandel (1991) to make it fit our domain (the metaCASE environment) better and to avoid overlapping concepts.

A search representation can be defined as a product of classification and indexing procedures. Classification is the process of assigning a component to a category and indexing is assigning it to several categories - finding or creating a record to represent the component. (Frakes & Gandel, 1989) The end result of these activities, some kind of record or abstraction of a component, can be seen as a *search representation*. Because we are interested in how this search representation should be generated to support component search, we will apply this representation to the requirements of component search. In this chapter we are particularly interested in how the user perceives the search task.

A search representation can be seen as a joint between the user and the data the user is querying. Search representation methods are needed to provide the user some abstraction of the data and to present essential information about the data in a compact form. The basic goal of search representation is to make components that are appropriate to the user also recognizable as appropriate, and to make it possible to find such components with a procedure that seems logical and acceptable to the user. Usually, defining a search representation for a component means assigning data with some descriptive information such as keywords or a free textual description, or assigning components to some

classifications. A good example is a traditional library - the one from which you can borrow books. It would be practically impossible, or at least very inconvenient and inefficient to thoroughly inspect every book when searching for books on some subject. Therefore there is always some kind of card index – nowadays usually a computer database. Every book on the shelves of the library has a corresponding card in the file that describes some basic information about the book: author, name of the book, year of publication, some keywords about the subject etc. These cards are arranged in alphabetical order by the author's or the book's name, so in traditional paper-based indexes it is necessary to know either of these. Instead, an electronic system provides a possibility to search for the books by subject by entering some keywords.

When representing component repository to the user, the goal is the same as in the traditional library: we want to create some kind of card file schema that includes enough information to decide whether the component is applicable to the current situation or not. Whilst it is important to let the user see some kind of abstract about the component, the most important feature of the component representation is to make it possible to query the repository with different search criteria. No matter how intelligent the search-matching algorithm, it will be very difficult to achieve good search and reuse performance if components are indexed and represented poorly. According to Prieto-Diaz (1991, p. 92), a good search representation for a collection of reusable components should meet the following criteria:

- It must accommodate continually expanding collections;
- It must support finding components that are similar, not just exact matches;
- It must support finding functionally similar components across different domains;
- It must be precise and have high descriptive power;

- It must be easy to maintain, that is, add, delete and update the class structure; and
- It must be amenable to automation.

Existing approaches to component retrieval cover a wide variety of search representations, and how these requirements are met vary widely. The more of these requirements are fulfilled, the more complex and costly the representation is. However, according to Mili, Mili and Mili (1995) there is a practical limit in how complex queries can be for component reuse and search time to be worthwhile. Additionally, overly complex search representations are wasteful unless developers using reusable components are provided with computer assistance in formulating equally complex queries.

3.1.1 Structure of a Search representation

Using the three-level hierarchy proposed by Frakes and Poole (1994), the search representation can be divided to the three following levels. We have changed some of the names to better fit our needs and also to avoid confusion with the other concepts in our research domain. We have also adapted this hierarchy to our research. These modifications are noted.

- **Search presentation** is the interface between the user and the search method. Search presentation is not the user interface of the system but rather a way to define how the user sees the system and how he interacts with it. Some examples of search presentations are: free text search, keyword search, faceted search and visual query in the form of tree or other hierarchical structure. In this study our objective is to solve what search representations are the most efficient in the case of component search. This must also take into account the psychological dimensions of

the search such as different ways the user approaches different search tasks.

- **Search encoding** specifies the logical model of the search at a high level. Certain encoding methods are more suitable to some search presentations, but presentation should still be independent of the search method. Instead, encoding method is dependent of the underlying metameta model, in this case GOPRR. Search encoding is coupled with a *retrieval model*, which specifies how the components are matched to the query. Because the retrieval model itself is not part of the search representation hierarchy, different retrieval models are discussed in Chapter 5. Some examples of different retrieval models are Boolean retrieval, fuzzy retrieval and vector retrieval. A carefully chosen retrieval model may also affect the efficiency of the search and what kinds of query possibilities are available. For example, some retrieval models such as vector model support partial matches, some do not. Frakes and Gandel (1989) call this level the *representation* level, which is quite confusing because all three levels are parts of the search representation. Therefore, we have adopted the term search encoding from Mili, Mili and Mili (1999), which is used in similar context in their work. Frakes and Poole (1994) take a less activity-based approach to this hierarchy level, as they are more interested in the structure of data than how the query is executed. However, when discussing component search it is necessary to discuss also the retrieval process itself. In addition, the retrieval process also automatically involves the structure of the data model being queried.

- **Implementation** is the actual data that the representation abstracts. It is typically stored in a database or repository, if the data structures are complex. Implementation must be constructed to fit the meta-metamodel the data is based on. This research concentrates on different search

representations and retrieval models, though in the constructive part of the, research also the knowledge and study of object databases and implementation level of the representation is necessary.

In the three-level representation hierarchy a lower level constrains the levels above it. For example, a standard implementation of a Boolean retrieval model would not support a hypertext search representation (Frakes & Gandel, 1989). We will try to provide a solution for each of these encoding areas, using the top-down approach. That is, we will approach the problem from the user's point of view, and first solve the search representation problems. Then we will move to the lower levels of the hierarchy (retrieval model, implementation). This is necessary due to the nature of encoding hierarchy – we do not want to constrain higher levels due to implementation decisions. Moreover, we want to choose the lower levels that support higher levels, not vice versa.

3.1.2 Search Representation Classification

Search representation and indexing methods are classified on a scale, the endpoints of which are controlled and uncontrolled vocabulary. This scale refers to the degree of freedom the indexer has in representing an object with indexing terms or other search elements. (Frakes et al., 1989) Naturally, the other side of the coin is that the same degree of freedom constrains the queries that can be conducted in the library. In an uncontrolled vocabulary, the indexer has complete freedom in representing objects, which may result in more accurate descriptions of components. On the other hand, accurate queries can be harder to create if the query system does not control the forming of queries. Next we will discuss different representation methods with both controlled and uncontrolled vocabulary.

Indexing languages that are used for defining search representations consist of three parts. The first part is the terms – or the elements – that make up the language. The second part is the syntax of the indexing language – rules for combining elements and using them together. The third part is the logical relationship between the terms or the elements – practically speaking, the semantics of the language. Each part determines some aspects of the behaviour of the indexing; for example, the first part can provide very tight control over the elements that can be used in representing components, whereas the second part can still allow flexible and complex synthesizing of these elements. (Frakes & Gandel, 1989)

Because information retrieval methods are so diverse, we have chosen to evaluate only the methods that are found the most promising in former studies. We have excluded some interesting technologies such as artificial intelligence (AI) and knowledge based technologies and hypertext, because no sufficient former research has been conducted in these areas and they have been used only experimentally. Therefore, we will discuss mostly representation, indexing and classification methods derived from traditional library and information science methods.

3.2 Representations with Controlled Vocabulary

As the title says, search representations using controlled vocabulary restrict the namespace the users can use to define the queries executed in a repository. They force the components' developers and users to limit their presentation to predefined index terms. Controlled vocabularies aim at consistency and simplicity of the representation but at the same time sacrifice much of the freedom and versatility. There are two commonly used methods to derive these index terms:

- Terms are derived from the examination of the subject area. An index term is used only if it occurs frequently enough in the literature of the subject area or domain – this is called *literary warrant;* or
- Index terms are included if it is of interest to the user population – this method for deriving terms is known as *user warrant.* Usually, a combination of these two methods is used. (Frakes & Gandel, 1989)

Controlled vocabularies exist in two main forms: classification-based systems and keyword-based systems. These two forms are relatively similar to each other, because both systems are based on assigning classes or terms to objects. Differences between these two systems come out mostly in the way the components are arranged. Different systems and some of their applications are described next.

3.2.1 Classification-based Systems

Classification-based systems are based on classification schema, which is created for classification and search purposes. (Zhang, 2000) As noted before, "classification is the process of grouping items or objects with a shared characteristic or attribute into categories or classes. "(Frakes & Gandel, 1989) This classification should concern, besides relationships between things, also relationships between classes of things. Library and information sciences know two general classification systems: enumerated and faceted.

Enumerated classification is a commonly used classification method, which is based on placing information in categories usually structured in a hierarchy of subcategories – eventually forming a tree-structured hierarchy. The greatest advantage of enumerated classification is the ability to iteratively divide an

information space into smaller pieces and therefore reducing the amount of information that needs to be examined. (Henninger, 1997) Modifying and refining the search is also easy, because the only thing the user has to do is to move up or down the hierarchy tree. Enumerated classification is easy to use and understand, and its similarity to the UNIX file system makes it easy to use for most users familiar with UNIX. (Frakes & Pole, 1994) It is also used in most existing component repositories, in Select Component Manager for example (Hänninen et al. 2000). The disadvantages of enumerated classification include its inherent inflexibility and problems with understanding large hierarchies. It is very hard to change an enumerated classification system without restructuring the entire hierarchy, so the structure of the hierarchy is quite fixed. Enumerated classification also requires thorough understanding of the structure and hierarchies for users to be able to effectively retrieve information. (Henninger, 1997) Enumerated classification is also known as taxonomy.

Faceted classification evolved to overcome the rigidity of an enumerated classification system. It avoids the enumeration of component definitions by defining attribute categories (or facets) that can be instantiated with different terms. A faceted scheme is constructed by analysing a subject area and breaking it down into elemental classes or facets. Components are then described by synthesizing these basic facets. (Frakes & Gandel, 1989) Then, each of these classes is assigned a limited number of consistent terms whose meaning is clear to both the user and the librarian. Components are searched for by specifying a term for each of the facets or possibly to some of them. (Prieto-Diáz, 1991) Table 2 presents an example of simplified possible faceted schema for different methods of Java classes from the *java.util* library of Sun Microsystem's Java Software Developer's Kit 2. This example is a slightly modified version of the example given by Prieto-Diáz (1991). Faceted classification is somewhat similar to the attribute-value structures used in a number of frame-based retrieval techniques, except that faceted techniques use a fixed number of attributes (facets) per domain. (Henninger 1997)

Table 2: Java methods in faceted classification

Domain: Java methods			
By purpose	By action	By data structure	By return value
collection	add	stack	String
timer	remove	list	int
date manipulation	sort	set	Object
string manipulation	contains	other	Iterator
		map	null

A major advantage of a faceted system is the improved flexibility compared to enumerated schemes, because individual facets can be changed and redesigned without an impact on other facets. (Henninger, 1997) Therefore, all concepts do not have to be predetermined at the time of creating the classification system (Frakes & Gandel 1989). Some problems still remain. While describing components by selecting terms for facets is easy, it may be difficult for users searching components to accurately describe the information they are looking for. Users must also be well aware of the significance of each facet and the terms that are used in the facets. (Henninger, 1997) Naturally, there will quite often occur situations, in which current faceted schema is not sufficient to describe the functionalities of the components. In these situations there has to be some organization wide standard for extending the faceted schema. If every developer may edit the faceted schema unilaterally the faceted schema will soon become cluttered and its usability will deteriorate radically. (Prieto-Diaz, 1991) Therefore, organizations need distinct rules for maintaining the faceted classification schema.

Of these two classification-based systems, enumerated classification seems to be more widely used. For example, ACM's *Computing Reviews Classification System* is based on enumerated classification. However, most studies (e.g. Zhang (2001), Henninger (1997), Frakes and Gandel (1990) along with Prieto-Díaz (1991) who "invented" the faceted classification) suggest that faceted classification would lead to better reuse performance and easier maintenance because of better or even superior flexibility compared to enumerated systems. It seems that while an enumerated approach may be sufficient for document retrieval, the more complex component retrieval probably would benefit from the flexibility of faceted classification.

3.2.2 Keyword-based Systems

Keyword systems are somewhat similar to classification-based systems, especially faceted classification. They also classify items in the sense that they group related items under individual terms or phrases. Therefore, each individual term could be thought of as a class. The difference between classification-based systems and keyword-based systems is that in keyword systems, components or objects are arranged alphabetically rather than by classes or facets. Keyword search would require the information providers to provide each piece of information in the repository with appropriate keywords, which is a manual indexing process requiring skilful personnel. (Zhang, 2000) Typically, each object can be assigned with as many keywords as necessary, although systems can also place restrictions on the number of keywords.

Keyword systems exist in several different variations. In Frakes and Gandel's (1989) representation classification, keyword-based systems are divided into three major forms:

- **Subject Headings** is the simplest form of keyword systems. Subject headings are mere lists containing acceptable natural language terms and phrases that can be used in describing components. Subject headings offer hardly any structure because the terms are arranged alphabetically. The advantage of subject headings over classed systems is that they are much easier to create and modify. (Henninger, 1997) The subject domain does not have to be completely analysed before building lists of terms, and terms can be added afterwards without a need to modify existing terms.
- **Descriptors** are closely related to subject headings. Their vocabulary of terms is also limited, and they do not offer any specific structure besides alphabetic listing. The difference between subject headings and descriptors is that whereas subject headings are typically phrases or complete sentences, descriptors are shorter, usually single words. Therefore, descriptors are referred in many references as keywords. However, also subject heading –based systems are sometimes referred to as keyword systems, and Frakes and Gandel (1989) use the term descriptor to distinguish these two controlled keyword systems.
- **Thesaurus** is a special form of controlled keyword list. According to Frakes and Gandel (1989), "Thesaurus is a mechanism for describing the variety of relationships between terms within an alphabetical listing." It provides a classed listing within a framework of an alphabetical controlled keyword list. Thesaurus is found to be a useful representation method when dealing with fuzzy retrieval methods because it allows semantic relationships between index words and therefore it can find indirect matches more easily than other representation methods (Damiani & Fungini, 1995). Thesaurus is closely related to the faceted classification system in that it usually begins with a facet analysis of a subject area. The difference from a faceted classification is that in a thesaurus the facets are classified in semantically similar groups.

3.3 Search Representations with Uncontrolled Vocabulary

The basic nature of an uncontrolled vocabulary is straightforward: it does not place any restrictions on what terms can be used to describe an item. Representation consists of the full textual description, which is indexed using a "stop list" to remove frequently occurring words such as "and" and "the". The remaining text is used as an index to the document. (Henninger, 1997) Hence, the term "free text" or "full text" is usually used for an uncontrolled vocabulary. Retrieval from these kinds of system is usually done with simple Boolean search or vector search where weight values can be derived from the occurring frequencies of the terms. Therefore, relevant keywords are derived from their statistical and positional properties, thus resulting in what is called *automatic indexing*. (Prieto-Diaz, 1991)

The greatest advantage of free text search is the minimal effort required in indexing: since the terms are often drawn directly from the text of the indexed objects, the indexing can be highly automated. (Frakes & Gandel, 1989) This also guarantees that an object is described accurately in a representation because all documentation contained in the object is analysed in the indexing process. The downside of uncontrolled vocabulary is that while it is relatively effective for text-intensive documents such as books and articles, software products typically have characteristics that make it a less attractive approach. (Prieto-Diaz, 1991) Some examples of these characteristics are highly inconsistent documenting styles and motivations of component developers and the heterogeneity of terminology used in the many domains of information technology.

3.4 Other Representation Methods

Although controlled and uncontrolled vocabularies are the main focus of this thesis, we also take a look at some other representation methods that are less used in practice, or are too complex to be thoroughly covered in this study. The most important and promising ones are knowledge-based methods and hypertext. Next, we give a short description of them.

3.4.1 Knowledge-based Methods

Knowledge-based methods exist in many forms, and many of them have been tried for representing reusable components. Their greatest advantage is that the representation offers a powerful way of expressing the relationships between system components, which is probably extremely important for helping a user to understand the function of components. The problem with knowledge-based methods is that the knowledge needed for knowledge-based methods may be very expensive and time-consuming to acquire. This has proven to be true in many other fields of artificial intelligence. (Frakes & Gandel, 1989) Some knowledge-based methods for reuse are semantic nets, rules and frames. We will not further discuss them in this thesis.

3.4.2 Hypertext

The basic idea underlying hypertext is decades old, but it has not become popular until the last few years when, due to the emergence of the World Wide Web and the hypertext techniques it utilizes, it has gained wider acceptance.

Hypertext-based search allows the user to move among hypertext documents through *links*. The basic idea of hypertext is to organize information with nodes and links instead of typical linear structure. *Nodes* are associated with information blocks, and links represent different kinds of relationship between the source node and target nodes. (Zhang, 2000) This approach allows abandoning the conventional linear browsing of the document, and makes a free progression between documents and their contents possible. Hypertext is a complex subject, and discussing it thoroughly is out of the scope of this study. However, some applications of hypertext in the representation of the components are discussed in Chapter 5, although our representation solution is mostly based on the more traditional representation methods.

3.5 Representations in Current CASE-tools

We have carried out an empirical study on component functionalities of different commercial CASE tools (Hänninen et al, 2000). Five commonly used tools were chosen, based on their marketing promises and component support. These tools were HOW, Rational Rose, Select Component Manager, COOL:Spex and MetaEdit+. Zhang and Lyytinen (2000) have suggested a component functionality framework consisting of five component functionalities: abstract, retrieval, specialization, integration and maintenance. This framework was used as a basis for the study. From retrieval, the functionality in which we are particularly interested in this study, five features were chosen for further evaluation. Frakes and Poole (1994) recommend, based on an empirical study, that at least keyword, enumerated and faceted search should be included in a search tool if possible. Thus, these search representations were selected as the key features evaluated in the tools. Also hypertext was defined as a key feature. Results of the survey are shown in Table 3:

Table 3: Search representations in commercial CASE tools (Hänninen et al, 2000)

	HOW	Rational Rose	Select CM	COOL:Spex	MetaEdit+
Keyword	-	-	X	-	-
Enumerated	-	-	-	-	-
Faceted	-	-	-	-	-
Hypertext	-	-	-	-	-

As the table shows, commercial CASE tools typically lack most of the important search features. Commonly, only uncontrolled vocabulary, a.k.a. free text search, is supported. Only Select Component Manager supports keyword representation yet it is very simple to implement and easy to use. This is probably due the fact that Select Component Manager is not a CASE tool in a real sense, but rather an extension that eases the management of components. It is also interesting that Rational Software's Rational Rose, which claims to be a leading CASE tool, does not support any of the key features chosen for the evaluation. According to this survey, it is quite clear that search functionalities of commercial CASE tools are far from decent, and even inadequate when it comes to managing large amounts of components.

It is notable that some of these tools' manufacturers offer additional programs for component management. Rational Rose, for example, comes with separate application called ClearCase, which allows distribution of models and ease the management of the change process (OMG, 1999). However, this tool cannot necessarily be called a component tool because it does not distinguish between models and model components – although its marketing may claim otherwise.

3.6 Different Search Representations in Component Search

No former research on representation methods of components has been available for this study. Instead, some studies on representing reusable software exist, and we will rely on some of their results. These studies will provide us with some guidelines on how the library of reusable objects should be implemented. However, because of several significant traits of the metaCASE environment, these guidelines cannot be applied without several modifications. These modifications are discussed in Chapter 5, in which the results are conclusively summed and applied to a metaCASE environment.

3.6.1 Comparison Between Search Representations

In a study conducted by Frakes & Poole (1994), a repository of UNIX tools was created, and it was represented with four distinct representation methods: attribute-value (which is a slight variation of faceted classification), enumerated classification, faceted classification and keyword classification. Some of the results collected from the study were precision, recall, overlap and search time. Precision is the number of relevant items retrieved over the total number of items retrieved, and recall is the number of relevant items retrieved over the number of relevant items in the repository. Overlap measures the fact that some methods may retrieve different documents compared to others, and it is defined as follows:

$$X = |m1 \cap m2| / |m1 \cup m2|$$

Where $m1$ is the set of documents retrieved by one method and $m2$ is the set of documents retrieved by other method. Therefore, overlap is the ratio of number of relevant documents in the intersection of two methods divided by the number of relevant methods in their union.

Another important aspect evaluated in the study was how the users experienced the different methods used. This was evaluated by asking subjects to rank and rate each representation method they used. Rating was done on 1-7 scale (7 being the best). Besides that, users were asked to rate representation methods based on how they felt the methods helped in understanding components.

In the relatively thorough empirical study, several conclusions were made:

- When measured by recall and precision, no significant differences between the four representation methods were found. No method performed better than moderately.
- Although methods were not significantly different measured by effectiveness, they found different documents. Average pair-wise overlaps for the methods ranged from 72% to 85%.
- Users had no clear preference for a representation method. No method was regarded the best, or even as adequate, by all subjects.
- Results of the study were consistent with the findings from document retrieval experiments. This indicates that findings of document retrieval studies may well be applicable also in regard to reuse representation.

3.6.2 Guidelines for Choosing a Search Representation Method

Based on these conclusions and other aspects noted in the study, Frakes and Poole (1994) offer several guidelines for building reuse libraries. These guidelines will be considered when implementing the search tool prototype,

because they are based on an empirical study rather than mere theoretical discussion.

- Collections of reusable software should be represented in as many ways as possible, because none of the methods is sufficient for finding all the components for a given search. Having more representations will increase the probability of finding relevant items. It is also notable that individual users prefer different methods.
- If the primary factor when choosing the representation method is the effectiveness and cost benefit, the free keyword search is probably the best alternative. It is the least expensive, because the indexing task can be automated and therefore does not require human indexers.
- If the primary factor is search time, enumerated representation is the best alternative.
- None of the methods adequately support the understanding of the components. It is possible that representations based on more esoteric approaches such as hypertext or knowledge-based methods will be better in this area, but this subject require more research.

As we can see, nothing really surprising or radical emerged in this empirical study. Rather, it seems that representation method selection alone does not dictate the usability of a tool very much. Much more important is the content of the information represented. According to Frakes and Poole (1994), these guidelines have been used in various practical situations and they have proven useful when building a library of reusable assets. Therefore, especially due to a lack of more thorough studies, we can assume that these guidelines are relevant when considering component search. How these guidelines should be applied to component search in a metaCASE environment will be discussed in Chapter 5, in which we will thoroughly discuss the applicability of different search representations and special traits of the GOPRR model.

3.7 Summary

In this section we discussed different information representation methods. We explained the reasons for using search representations and presented the basic requirements for an adequate search representation proposed by Frakes and Gandel (1989). We divided different search representations in two main groups, controlled and uncontrolled vocabularies, and discussed the most used and field-tested representation methods from both of these groups. We shortly discussed what component representation capabilities are implemented in different commercial CASE tools. It seems that most current CASE tools provide very weak support for query needs, although some of the manufacturers provide separate tools for component management. Finally, we discussed a recent study where different search representations were compared empirically and presented some basic guidelines that were proposed based on the results of that study.

4 Component Retrieval Models

Classic models of information retrieval are based on the idea that every document can be presented with an adequate number of keywords describing its contents. These keywords are typically called index terms. Index terms are typically substantives, and they are derived from the representation method used. So, the search method is somewhat independent of the representation: any of the traditional representation methods can be expressed with a group of keywords. Index terms are rarely equal in the way they represent the contents of a document. Therefore, they can be assigned with the weight value that expresses how important the index term is to the representation of the document. (Baeza-Yates et. al, 1999) This feature requires support on the representation level.

According to Baeza-Yates and Ribeiro-Neto (1999), index terms form a logical view of the document whether they are generated automatically or manually, or whether they are derived from controlled or uncontrolled vocabulary. When this outlook is applied to the concept of the component, we can say that the logical view of a component consists of information in the component's interface and internal implementation because they represent the necessary facets of the component, forming an abstraction comparable to one created with index terms. (Hänninen & Äijänen, 2000)

According to Mili, Mili & Mili (1995), the purpose of a component retrieval method is to match an encoded description of a component's requirements against encoded descriptions (the representation) of the components in the library. They have also presented a model to formalize the retrieval problem. This formal model divides the problem space into three subspaces: actual problem space, problem space understood by the developer and query space.

Query space consists of the need the developer perceives translated into a query that component retrieval system can understand.

4.1 Ranking the Results

When retrieving from large software libraries, the query may return hundreds or even thousands of matches, especially if the defined query is broad. In these situations it is exhausting to inspect all of the retrieved objects, and sometimes defining a narrower query may be difficult. Here the retrieval model plays another significant role: the results must be ranked based on the estimated relevance to the user. Typically, documents or objects considered more relevant to the user are shown at the top of the results list. Ranking decisions are usually dependent on the ranking algorithm that "attempts to establish a simple ordering of the documents retrieved." (Baeza-Yates & Ribeiro-Neto, 1999)

A ranking algorithm is founded on some basic premises of document relevance. Distinct sets of premises constitute a starting point for distinct information retrieval models. The adopted retrieval model determines what is predicted relevant and what is not. Different retrieval models may yield radically different ranking results. Therefore, the retrieval model should be chosen carefully. For example, no ranking of documents is possible for a Boolean retrieval model, because it allows only binary weights, and therefore each object is either relevant or irrelevant to the query. The more advanced models such as a vector model, an extended Boolean model or a probabilistic model allow partial matches and therefore also ranking of the results, but the indexing task in these retrieval models is more complex. With traditional documents the weight values can be based on relatively simple metrics like word occurrences. However, when indexing software components, either we have to come up with new automated methods to set weight values, or the indexing task must be

conducted manually by component developers. This is probably acceptable because components are added to the repository rather infrequently.

4.2 Common Retrieval Models

Next, we will discuss four most essential information retrieval models according to Baeza-Yates and Ribeiro-Neto (1999). These include Boolean search, extended Boolean search, vector search, probabilistic search and fuzzy search models. Because these retrieval models are developed particularly for full text search, we will use the term document rather than component. We will discuss these search methods and their applicability to component search in Section 4.3. and their applicability to a metaCASE environment in Chapter 5.

The following is by no means a complete list of the retrieval models in use, and for example the coverage of AI-based retrieval models is intentionally left quite brief. Rather, we have concentrated on the models that have proven their applicability to complex retrieval tasks in the field.

4.2.1 Boolean Model

The Boolean model is based on set theory and Boolean algebra, and it is the simplest of the common search methods. A Boolean model does not take account the different weight values but every document is either relevant or not relevant. In the Boolean model, the query is formed by combining index terms and using connectivities AND, OR and NOT. The result of the search is the list of the documents that completely match the query. (Baeza-Yates et. al, 1999) The advantage of this model is the ease of use, formality and the fact that no additional information is needed: queries can be formed on the values of facets,

keywords or free text, whatever the search representation provides. The disadvantage of the model is that query often results in too many or too few documents due to the binary nature of the model. (Baeza-Yates et. al, 1999) Ranking the results is not possible for the same reason, which is probably the greatest problem with Boolean retrieval model in large libraries.

The Boolean retrieval model is the most common retrieval model in numerous legacy systems (most library systems, for example), and although more advanced models such as vector retrieval model are becoming increasingly popular, the Boolean model is somewhat popular due to ease of implementation and convenience of use. For example, Wildemuth, Friedman and Downs (1998) have studied the differences between Boolean search and the much more advanced hypertext, focusing mostly on efficiency and user satisfaction. According to their study, there were no statistical differences between the search methods, but users found the Boolean search more convenient to use. The Boolean retrieval model seems the simplest to implement and relatively efficient when the users are well aware of the contents of the repository.

4.2.2 Vector Model

Vector retrieval was developed to overcome the limitations of the Boolean model, whose binary values are too strict for the needs of information retrieval. Instead of using binary values with index terms, the vector model allows also partial matches to be included in the search results. This can be achieved by assigning the index terms with non-binary values. Search results are arranged in descending order based on the degree of similarity, which can be defined as correlation between vectors dj (document) and Q (query) (see Figure 1). The degree of similarity is the cosine of the angle of these two vectors. Because

arranging based on the degree of similarity typically corresponds better to the user's needs than Boolean results, it is proper to say that vector search gives more accurate search results. (Baeza-Yates & Ribeiro-Neto 1999, p. 27-30)

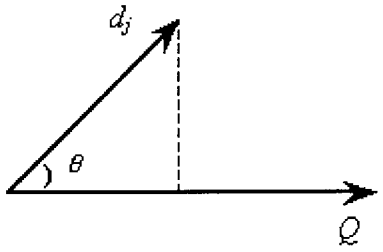

Figure 1: Degree of similarity between document dj and query Q

The elemental question in the vector model is how should we define the weight value of each term. In vector model queries are handled like documents, which makes matching them to each other easy. Let us assume that we have two sets of index terms, another representing the document (dj) and another representing the query (Q). First we must recognize the index terms that best describe the set Q. Then we must recognize the index terms that distinguish set Q from set dj. The former case is simple: the index terms can be selected explicitly (controlled vocabularies typically work this way) or implicitly by calculating occurrences of term k in document d. This incidence is usually referred to as a tf-factor (term frequency factor). In the latter case, we can use the inversion of tf-factor or idf-factor (inverse document frequency factor). The idf-factor is necessary, because index terms occurring in many different documents are not very good in distinguishing relevant documents from irrelevant ones. To put it simply, the vector model works by transforming documents (including query) into a vector space and comparing them to each other. (Baeza-Yates and Ribeiro-Neto 1999, p. 27-30)

According to Baeza-Yates and Ribeiro-Neto (1999, p. 30), a vector search model has the following advantages:

1. A vector model offers improved efficiency over a Boolean model, because the index terms may have weight values rather than being just binary.
2. Partial matches allow also documents that do not exactly match the query to be included in the result list.
3. Retrieved documents can be arranged by the similarity with the query.

The downside of the vector model (besides extra effort required when defining the weight values) is the fact that index terms do not have any relations with each other. However, this does not necessarily affect the search results and because of its simplicity, the vector model is one of the most popular information retrieval models. (Baeza-Yates and Ribeiro-Neto 1999, p.30)

4.2.3 Extended Boolean Model

The greatest disadvantage of the traditional Boolean model is the lack of term weighting and therefore also the lack of any kind of ranking within the answer set. This may often result in answer sets that are either too large or too small. Because of these problems, modern information systems are moving towards more advanced retrieval models, usually some kind of vector retrieval model. However, to accommodate the Boolean model to the requirements of modern information retrieval, various extensions have been proposed. (Baeza-Yates and Ribeiro-Neto 1999) One of the most promising is known as the *extended Boolean model*, which was introduced by Salton, Fox and Wu (1983). It extends the Boolean model with the functionality of partial matching and term weighting. This strategy allows one to combine Boolean query formulation with features of the vector model.

The extended Boolean model is based on a critique of a basic assumption in Boolean logic. For example, consider a conjunctive Boolean query q = kx Λ ky. According to the traditional Boolean model a document, which contains either term kx or the term ky is as irrelevant as another document, which contains neither of them. This binary decision criteria works in a simple query with only few query terms, but it frequently defies common sense. An analogous problem occurs when one considers purely disjunctive queries.

The basic idea of the extended Boolean model is to represent documents as vectors – similarly to the vector model. The model calculates the similarity of a document to a given query as the normalized distance between the document values and the query values in n-dimensioned space, where n is the number of query terms. The query may be parameterised with a parameter that selects the "order" of normalization. Let us assume this parameter is called p. A p value of 1 gives the average. A p value of 2 gives the Euclidean distance. Furthermore, a p value of infinity gives the MIN or MAX value of the document. The p = infinity model is equivalent to a strict "fuzzy Boolean retrieval". A p value of 1 corresponds to vector model retrieval. (Salton et. al, 1983)

4.2.4 Probabilistic Model

A probabilistic retrieval model assumes that every query defined by the user has a corresponding set of relevant documents. This document set has a corresponding ideal answer set. Query process can be seen as a process of describing this ideal answer set as accurately as possible. Because the user cannot be aware of all details of this ideal answer set, he or she must first approximate and - ultimately - guess them. The first result list is created from this first "guess". After that, the user must select from this result list the documents he or she considers relevant or irrelevant. Based on this answer, the

system refines the search. This process continues until the user is content with the result list, which is typically when it is as close to the ideal answer set as possible. (Baeza-Yates et. al 1999, p. 30-31)

The problem with probabilistic search is how to make the first guess as good a starting point as possible. Also, the model does not consider possible weight values in cognisance. An advantage of the model is that documents can be arranged based on the probability of relevance. (Baeza-Yates & Ribeiro-Neto 1999, p. 34)

The probabilistic model has many variations, among them the *inference network* model and the *belief network model*. These models use Bayesian networks and are based on the epistemological interpretation of probabilities. The queries are modelled as networks, and techniques resembling the ones used in artificial intelligence field are applied. A more thorough discussion of probabilistic models is way out of the scope of this study, and since most of these models are still in their experimental phase and not in commercial use, we can assume that they are yet not a relevant alternative in the context of component retrieval.

4.2.5 Fuzzy Retrieval Model

It is commonly acknowledged that when retrieving complex objects the retrieval will be more effective if the retrieval is based also on imprecise queries (Damiani & Fugini, 1995). The basic idea underlying the fuzzy search model is to classify objects in a way that borders between sets are fuzzy rather than exact as in a Boolean search. Objects are classified by assigning each object a value that represents its membership in the set. These values are real numbers between [0,1], where 0 means that object is not a member of the set, and 1 means a full membership of the set. The values between 0 and 1 represent

partial membership, and therefore the membership in the fuzzy search is gradual. The three most common operations on fuzzy sets are complement, union and intersection. (Baeza-Yates & Ribeiro-Neto, 1999, p. 35)

The fuzzy model can be applied to information search by building a synonym dictionary (or thesaurus) from the keywords. This thesaurus contains relationships between keywords, which are formed using a term-term correlation matrix. This matrix makes it possible to calculate the correlations between the document and fuzzy terms. Furthermore, the model enables the calculation of a document's membership in a group that the user has specified with his query. In this way, also partial matches can be retrieved by the query based on partial memberships. (Baeza-Yates & Ribeiro-Neto, 1999, p. 36-38)

Damiani and Fugini (1995) have proposed a concrete example of how the thesaurus can be constructed automatically, and how it can be used to support fuzzy component retrieval. Their study is especially relevant for us because the scope of the study was primarily ranged to software components. The thesaurus uses descriptors that are constructed from the code and its accompanying documentation. Descriptors are stored in an object-oriented repository, whose descriptors are classes with a class attribute part and an additional keyword list. These keywords are weighted with fuzzy values to "describe the behavioural properties of reusable components". (Damiani & Fungini, 1995)

The fuzzy model has not gained large popularity for information retrieval purposes. Existing experiments are conducted on small sets of documents and this makes it difficult to compare fuzzy search to other search methods. (Baeza-Yates & Ribeiro-Neto, 1999, p. 38) In component search, the biggest advantage of a fuzzy method is its ability to retrieve components that match the query indirectly. However, fuzzy search is probably not a relevant search method

until fuzzy search is further and more thoroughly experimented in information retrieval. (Hänninen & Äijänen, 2000)

4.3 Retrieval Models in Component Retrieval

The component search functionalities of different commercial CASE tools were evaluated in the empirical study by the Combo project (Hänninen et al, 2000). Most tools provide marginal support for any search methods, and those that did supported only Boolean retrieval. This works adequately with libraries consisting of less than a thousand components, but for libraries with thousands of components it is often too limited (Henninger, 1997). The reason for the popularity of the Boolean model is undoubtedly its low need for technical support and the possibility to automate the indexing process (Frakes & Poole, 1994). However, we think that it should be possible to assign also non-binary weights to the facets of the components. Because the traditional Boolean model does not allow non-binary weights it may not be the most appropriate model for the complex queries needed in component search.

In component retrieval – as well as in any kind of information retrieval, we usually want a close match and possibly might be willing to sacrifice precision for recall. That is, we might be willing to get some false matches as long as we do not miss any (or too many) true matches. In determining substitutability, we do not need the substituting component to have exactly the same behaviour as the substituted, only the same behaviour relative to the environment that contains it. (Zaremski, 1997) For this reason, some way of using non-binary values in component representation is necessary for efficient retrieval.

There are numerous techniques to assign non-binary values to the keywords of text documents, among them are: a popular model by Salton and Buckley (1988)

and one by Damiani and Fungini (1995). These techniques typically count occurrences of words in the text and use some kind of *Feature Weighting Function* to calculate non-binary values (or weights) for keywords. Although this technique is not perfect, it typically retrieves relevant documents with adequate accuracy. The performance can be improved by using some form of thesaurus.

In the case of components, using non-binary weights is not as straightforward as with traditional documents containing only text and possibly some information about the structure of the document. It is impossible to provide universally acceptable guidelines to component representation weighting, because different representations and retrieval models require different forms of non-binary values. The simplest way to assign weights is to use component documentation or its source code – if it is available – to derive the keywords and their weights. In this case, we simply count the occurrences of words and find the keywords and their respective weights by using some Feature Weighting Function. This technique is similar to one used with traditional document retrieval systems.

Although the technique described above is somewhat attractive because of its simplicity, it is usually too restricted and simplified for complex queries. This is due to the following reasons:

- The documentation of components typically varies too much in detail and formality to be decent sources for an automatically constructed keyword index.
- Keyword meanings are usually assigned by convention or by programmer preference.
- Human intervention is typical when describing component functionality, so at least the source code is too limited. (Prieto-Diaz, 1991, p. 92)

- Components are usually represented in more complex ways than keyword representation and commonly some form of more controlled and sophisticated classification such as faceted representation is used. (Henninger, 1997)

In enumerated, faceted or controlled keyword representations using some non-binary retrieval model is harder to accomplish and even harder or impractical to automate. We will discuss some rather limited possibilities to do this in Chapter 5.

4.4 Summary

In this chapter we concentrated on component retrieval: the process of matching the components in the repository to the matching algorithm with given conditions. This process also involves another very significant feature: the ranking of the query results. We discussed several retrieval models, which allow different retrieval features and ranking possibilities and varying accuracy and flexibility. Retrieval models discussed more thoroughly were traditional Boolean model, vector model, fuzzy retrieval model, probabilistic model and extended Boolean model. Of these models, Boolean model supports only binary values for queries whereas other models support non-binary values to different extents. Finally we discussed these retrieval models in the context of component retrieval and difficulties encountered when applying non-binary values to the facets of the components.

5 Model Components in a MetaCASE Environment

Our previous discussion on search representations and retrieval models is applicable to components in general, but from this point on we will narrow our scope and will concentrate on model components in a metaCASE environment. The purpose of this section is to introduce these concepts. We will propose a definition of model component and address the differences between code components and model components. We will also describe GOPRR meta-metamodel, which serves as a basis for all models in MetaEdit+ metaCASE environment.

Finally, we will introduce a component metamodel which will serve as a basis for all component models in MetaEdit+ and which our search tool prototype will be based on. The component model is mostly similar to the model proposed by Zhang and Rossi (2000) and it is greatly based on the 3C model by Tracz (1991) that is also introduced in this section. We have made some modifications, that we find are necessary, to reflect the requirements of modern component-based software development and, on the other hand, simplify the model. These modifications are described and justified in this section.

5.1 Code Components vs. Model Components

Usually, term 'component' is used to refer to a binary-form or to a code component (as defined in section 2.1.). The concept of code component is rather straightforward. In this thesis, we defined a software component as a software building block in binary form, which has a standard interface that other components or software systems using it understand and which is designed and built for reuse purposes exclusively. Although code components may ease

and speed up the implementation phase very radically (in fact, in modern ISD they are even crucial to the development process), their usability is limited only to this phase. However, in modern software engineering, more and more emphasis is placed on the earlier phases of software development process such as analysis and design. One example of the concrete evidence towards this tendency is the growing popularity of CASE tools (Laitkorpi & Jaaksi, 1999).

Binary code components can be used in conjunction with traditional software design methods, but true component based development can be only achieved if the components are also used in analysis and design phases. This brings in a new question: because binary components cannot be utilized in earlier phases, what kinds of components are used and how can they be presented in those phases? Zhang and Rossi (2000) provide one solution to this problem, and we will use it as a basis of model components in our study. Because these components are not binary entities like code components but rather a design artefact that represents the relevant aspects of the component in analysis and design phases we will call them model components. These design artefacts are typically presented in some kind of graphical notation. Whereas code components are used in the implementation phase in the integrated software development environment (IDE) such as Visual Basic or Delphi, the model components are used in analysis and design phases in CASE tools such as MetaEdit+ and Rational Rose.

Whereas code components already have some strong standards like CORBA, COM and JavaBeans, model components lack widely accepted standards. It is very common to every CASE tool to have its own view of model components – a situation that can be compared to the one with code components ten years ago. UML defines a component as a "physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces." (OMG, 1999) This is also the definition used in a large portion of CASE tools. However, this definition also allows using conventional function libraries and

files as components, and we find this kind of component use too wide and therefore weakening the concept of component. Component use and general reuse in CASE tools are less studied issues, and as Frakes and Fox (1995) state, CASE tools are not currently effective enough in promoting reuse. (Frakes & Fox, 1995) However, it should be noted that a current trend in CASE tools seems to be emphasizing reuse more than earlier, and some tools like Rational Rose and COOL:Spex provide reuse support to some degree. Unfortunately, the support for reuse seems to be inadequate in practical situations. (Hänninen et. al, 2000)

5.2 GOPRR Metameta Model

The purpose of a meta-metamodel in a metaCASE environment is to be a generic and universal model that is able to define all aspects of methods without the need to codify method knowledge into tools. This is because every hard-coded piece of method knowledge will affect and ultimately reduce the tool's flexibility to define completely new, innovative methods. Therefore, the goal of the meta-metamodel is to define a small set of conceptual elements that can be combined to form any method. One candidate data model of metamodelling (for the definition of metamodelling see to Chapter 1), called GOPRR, is proposed by Smolander (1993). GOPRR stands for Graph, Object, Property, Relationship, and Role, which constitute its five conceptual elements. GOPRR model is an evolutionary extension to the OPRR (Smolander, 1991) model, which in turn has evolved from the well-known ER (Entity-Relationship) model. (Kelly et al, 1996) The conceptual elements of the GOPRR meta-metamodel are:

- **Object**, which represents the independent design element. Objects are typically represented by graphical symbols in graphs and they present

different singular entities of the system. Objects may contain Graph elements via techniques known as *decompositions* and *explosions*. These graphs usually contain more accurate presentation of the object in a lower level. A typical example of object element is a class type in UML.

- **Property**, which represents an attribute of object or other non-trivial GOPRR element (that is, any other than Property itself) and is always associated with some other element. It typically does not have any graphical presentation other than simple textual form – and even this is not mandatory. For example, class name or method name in UML.
- **Relationship** is an association between two objects. It is presented typically with a line between objects, and it may have describing properties. For example, inheritance between classes in UML is a relationship.
- **Role** defines how an object participates in a relationship with another object. Like relationships and objects, also roles may have properties. For example, superclass in UML inheritance relationship.
- **Graph** is an aggregate concept, which contains other conceptual elements – objects and their roles and relationships. Graphs can also contain objects that contain other graphs, thus providing possibility to create recursive structures. Some examples of graphs are UML class diagrams or data flow diagrams in Yourdon's SA/SD. (Kelly et. al, 1996)

Besides defining conceptual elements, the GOPRR meta-metamodel provides the following additional features:

- Modelling a unit called project, which can contain several graphs using different notation techniques and helps the organization and linking of graphs and methods
- Constructs for generalization, specialization and polymorphisms.

- Method integration constructs such a possibility to reuse all GOPRR objects in several graphs. Changes made to one object reflect to other graphs where the same object is used.
- Rules for checking model integrity. These rules can be set by defining allowed bindings (combinations of roles, relationships and objects) for different graph types. Properties may have rules or constraints in addition to normal type restrictions.
- Multiple presentations for same object. These different presentations include graph, table and matrix presentation.

(Kelly et. al, 1996)

When implementing the search tool for a GOPRR based repository, it is necessary to take into account the flexibility of the data structures. Because GOPRR allows users to define new metamodels, the search tool must be flexible enough to be able to query these custom data structures.

5.3 Component Model for Analysis and Design Phases

One of the reasons why component-based design has not been widely adopted is that most current CASE tools lack adequate support for components. Most CASE developers have merely created a new symbol and called it 'component', without further thinking of its characteristics and features. Many tools have also adopted UML's definition for component, which is much too wide to be useful in many reuse situations. For MetaEdit+, Zhang and Rossi (2000) have proposed a component model based on the GOPRR data model, which formally describes how a component should be composed in a CASE tool so that it can be used in analysis and design phases. Next, we will discuss this component

model and theoretical foundation it is based on – particularly the 3C model by Tracz (1991).

5.3.1 3C Model

The model proposed by Zhang and Rossi is based on (Tracz, 1991), which introduces a framework for software components known as the "3C model". This framework views a component as a compound of three different perspectives: Concept, Content and Context. These three different perspectives are described as follows:

- Concept of the component represents the part of the component that is visible to the user, that is a description and documentation of the component.
- Content is the implementation of the component, the part that describes how the component does what it is supposed to do.
- Context is the most abstract and quite rarely perceived part of the component. It represents the environment of the component including other components and resources the component requires to work properly. This may also include descriptions of former uses of the component. Tracz divides the context in three parts: conceptual context, operational context and implementation context, each presenting different view on the component's environment.

The purpose of Tracz's framework is to provide formal basis for different aspects of the component. (Tracz, 1991)

Zhang (2000) has used this 3C model as a foundation for her earlier component specification in a metaCASE environment. In her model the concept is realized

by interface specification. As we have previously described, an interface provides abstract information about the component's functionality and usage. As stated in Chapter 2, in our opinion it is an elementary part of a software component. In Zhang's model, the interface specification is defined as a faceted schema, which uses predetermined facets and their values to specify a component interface. The faceted schema with the values that are set to the facets is typically the part that is queried the most when searching for a component, and it should include most of the necessary information about the component and its usage. The content, which may be a chunk of source code or a binary executable in a code component, is typically provided as a diagram or graph (or a set of graphs) presenting a component's functionality in the CASE environment. The context part of the component is a complementary part that defines the contextual dependencies between components and, in Zhang's (2000) words, "domain of applicability". It "provides enriched contextual information including definitional dependency, reuse dependency, usage context and the implementation context". Zhang (2000) suggests that the context part could be realized using some kind of hypertext link that she calls contextual link.

5.3.2 RAMSES Component Model in 3C Framework

The RAMSES-project (Lyytinen et al., 1999) has improved this component model further. According to Karlsson (1996), a component must include diverse information such as component classification, information on the functionalities of the component, information on how to use the component etc. The original component model does not offer a possibility to present this information sufficiently, so the model is improved to better correspond with Karlsson's requirements. Another important aspect that applies to a metaCASE environment is the requirement of flexibility. Because in a metaCASE

environment, designers of the components may want to use their own faceted schema instead of the one provided with the tool, the component model must be flexible enough to allow creating new schemas. (Zhang, 2000)

Below is a graphical representation of the metaCASE component model proposed by the RAMSES project (Figure 2)(Zhang & Rossi, 2000). This model is more a metatype of components because it is not used "as-is". Rather, users of a metaCASE tool will define their own methods and instantiate more specific component models from this metatype. The purpose of different elements and how they correspond to the 3C model by Tracz (1991) is explained next. These definitions are mostly from Zhang & Rossi (2000), although we disagree with them on some details.

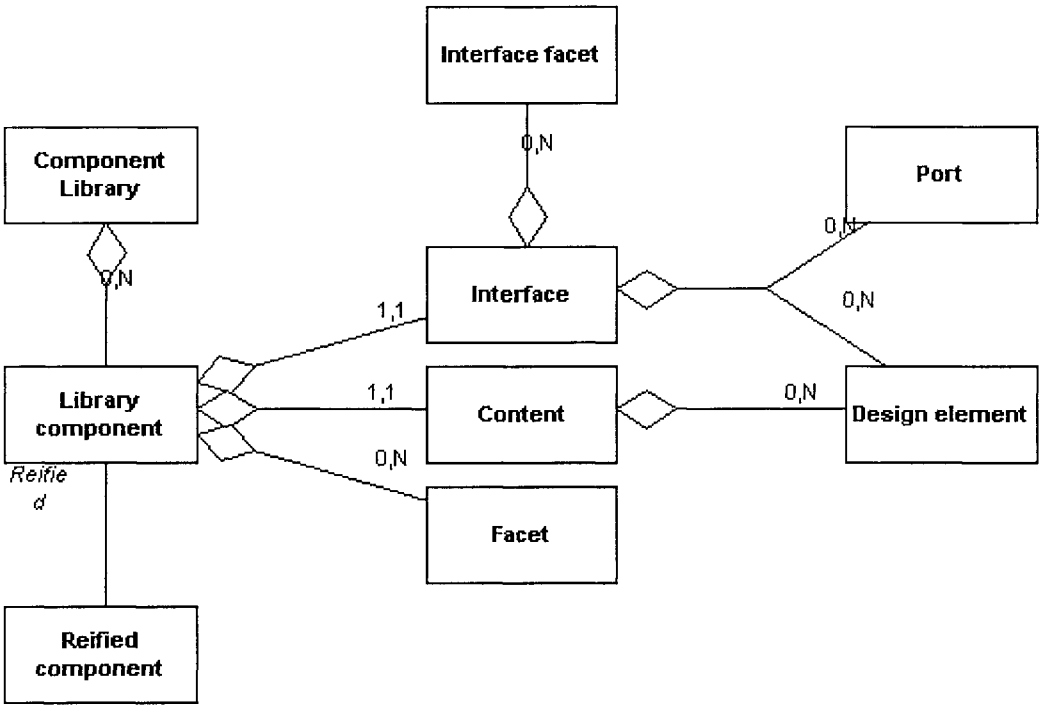

Figure 2: Component model for metaCASE environment by RAMSES (in UML class diagram notation)

Content, the internal implementation of the component, is represented by content part, which includes one or more design elements. These design elements can be any GOPRR-elements or their combinations. Thus, the content part can vary from a single class or property to a complex diagram. In the RAMSES component model, as seen in Figure 2, the content part is represented as a *content part*. This part does not contain the implementation itself but is rather an aggregate of *design elements*. These design elements contain the actual implementation, and they may be any GOPRR element. In practice, the content part is implemented as a GOPRR object element that contains a graph element as decomposition. The objects in this graph are the initial content parts, which naturally may contain other graphs of various types. The reason for this arrangement is the simplicity of graph element as a concept and that it allows perfect compatibility with all models formed from GOPRR elements.

Concept, which consists of the interface part of the component, provides the list of properties of the component, which represents the general information about component status, functionality and constraints. In the RAMSES component model, as presented in Figure 2, this part consists of the *interface* part, which in turn is an aggregate of *interface facets* that are interface properties describing the component's services, attributes and other traits. In theory these interface facets can be any GOPRR elements, but it may be necessary to make some restrictions on how complex structures can be used. This is due to the reason that if the interface becomes over-complicated its value to the user may deteriorate. The whole purpose of the interface becomes questionable if it cannot abstract the contents of the components in a relatively simple structure. Also, forming sensible queries may become overly challenging if the interface part is overly complex. We think that the interface should contain all the information that is necessary to the user of the component. In the case of black-box reuse, the interface should be the only part visible to the user and the content part should be revealed only when the developer wants to modify the implementation of the component.

Context, describes how the component communicates with its environment. We are particularly interested in *conceptual context* as defined by Tracz (1991). The conceptual context concentrates on the relationships between the component and other components. Our view of conceptual context is somewhat different from Zhang's (2000). Zhang sees the context more as a collection of the situations and the project's the component is used in. This involves also the timeframe of the component's usage and modifications made to it such as different versions. Zhang proposes that some kind of hypertext technique could be utilized in realizing these features. In our opinion, this kind of information should be provided by component repository rather than contained in the component itself. Naturally this presumes that the component repository keeps track of component use and the relationships between them (e.g. version control).

In our opinion, the context should describe how other components connect to the component to make use of its services. Some components may also require some services from other components to function completely, and the context part also describes these requirements. In the RAMSES component model these functionalities are implemented via the use of *ports*, as seen in Figure 2. These ports provide connection points to the design elements (content) through the component's interface. Relationships between ports, interface and design elements are illustrated in Figure 2. Ports exist in three different forms. The first form can be thought as a GOPRR relationship, in which one end is left open and the role is defined only on the other end – the end residing inside the component. The second form provides only the role; otherwise it is similar to the first form. The third form is an object that may participate in relationships that are connected to the component. These ports are a part of the interface, but because they have a special role in the component's usage (whereas the concept part is more a mere description of the component) they are not classified as a part of the concept but rather a unique feature.

The *library component* and *reified component* on the left side of Figure 2 illustrate the process of extracting the component from the library. The instance of a component in a software project is called reified component. The alternative name for reified component could be instantiated component with the difference that the reified component may be merely a reference to the component in the database. In this situation, all the changes made to the component will be reflected in each reference, whereas the instantiated component is totally an independent entity. Components in the libraries are called library components.

5.3.3 Modifications to the Component Model

We argue against this component model in certain details. First, in the original model, it is possible to define additional facets that are "properties of component not presented in content or interface" (Korhonen (ed.), 2000). In our opinion, these kinds of facet are not necessary. All of the information we want to show to the user of the component in black-box reuse should be contained in the component's interface. The reason for this is twofold: First, it is much clearer to include all the necessary information in one part of the component than to divide it into separate parts. Second, in white-box reuse all additional information needed to describe the internal functionality can be presented with the design elements because as GOPRR elements they are flexible enough to contain any necessary information. Moreover, if we want to include some emergent properties in the contents of the component we may add these properties to the content part, that is the object that contains the design element via decomposition. Thus, there is no need to separate additional data in the facets. For these reasons, we will leave the facet part out of the component model.

Another modification we propose to this component model concerns the relationship between a component and its interfaces. In the model proposed by Zhang & Rossi (2000), each component has always one - and only one - interface, which contains all necessary interface elements, documentation and ports. However, we see that this practice may be too constraining in some applications. We see no good reason to limit the number of interface to one; instead, many situations may arise, in which allowing multiple interfaces on a single component may come very handy. For example, some component models like Microsoft's Component Object Model (COM) and JavaBeans allow multiple interfaces, and when modelling software based on one of these models allowing only one interface would be a severely limiting factor. More generally speaking, it is probable that interfaces of large components are divided in several smaller interfaces because of the overwhelming amount of services (operations) they provide. These interfaces may group the services in logical sets, and users of the component may ask for one interface at a time. Also, for example D'Souza (1999) and Laitkorpi and Jaaksi (1999) suggest that components offering different interfaces for different tasks are one of the key elements of component based software development. We think it is quite clear that to be able to model situations emerging in software development using current technologies, the possibility to use multiple interfaces whenever necessary is required.

5.4 Summary

This section laid a theoretical foundation for research on metaCASE environments using model components. We explained these concepts and addressed their significance in information system development. We introduced one meta-metamodel, GOPRR, which is used in the MetaEdit+

metaCASE environment. This model consists of five basic concepts that are building blocks for all the methods that can be formed with GOPRR.

The other important purpose of this section was to define a component model for MetaEdit+. This model is mostly based on the work by Zhang and Rossi (2000) whose view on components is partially based on Tracz's theory on megaprogramming (1991). The basic concept of the model is to construct components by defining their three key areas: content, concept and context. Each of these areas provide a different perspective on the component. We described how these perspectives are reflected in the actual component model by defining corresponding parts in the model. Mostly we agreed with Zhang and Rossi, but we have refined some details and modified some parts to better reflect the practices of modern component based software development.

6 Component Search in a MetaCASE Environment

This is a conclusive chapter based on the four previous ones. Here we will sum up previous conclusions and results, and present a model for the search tool in the metaCASE environment. Concepts discussed in this chapter are mostly based on the GOPRR meta-metamodel (Smolander, 1991) introduced in the previous chapter. The reason for this is that GOPRR is the foundation of MetaEdit+ metaCASE environment's object-oriented repository, basis for all objects that may exist in this repository, and serves as a basis for our component model also introduced in the previous chapter. We try to address special requirements stemming from the GOPRR based object-oriented repository. The final goal is to propose a model for component search, which would be flexible enough to support different component models customized by users, but which would also provide an adequately robust search system for sufficiently exact search results. This is difficult, because these two requirements are somewhat contradictory: When we make the search tool more general and customisable, we must sacrifice part of the accuracy because the queries must be presented at a more general level. The other challenge emerges from the nature of GOPRR and object-oriented databases in general. Although there are some query languages for object-oriented databases such as OQL (Object Query Language), many object-oriented databases do not utilize these languages and the only way to implement queries is with a lower-level programming language.

The model for search and representation, which we will propose in this chapter is based on the primitives and tools of MetaEdit+ metaCASE environment implemented in Smalltalk language, thus allowing the same flexibility that the GOPRR metameta model allows in other design solutions. Also, this model can be in large part implemented on the top of the existing structures of MetaEdit+

(particularly the dialog system) so the additional programming required can be kept to a minimum. This will also make the maintenance and later modifications easier.

6.1 Characteristics of Search Representations in a MetaCASE Environment

A metaCASE environment such as MetaEdit+ involves several unique challenges in representing components. Because users of a metaCASE environment can create their own methods and respective component models via method engineering and metamodelling, we should not restrict the ways users can represent their components. Therefore, in principle a metaCASE environment should include at least fundamental support for as many representation methods as possible to be able to satisfy the needs of different users in different domains. However, implementing all possible representation methods may be an overwhelming task, and in some repositories far too costly. For example, it would be quite unwise to spend hundreds of programming hours to implement some exotic artificial intelligence –based representation method, when its practical benefits may be very few. It is wiser to offer users a basic set of well-implemented and well-tested representation methods that can respond to most representation needs. (Frakes & Poole, 1994)

Another special trait of the metaCASE environment is the different "sizes" of the components. The attribute representing the variety of component size is called granularity level. The smallest components are atomic units such as class, inheritance or property on the meta level and instances of these on the instance level. Some components are aggregates, which form larger structures tying smaller entities together semantically. These kinds of structures are graphs such as class diagrams or data flow diagrams. MetaEdit+ also allows graphs to

contain other graphs or sub-graphs via explosion or decomposition. Furthermore, there are larger components that form collections of several (meta)models to represent a complete methodology or an information system development project. (Zhang, 2000) The term *granularity* is used to describe the "size" of a component. MetaEdit+ divides components into three different granularity levels: the project level, graph level and unit level. These three levels correspond to different kinds of components described above, the project level being the most coarse-grained and the unit level the most fine-grained one. (Zhang & Lyytinen, 2000) Naturally, when using graphs or even larger entities as components the concept of interface becomes somewhat faltering. The solution we will propose for this is to encapsulate a graph into so called *component object*, which includes necessary interface facets for the component. Also parts of the graph may be used in the interface. We will discuss this approach in more detail later in this chapter.

6.1.1 Representing GOPRR Data Model

MetaEdit+'s data model, GOPRR, restricts its requirements for the representation methods. In GOPRR, all objects are formed from five basic elements, and therefore our representation method only needs to be able to represent these elements. Things are complicated by the fact that arbitrarily complex recursive data structures can be created from these five elements. The simplest example of a complex data structure could be something like two classes - with some properties such as class name and attributes - that are connected to each other with a relationship and roles. More complex structures occur when objects in higher-level graphs are represented more accurately in lower-level graphs. These hierarchical structures are formed in MetaEdit+ by using techniques of explosion and decomposition. Theoretically, these structures - as well as their properties - can be recursive, which of course makes

the query task extremely complicated. MetaEdit+ also supports project-granularity components that can contain many kinds of graphs even using different modelling techniques – for example, SA/SD data flow diagrams and UML class diagrams may exist in the same project.

The final, and very focal trait derives from the fact that in a metaCASE environment the methods and their data types - including components – are fully customisable, within the constraints provided by the GOPRR model. We have introduced a strict yet flexible component model in Chapter 5, and our representation must adapt to the constraints and limitations it creates. On the other hand, the representation method should be flexible enough to provide sufficiently accurate search results in all different custom component models derived from the model proposed by the RAMSES project.

Because of the arbitrarily complex structures MetaEdit+ allows it is quite tempting to restrict queries to the component's interface(s) and hide all the content from the queries. This is so called black-box reuse. In our opinion, this should be enough in most reuse situations, because - as stated earlier – the interface(s) should contain all the necessary data about the component and its usage. In most cases, the internal implementation of the component is irrelevant to reuse purposes if the component works as it should work and its documentation is sufficient.

6.1.2 Representing RAMSES Component Model

The elemental problems involved in defining the component representation in MetaEdit+ can naturally be solved by defining component representation for the general component model proposed by the RAMSES project (Zhang & Rossi, 2000). When we define how to represent different parts of this generic

component type we can generate a representation to all component models inherited from the general model.

The first issue to decide when creating representation for components is the scope of the search. It is possible to include all the aspects of the component in the search – including content and its design elements. However, this search approach has some shortcomings. First, the contents usually contain mostly implementation information that is irrelevant to the user of the component and this may result in irrelevant matches. (Henninger, 1997) Second, because arbitrarily complex structures may occur in a metaCASE environment the retrieval process becomes very complex if the design elements contain recursive structures. If we choose to allow the query tool to access only the interfaces of the components, representation issues are simplified radically. In theory, it is sufficient to allow the query to access interface facets and ports (as defined in Section 5.1.1) of the component. Of course, because also interface facets can be any GOPRR-elements, they can also contain complex structures and query must be able to handle these kinds of situations effectively. Probably some kind of restrictions in interface facets may be wise, or at least the retrieval process should be limited.

MetaEdit+ features the MetaTypePainter class (Kelly, 1997), which automatically generates window specifications of dialog windows for different GOPRR data types defined in MetaEdit+. This feature is greatly based on Smalltalk's ability to associate variables in program code with UI controls in the dialog. It creates dialog windows based on the type's properties dynamically. When the user creates a custom component model, the dialog window based on the new model is created automatically whenever it is needed. This window is similar to standard dialogs in window environments and supports the following controls in presenting the properties of the GOPRR data types such as diagrams and objects:

- *Textbox* is used to enter a free text value. These are used for properties, the values of which are not limited to existing, pre-defined ones. *Textfield* is a variation of textbox that allows larger chunks of texts with multi-line and scrolling support.
- *Combo box* is used to choose one value from a list of pre-defined values. This control exists in two different variations: one is strictly limited to the pre-defined values while the other permits adding new values. New values can be written as free text, and they are added to the list of values the combo box provides. This should be used with caution because it can easily lead to overuse of entering capability and too many values in the list. In the search context, adding new values should be restricted.
- *Listbox* can be utilized in two different ways. First, it can be used similarly to the combo box, in the sense that it is used to select property values from a pre-defined set. The difference to combo box is that while combo box allows only one selected value at time, the listbox allows multiple selections at a time. Second, it can be used to represent a property which may have multiple values defined by the user. An example of this kind of property is the attribute-property of an OOP class.
- *Radio buttons* always exist in groups of two or more. Their functionality is similar to combo box, but they are used when there is only a limited choice of values.
- *Checkboxes* are associated with Boolean values, usually to present whether some feature exists in an object or not.

Figure 3 shows an example of UML class dialog generated with MetaTypePainter. It features the following controls:

- Textbox is used to represent **Class name** property.
- Combo box is used to represent **Concurrency** property.

- List boxes are used to represent **Attributes, Operations** and **Constraints** properties.
- Textfield is used to represent Documentation property.
- Checkboxes are used to represent **Persistent** and **Abstract** properties.

The benefit of MetaEdit+'s dialog system allows the user to ignore the details of dialog creation because these dialogs can be generated automatically straight from the GOPRR element's specifications. The dialog system chooses the controls that are used to represent different properties of element.

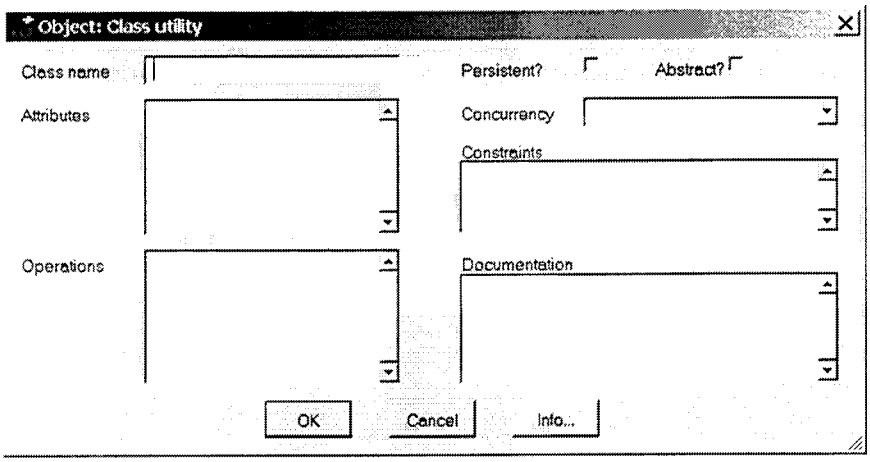

Figure 3: Example of a MetaEdit+ generated property dialog

One possibility is to represent component search using these automatically generated property dialogs. The great benefits of this approach are, naturally, the low amount of extra work in implementing the user interface, and the MetaEdit+'s user's familiarity with these dialogs. If the user can enter the search conditions exactly the same way he or she would enter the descriptions of new components, the problem is probably easier to perceive than with an exclusive search dialog. There are also downsides for this approach. First, we cannot use the dialog without any modifications since we can assume that at least the simplest Boolean operations (AND, OR, NOT) are required for the query. (Frakes & Poole, 1994) We need some intuitive and user-friendly way to combine conditions defined in the dialog with these operations, and this

requires a separate dialog for the component search. However, since these modifications are probably not very significant the search dialog can be inherited from the property dialog to alleviate the task of specifying a search dialog. Naturally, if we choose some more esoteric representation method, we may find the property dialog too limited.

Understandably, when this dialog is used in a search context the requirements for the interaction possibilities differ from the data insertion. When adding objects (e.g. classes, associations, processes) to the project we naturally know very clearly the values we want to use and this does not address any specific requirements for the user interface. However, when the user is forming the query with the dialog the situation is different. The user may know exact values for some of the properties, may know some alternative values for some properties and may have a slight but unclear idea about some values. All these situations require different kinds of querying possibilities from the user interface.

Other possibilities for representing the component model - and models inherited from it - naturally exist, but it must always be noticed that it should be possible to generate a representation automatically from component model definitions. Probably it is a good idea also to provide users with a possibility to create representation manually; in fact, this feature (although very limited) is already implemented in MetaEdit+.

In MetaEdit+ it is very common for a property field to contain another object rather than an atomic value such as an integer. One example of this kind of aggregation is Method property of Class type in UML. Methods may not be presented with atomic types because they contain several properties themselves such as parameters, return value, protection level etc. Defining search criteria for these kinds of objects is naturally more complex. Fortunately, MetaEdit+'s dialog system can manage these situations by opening new subdialogs for each

part of the aggregate object. There is no theoretical limit for the depth of the structures, so these parts may have objects as their properties and so on - without any significant presentation problems. The search conditions for these aggregate objects may be defined using this nested dialog structure, ultimately forming a tree of search conditions. It would be reasonable to place some restriction on how complex this search condition tree may be, e.g. only on two hierarchical levels. Otherwise the search may grow overly complex. This is probably more a theoretical possibility because it would be impractical for users to form very complex queries. This is because defining the query would become a time-consuming process and perceiving the meaning of the query would become too hard. A much more simpler approach is to allow user of search tool select from existing objects, and the query is matched only if the target contains the same object. This approach is much easier to implement and forming the query is easier, but it is also more restricting. Naturally, the search tool should support both methods.

6.1.3 Choosing a Representation Method

Some very rough guidelines on the usability and user satisfaction of different search representations, based of the empirical tests by Frakes and Poole (1994), were presented in Chapter 3. Based on these results, it seems that the most common representation methods – enumerated, faceted and keyword representation – are quite similar in terms of usability and efficiency. Therefore, if we limit the representation possibilities of a search tool to one or some of these methods the choices may be based more on implementation issues rather than user-centred aspects.

Numerous studies (e.g. Prieto-Diaz, 1991, Zhang, 2000, Henninger, 1997) suggest a faceted representation method or its variation as a most suitable

method for component representation. A faceted scheme is easy to maintain and expand, whereas enumerated and keyword approaches both have their downsides (described in Chapter 3) in these areas. In faceted representation, it is easy to add, delete, and update repository components and the facet list in the classification scheme without reclassification - this is difficult for an enumerated classification scheme. (Zhang, 2000) Faceted classification is used successfully in component repositories such as GTE Data Services' Asset Management Program (Prieto-Díaz 1991) and ORCA (Object Reuse Classification Analyser) in ICE (Isakowitz & Kauffman 1996). It is also easy to implement our component model (Zhang, 2000) using a faceted scheme, because the interface facet part of the model is relatively easy to convert into a part of the faceted classification in the models inherited from the RAMSES model.

6.1.4 Faceted Representation and Boolean Queries

In modern information access systems, the matching process usually employs a statistical ranking algorithm or a similar system such as domain-specific heuristics. (Baeza-Yates, 1999) However, in the case of component libraries in which the amount of data is rarely as massive as in text databases the support for Boolean queries may be sufficient for most needs. Boolean syntax can be presented in numerous formats, and it is crucial to present it in an unambiguous way because there are several common conceptual misunderstandings related to Boolean logic. However, we can assume that the metaCASE tool users are somewhat used to Boolean syntax and its concepts. (Baeza-Yates, 1999) One - and probably the simplest and therefore most limited - method to include Boolean operators to the faceted representation is to simply associate a Boolean operator with every facet of the property dialog. If the facet

is not included in the search (that is, its value is not defined), the operator can be simply ignored.

One but rather limited possibility is to represent the operator selection in the user interfaces with a group of radio buttons (selection buttons that are usually grouped, with only one selected at a time). (Baeza-Yates, 1999) These controls present three Boolean operators, and the user must select one of them for each property defined in the search. Alternatively some other interface widget can be used, the only requirement being the ability to select a single item from the list. This associates each defined search condition with a Boolean operator.

Using plain property dialogs to form Boolean queries provides only AND and NOT operations due to the ambiguous nature of the OR operation when multiple conditions are given. Also, more complicated grouping of Boolean conditions (particularly nested conditions) are not possible with the method described because all the conditions are on the same hierarchy level and therefore evaluated in parallel. Therefore we need some alternative graphical (or possibly non-graphical) representation if we want to provide more complex query capabilities to the users. Because the need to use very complex Boolean queries arises probably quite rarely, this alternative Boolean search can be provided as an advanced search feature, for example it may be accessed from the normal search condition dialog. This advanced search tool may represent Boolean conditions as a tree (each branch presenting one nested structure in a Boolean query), or a table, or using a simple high-level query language (naturally, support for multiple representations is possible).

Probably the use of the OR operator in combining different facets (or properties) of the component in search is relatively rare. We may enlighten this by an example: let us assume we have a component model with Domain and Method properties (domain describing the application domain the component is used in, method describing in which method the component is implemented

in). It is quite irrelevant to form a query that states, for example, that domain of the component must be networking OR the method of the component must be UML. Usage of the AND operator to combine different properties, however, is perfectly relevant. The use of the OR operator is probably much more practical when allowing several matching values to single property when forming the query. This is especially true with properties, the values of which are derived from uncontrolled vocabulary (that is, values are entered freely by user) while properties that have a predefined list of possible values probably do not benefit from this capability that much.

How the search conditions should be evaluated depends on the type of the data of the facet (property). With strings, use of wildcards (typically '?' for single character and '*' for arbitrary sequence of characters) should be allowed to make wider queries possible. With numeric data, it should be possible to define a range or to use operators such as > (greater than) and < (smaller than). If the facet contains another object, there are two alternatives: the user may define properties of the inner object or he or she may define that the value of the facet must be exactly the same object.

6.1.5 Alternative Representation Solutions

Alternative solutions for retrieval using faceted representation with Boolean logic have been proposed, mainly forms of visual querying (for example, Liu, 1996; Jones, 1998). The most interesting of these are based on the *direct manipulation approach* –concept that provides an alternative to a complex command line or a dialog-based syntax. Direct manipulation systems include the following common characteristics (Shneiderman, 1997):

- Continuous representation of objects of interest.

- Physical actions or buttons pressed instead of a complex syntax. Therefore, visual representation provides a high-level encapsulation of the underlying retrieval logic.
- Rapid incremental reversible operations, the impact of which on the object of interest is immediately visible.

Direct manipulation interfaces often evoke enthusiasm from users (Baeza-Yates, 1999), and for this reason alone their use is quite an attractive solution and at least worth exploring. Two approaches that use graphical interfaces to simplify specification of Boolean syntax are described below.

Graphical depictions of *Venn diagrams* have been proposed several times as a way to improve Boolean query specification. A query term is associated with a ring or circle and intersections of these rings indicate conjunctions of terms. Typically, a number of components (documents) matching with the various conjuncts are displayed in the appropriate segment of the diagram. Several studies have found such interfaces more effective than their command-language based syntax. (Baeza-Yates, 1999) Venn diagrams have also some major drawbacks such as the limitation to Boolean logic, but some improvements are made to overcome these limitations.

Another innovative direct manipulation interface for Boolean queries is described by Anick et al. (1990) and uses natural language and blocks arranged into columns and rows. The user types the queries in natural language, which is automatically converted into a block representation. If two or more blocks appear along the same row they are considered to form a conjunction (equivalent to an AND operator). Two or more blocks within the same column form a disjunction (equivalent to an OR operator). The benefit of this approach is that "users can quickly experiment with different combinations of terms within Boolean queries simply by activating and deactivating blocks." (Baeza-Yates, 1999)

Although some of these visual approaches are very promising and probably will become more popular in the coming years we will not cover them in depth in this thesis. The semantics of visual query languages are often somewhat confusing, and due to the nature of a metaCASE environment – in which every hard-coded piece should be as generalized as possible – inclusion of exotic approaches should be considered carefully. Although Liu (1996) has proposed and implemented a visual query tool for MetaEdit+ it has not been adopted for practical use.

6.2 Characteristics of Component Retrieval Models in a MetaCASE Environment

The metaCASE environment does not address very unique features to the retrieval models introduced in Chapter 4. However, because the structure of a metaCASE repository is very different from most document databases or relational databases - and ultimately more complicated - the implementation is more complex. According to Liu (1996), this complexity can be observed from three different dimensions: Abstraction, Construction and Representation complexity. Abstraction and construction dimensions are relevant in component search while the representation dimension is more relevant to the basic GOPRR objects. Next, we will discuss abstraction and construction dimensions of the repository and their influence on component retrieval.

6.2.1 Abstraction Complexity in a Retrieval

Abstraction complexity refers to the multiple abstraction levels of a metaCASE repository. In a metaCASE repository, information exists on three levels (Liu, 1996):

- *Model level* holds models of applications created with the tool, typically graphs and other descriptions of the target system. Examples of data on this level are a class diagram of warehouse software, sequence diagram of a bank transaction and a state diagram of a vending machine,
- *Meta model level* contains data describing the types of notations and rules governing how to specify models on the model level. Models on this level usually represent modelling techniques of and methods. Examples of these techniques are a specification of class diagram in UML and a specification of data flow diagram in SA/SD.
- *Meta-metamodel level* is the highest level describing what elements and capabilities are available for creating metamodels on the metamodel level. In MetaEdit+, this abstraction level contains specifications of the GOPRR data model. This level exists exclusively in metaCASE tools while in traditional CASE tools the meta model level is the highest level used.

In addition to these, Liu also mentions an *Operational level* (also *Application level*), which is below the model level and involves data used in daily operations of the applications. This level is not managed in the repository of the MetaEdit+ metaCASE environment, so it is ignored here.

Components exist on two of these levels: model level and metamodel level. Model level components are the most common type of components that are integrated in design (for example, an accounting component for a financial application) while metamodel level components are components consisting of, for example, method fragments or other metalevel information. (Liu, 1996)

Dealing with abstraction complexity in component search should not be made a very challenging task. It should be possible to define, in a query, whether the user wants to search model level components, metamodel level components or both. If the component models differ, this difference should be handled in the search representation phase and a query should be formed on the chosen retrieval model. The implementation of the retrieval process itself is similar in all cases because both the metamodel level and the model level are eventually based on GOPRR elements and therefore MetaEdit+'s repository uses the same underlying data structures to handle both of these levels.

6.2.2 Construction Complexity in a Retrieval

Construction complexity refers to the possibility to construct arbitrarily complex objects. In the simplest form, complex objects are networks of objects bound to each other with GOPRR relationships and roles, and thereby forming larger entities. More complicated objects may be hierarchical structures in which a single object may explode into a graph at a lower hierarchy level. Especially the latter case is a difficult challenge to the retrieval model because hierarchical trees of objects may be colossal, and eventually they may form recursive structures that cannot be handled in traditional manners of a recursively advancing search.

A component interface is one answer to the problems emerging when handling queries in repository containing complex objects. If we restrict the retrieval process purely to the interfaces of components (black-box search) the inner structures and contents of the component do not have to be included in the search. This is quite a radical restriction that assumes that all components in the repository are documented thoroughly and that the interface part of the component model in use is sufficiently extensive that all the necessary aspects of the components become covered. However, as we state in Section 5.2., the

purpose of the component's interface is to provide enough information about the component's functionalities and consequently eliminate the need to investigate its contents.

In our component model any GOPRR element may be an interface element, and therefore possibilities for defining the interface are extensive. However, there may emerge situations where we have to restrict the search tool to include only objects emerging on the first hierarchical level of the interface to the query. Naturally the interface elements should be kept simple to maximize their descriptive power, but we cannot prevent users from creating arbitrarily complex objects. Therefore, some artificial restriction may be necessary. Ideally, this restriction could be defined in the metamodel of the component.

6.3 Component Browsing Discussions

Although we do not want to discuss in detail the user interface design needed to present the search, we would like to address several important issues that should be considered when implementing a search tool in a component repository system. These issues mostly deal with browsing the component candidate list when the preliminary query is conducted, or the whole library if the number of the components in the library is relatively low. The following discussion is applicable to both situations.

It is remarkable that complex query operations are beneficial only if the library is rather large. Users might be willing to invest some time in exploring the component library looking for potential components. In component libraries consisting of few hundreds of components this browsing approach is often much easier than spending time forming often complex queries – which will not necessarily retrieve all the potential components. (Baeza-Yates, 1999, p. 65) In

general, the goal of the searching tasks is clearer in the mind of the user than the goal of a browsing task in which the user might be looking for something less specific or a "close enough" solution. In smaller libraries, developers also typically know the contents of the library quite well and therefore tend to browse the library rather than use a query tool. On the other hand, even in small libraries situations may emerge, in which finding the component with a query is faster than by browsing. This is especially true when the developer or another user of a component repository is aware of the component's literal name, author or any other identifying attribute. Also, it is reasonable to prepare for the possible expansion of the component library. It is elemental for the user interface of the search tool to support both forms of component search equally. (Baeza-Yates, 1999)

The most crucial characteristics or facets of the component should be accessible in a component browsing mode. It should not be necessary for the user to pick up the whole component from the library to view its attributes. One possibility to implement this feature is to divide the browser into two areas with one area showing the list of the candidate components (or initial component list if query is not yet made) and another area showing the attributes of the active or selected component. This view of attributes provides a kind of abstract of the component with most significant information encapsulated in little space. Which information should included in this component "abstract" is up to the users' decisions. Possibly it could be among the issues defined in the customized component model or it could be defined for each component individually.

6.4 Summary

This chapter contained the main contribution of this study. We discussed the practical issues of component search in the GOPRR –based MetaEdit+ metaCASE environment. We did not concentrate on implementation issues such as algorithm choices but approached the problem more from a user-centered perspective. We discussed possibilities of representing component model introduced in Chapter 5 and described different techniques MetaEdit+ uses to represent GOPRR objects. Then we discussed how these techniques might be applied to component search, particularly to component search using faceted representation with Boolean logic. We also discussed some problems that emerge with complex objects and multiple abstraction levels of GOPRR.

In this chapter we tried to present some answers to the research problems. Our coverage on the subject is not exhaustive, but rather described one possible solution to the problem.

7 Conclusion

In small organizations that do not rely very much on component technologies or the component libraries of which are relatively small (no more than a couple of dozens components) the task of finding an appropriate component is mostly a matter of browsing through the component library or utilising former knowledge of the contents of the library. (Frakes & Poole, 1994) In these environments more advanced component retrieval tools do not benefit the library's users very much because defining the right queries is a time-consuming process and it is often more convenient to browse through the small library. However, when the size of the library grows, the process of finding the right components via querying becomes an elemental issue. That is when the concepts and solutions presented in this thesis become relevant and even elemental.

In this thesis, we have presented the basic concepts and the main problem areas concerning the querying techniques concentrating on software components. We have specialized in components in the CASE and the metaCASE environments for two reasons. First, recent studies concerning components mostly concentrate on "binary components" - components that are used in implementation phase rather than design or analysis phases. We think that also the earlier phases of ISD should benefit from components and therefore this area should be studied more thoroughly. Second, a metaCASE environment sets slightly different requirements to the retrieval process itself. This is due to the reason that we do not want to make fixes to the component model (or framework, if speaking in implementation terms) but rather a metamodel that the actual component models are derived from. The retrieval tool implemented in this kind of environment should be able to retrieve all components based on models

inherited from the component's metamodel, and this is one of the basic goals we have tried to achieve in this thesis.

7.1 Previous studies

As far as we know no previous studies on component search in CASE tools have been conducted. Also the studies involving a metaCASE environment in general are relatively scarce. Studies on component search and searching reusable assets have been carried out (Frakes & Gandel, 1989; Henninger, 1997, Mili, Mili & Mili, 1995) but they have concentrated solely on binary components or files. We found only one study where the component search was studied empirically, a study by Frakes and Poole (1994). Information retrieval in general has been a very popular research area, and numerous recent studies on the subject were utilized in this thesis. However, most studies involving advanced search techniques concentrate on text document search and therefore their analogy with our thesis is rather limited. Our study differs from these earlier studies in two main aspects: first, the study focuses on CASE tools instead of tools or development environments used in the implementation phase; second, our study takes a very strict interface-centric approach to the components and therefore leaves out many reusable assets such as OOP classes.

As stated earlier, model components are rarely discussed in academic literature, and model components lack standards both structurally and conceptually. With the latter we mean that the concept of a model component is indefinite and seems to vary greatly between different methods and CASE tools. For these reasons it is practically impossible to propose any guidelines for a model component search that would be applicable to the majority of CASE tools. Therefore after some theoretical discussion we focused on MetaEdit+ metaCASE environment. Due to its origin as an academic project it provides

decent theoretical support for research, and its GOPRR data model is in its completeness and simplicity a good platform for experimental prototypes.

We believe that this study on component search will benefit the further development of MetaEdit+ metaCASE tool as well as shed light on the complex and mostly unexplored area of components and their retrieval in CASE tools. The prototype we implemented can be utilized in MetaEdit+ as is, and although it probably requires some heavy modifications before becoming a part of the commercial product it is an important basis that can be used as a foundation or reference for more advanced search tools.

7.2 Theoretical Background

We covered the theoretical background of the component search in four chapters, each one discussing a particular aspect of the subject. Chapters were divided as follows:

The first of these foundational chapters was dedicated to the basic concepts and issues of component-based software development. These concepts included the definition of software component (drawing a distinction between binary and model components), addressing differences between OOP classes and components, component libraries and requirements for organizational support for reuse of the components.

The second theoretical section discussed component representation methods. This section was largely based on research and empirical studies carried out by Frakes and Poole (1994) as well as numerous other sources. The choice of representation method is one of the most important issues in search because it dominates how the components are represented to the users of a component

library and what kind of queries the users can create for component retrieval. We discussed the most common representation methods derived from library and information sciences: enumerated representation, faceted representation, keyword representation and free text representation. Also, some more esoteric representation methods such as hypertext and AI-based representations were discussed briefly. In the end of the chapter, some very rough guidelines to representing a component library were proposed.

In the third chapter we discussed the possibilities of the internal implementation of the retrieval process. We presented several retrieval models, mostly based on Baeza-Yates (1999), and discussed their suitability to component search. The retrieval models discussed were from the more traditional side of the field (Boolean and vector model) and more esoteric models such as AI-based and fuzzy retrieval models were discussed in less detail. Some suggestions on how to utilize the fuzziness of the retrieval in component search were also given.

In the fourth chapter, concepts involved in a metaCASE environment were introduced. One metameta model, GOPRR, was discussed as well as a generic component model using this model.

These three, mostly theoretical chapters laid down a foundation for the constructive part of the study, which involved providing guidelines for implementing a retrieval tool for MetaEdit+ metaCASE environment. Our view of the search process was mostly derived from Frakes and Gandel (1989) and Frakes and Poole (1994) and therefore there is a considerable possibility that some of our views are somewhat outdated or too narrow. The component search is a complex issue and there are numerous schools suggesting somewhat different perspectives on it. In this study, our goal was to delve deep into the subject from one point and this may reflect as a somewhat two-dimensional approach to the subject matter.

7.3 Constructive Part

The constructive part of this thesis (Chapter 6) consisted of defining requirements and guidelines for a retrieval tool. The objectives of this constructive part were twofold: to introduce a component model metatype to use in a metaCASE environment and to propose requirements for the component retrieval tool for the components inherited from the aforementioned component model. The reference metaCASE environment used in this thesis was MetaCase Consulting's MetaEdit+.

The requirements we have defined are not strict in the sense that they leave many implementation decisions open. A MetaCASE environment is a challenging and unique platform to work with because solutions proposed should be as general as possible. This is because they should be applicable to all metamodels that may be defined with the metameta model of the environment – GOPRR, in case of MetaEdit+. We have tried to propose solutions for how to overcome these challenges and to a build search tool that is simple and flexible yet versatile enough to produce sufficiently accurate query results. However, the lack of concrete implementation solutions, for example search algorithms can be seen as one of the major weaknesses of this thesis. We had to make some tough decisions on our focus between theoretical and practical, and inevitably this will result in some imperfections.

7.4 Subjects for Further Research

When a new piece of software is introduced it should never be based merely on a theoretical discussion. Rather, an empirical study should be carried out before the new software can be accepted as a viable solution. Because this thesis focused only on the theoretical part of the development process, the empirical testing is a logical next step on establishing the search tool proposed here. The empirical testing should utilize several different metamodels and several practical situations to gather an adequate amount of data.

As stated earlier we left most of the implementation decisions open. These include many interesting and fundamental issues such as search algorithms, usability of the search tool and the integration of the search tool. All these issues should be studied carefully before a robust and efficient search tool for managing large component storages may be realized.

References

Allen, P., Frost, S. 1998 Component-Based Development for Enterprise Systems. Cambridge University Press, Cambridge, UK.

Anick, P. Brennan, J., Flynn, R., Hanssen, D., Alvey, B., Robbins, J. 1990. A direct manipulation interface for Boolean information retrieval via natural query language. In W. Bruce Croft, C. J. van Rijsbergen (Eds.) Proceedings of the 13th annual International ACM/SIGIR Conference, October 1990, Brussels, Belgium, 135-150.

Baeza-Yates, R., Ribeiro-Neto, B. 1999. Modern Information Retrieval. Essex: Addison Wesley.

Blair, D.C., Maron, M.E. 1985. An evaluation of Retrieval Effectiveness for a Full-Text Document-Retrieval System, Communications of ACM, 28, 3, 289-299.

Banker, R., Kauffman, R., Zweig, D. 1993. Repository Evaluation of Software Reuse. IEEE Transactions on Software Engineering, 19, 4, 379-389.

Barnes, B., Bollinger, T. 1991. Making Reuse Cost-Effective. IEEE Software. 8, 1, 13-24.

Campbell, G.H. 1999. Adaptable Components. Proceedings of the 21st international conference on software engineering, May 16 - 22, 1999, Los Angeles, CA USA. 685-686.

Damiani, E., Fugini, M.G. 1995. Automatic thesaurus construction supporting fuzzy retrieval of reusable components. Proceedings of the 10th ACM symposium on applied computing, February 26 - 28, 1995, Nashville, TN, USA, 542-547.

D´Souza, D. 1999. Components In a Nutshell, Part 1. Journal of Object-Oriented Programming, 11, 9, 15-18.

Frakes, W., Gandel, P. 1989. Representation Methods For Software Reuse. Conference proceedings on Ada technology in context: application, development, and deployment, October 1989, Pittsburgh, PA, USA, 302-314.

Frakes, W., Gandel, P., Belkin, N., Rutgers, Prieto-Diaz, R. 1989. Panel Session: Information Retrieval and Software Reuse. In Belkin, N. & van Rijsbergen, C. J. (Eds.), Proceedings of the 12th annual international ACM/SIGIR conference on Research and development in information retrieval, June 1989, Cambridge, Massachusetts, USA, 251-256.

Frakes, W., Poole, T. 1994. An Empirical Study of Representation Methods for Reusable Software Components. IEEE Transactions on Software Engineering. 20, 8, 617-630.

Frakes, W., Fox, C. 1995. Sixteen Questions About Software Reuse. Communications of ACM. 38, 6, 75-83.

Frakes, W. 1991. Software Reuse: Is It Delivering? Proceedings of the 13th international conference on software engineering, Austin, Texas, May 1991, 52-59.

Frakes, W., Terry, C. Software Reuse: metrics and models. ACM Computing Surveys, 28, 2, 415-435.

Hänninen, S. K., Jansson, M., Manninen, A., Raunio, A., Äijänen, M. 2000. Survey Report. Internal Report. University of Jyväskylä, Department of Computer Science and Information Systems. (Available at request)

Hänninen & Äijänen. 2000. Komponenttien haku CASE-välineympäristössä. Bachelor's thesis. University of Jyväskylä, Department of Computer Science and Information Systems.

Hadjami, H., Ghezala, B. 1995. A Reuse Approach Based on Object Orientation: Its Contributions in the Development of CASE Tools. M. H. Samadzadeh & Mansour K. Zand (Eds.) Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'95), April 1995, Seattle, WA, USA, 53-62.

Henninger, S. 1997. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. ACM Transactions on Software Engineering and Methodology, 6, 2, 111-140.

Heym, M., Österle, H. 1993. Computer-aided Methodology Engineering. Information and Software Technology, 6/7, 35, 345-354.

Isakowitz, T., Kauffman, R. 1996. Supporting Search for Reusable Objects. IEEE Transactions on Software Engineering. 22, 6, 407-423.

Jeng, J., Cheng, B.H.C. 1993. Using Formal Methods to Construct a Sofware Component Library. In I. Sommerville & M. Paul (Eds.) Proceedings of 4th Software Engineering Conference, 1993, Garmisch-Partenkirchen, Germany, 717, 397-417,. Springer Verlag.

Jones, S. 1998. Graphical query specification and dynamic result previewing for a digital library. Proceedings of 11th Annual Symposium on User Interface Software and Technology, November 1998, San Francisco, California, USA, 143-151.

Karlsson, E.-A., Ed. (1996). Software Reuse: A Holistic Approach, John Wiley & Sons, West Sussex, UK.

Kelly, S., Lyytinen, K., Rossi, M. 1996. MetaEdit+ - A fully configurable multi-user and multi-tool CASE and CAME environment. In P. Constantopoulos, J. Mylopoulos & Y. Vassiliou (Eds.) Advanced Information Systems Engineering, LNCS 1080. Berlin: Springer-Verlag, 1-21.

Kelly, S. 1997. Towards a Comprehensive MetaCASE and CAME Environment. Jyväskylä Studies in Computer Science, Economics and Statistics.

Korhonen, K. 2000. Build Your Own Lego: Components, Frameworks and Processes. Master's Thesis in Computer Science and Information Systems, University of Jyväskylä.

Korhonen, K. (ed.) 2000. RAMSES: Requirements for component functionality in MetaEdit+. Working Paper. University of Jyväskylä. Department of Computer Science and Information Systems.

Koskinen, M. 2000. Process Metamodelling: Conceptual Foundations and Application. Jyväskylä University Studies in Computing 7, University of Jyväskylä.

Lending, D., Chervany, N. 1998. The Use of CASE Tools. R. Agarwal, (Ed.) Proceedings of the 1998 ACM SIGCPR Conference, March 1998, Boston, Massachusetts, USA, ACM Press, NY, 49-58.

Liu, H. 1996. On a Visual Approach of Querying the CASE Repository. Computer Science and Information Systems Reports TR-16. University of Jyväskylä, Jyväskylä.

Maarek, Y.S., Berry, D.M., Kaiser, G. 1991. An Information Retrieval Approach for Automatically Constructing Software Libraries. IEEE Transactions on Software Engineering, 17, 8, 1991, pp. 800-813.

Mili, A., Mili, R., Mittermeir, R. 1997. Storing and Retrieving Software Components: A Refinement Based System. IEEE Transactions on Software Engineering. 23, 7.

Mili, H., Mili, F., Mili, A. 1995. Reusing Software: Issues and Research Directions. IEEE Transactions on Software Engineering. 21, 6, 528-561.

Nunamaker, J.F. jr., Chen, M., & Purdin, T.D.M. 1991. Systems development in Information Systems Research, Journal of Management Information Systems, 7, 3, 89-106.

OMG Unified Modelling Language Specification version 1.3. 1999. Available at <http://www.rational.com/uml/resources/documentation/index.jsp>

Persson, E. 1998. The quest for the software chip. The roots of software components. A study and some speculations. In J. Bosch, G. Hedin, K. Koskimies, B. Bruun Kristensen (Eds.) Proceedings of the First Nordic

Workshop on Software Architecture (NOSA'98), August 1998, Ronneby, Sweden.

Prieto-Diaz, R. 1991. Implementing Faceted Classification for Software Reuse. Communications of the ACM. 34, 5, 88-99.

Rine, D.C., Nada, N. 2000. An Empirical Study of a Software Reuse Reference Model. Information and Software Technology, 42, 1, 47-66.

Salton, G., Yang, C.S., Yu, C.T. 1983. Extended Boolean information Retrieval. Communications of the ACM, 26, 11, 1022-1036.

Salton G., Buckley, C. 1988. Term-weighting approaches in automatic retrieval. Information Processing & Management, 24, 5, 513-523.

Shneiderman, B. 1997. Designing the User Interface: Strategies for Effective Human-computer Interaction. Addison-Wesley, Reading, MA.

Smolander, K. 1993. GOPRR: a proposal for a metal level model. (unpublished working paper) University of Jyväskylä, Department of Computer Science and Information Systems.

Tracz, W. 1991. A Conceptual Model for Megaprogramming. Software Engineering Notes, 16, 3, 36-45.

Zhang, Z. 2000. Defining components in a MetaCASE environment. Proceedings of the 12th Conference on Advanced Information Systems Engineering (CAiSE*00), June 2000, Stockholm, Sweden.

Zhang, Z. 2000. Enhancing Component Reuse Using Search Techniques. Svenson, et al. (Eds.) Proc. 23rd conference on Information System Research in Scandinavia (IRIS23), August, 2000, Lingatan, Sweden.

Zhang, Z. & Lyytinen, K. 2000. A framework for component reuse in a metaCASE based software development. S. Brinkkemper, E. Lindencrona & A. Solvberg (Eds.), Systems Engineering: State of the Art and Research Themes, June 2000, Stockholm, Sweden.

Zhang, Z., Rossi, M. 2000. Improved Component Structure for System Analysis and Design. Working Paper. University of Jyväskylä, Department of Computer Science and Information Systems.