Components Analysis in the Metamodelling Based Information System Development

Zheying Zhang

Department of Computer Science and Information Systems
University of Jyväskylä, PL 35, FIN-40351 Jyväskylä, Finland
zhezhan@cc.jyu.fi

Abstract

Component-based technology has emerged as a key element in the development of information systems. More and more systems are designed and developed from components instead of from scratch. This paper presents a component taxonomy in the metamodelling based systems development environments, called metaCASE environments. It elaborates on the aspects of structure, functionality, supporting environment, and reusability and provides comparisons between a code component, a model component, and a metamodel component. Through comparison it presents the current state of component based development in metaCASE environments, and reveals challenges and research directions in further research of component based metaCASE environments.

1 Introduction

Component-based technology has emerged as a key element in the development of complex information systems. It was initiated by the object-oriented paradigm, inspired by distributed object technology, and eventually formed a way of packaging object implementations to ease their use [Cheesman and Daniels 2000]. Recently, the growing popularity and availability of component-based technologies is fueling a change in the habits and expectations of software engineers. Many systems development tools and techniques have regarded components as the key to reuse and for building systems effectively and efficiently [Jacobson et al., 1997].

As an application of component technology, component based development (CBD) represents an important stage in the maturation of the field of information system development (ISD). It shifts the focus from new system development to the integration of existing components to perform new tasks. At the same time, CASE tools and metamodelling techniques enhance the system analysis and design process by providing facilities to define and deploy domain-specific visual languages. It helps advance the component concept during the earlier stages of ISD. Besides software components, components for requirements definition, solution specification, or testing suites exist. The information system community is starting to concentrate attention on the diverse types of components.

Our goal is to drive the diffusion of component technology into the ISD lifecycle. In this paper we illustrate, analyze and compare possible types of components used in the system development environments, especially in the metaCASE environments which provide the metamodelling language, or visual languages which are used for system analysis and design. Although the research and development of component technology in some phases of ISD is still at the exploration stage, our discussion will

expose all possibilities that the component technology can be applied in theory. It presents current research work and raises questions for future research in the component based metaCASE environments.

The paper is organized as follows: Section 2 briefly discusses the nature of components and presents three types of components used in a metaCASE environment in terms of information abstraction levels: code component, model component and metamodel component. Based on a concrete metaCASE environment - MetaEdit+, Section 3 compares and analyzes three types of components from the perspectives of structure, functionality, supporting environment, and reusability. Section 4 summaries the discussions and points out future research directions.

2 Component taxonomy

What constitutes a component? It is an obvious starting point when a component concept is introduced into ISD. Many initial ideas of CBD come from structured methods with modular design techniques, reuse-oriented approaches, and object-oriented analyses and design methods. In general, [Szyperski 1998] defines a software component as a unit of composition with contractually specified interfaces and explicit context dependencies. Components are reusable by third parties.

Most often, a single component is not useful alone but is reused in conjunction with many other components. Such related components are organized together into component systems. A component system can range in size from only a few components to a large number of components, developed by a team of engineers. At the same time, a component based development environment provides facilities to develop different component system projects. It possesses a large number of components from the requirements analysis to the final implementation and system testing. In order to systematically manage and reuse components, it is necessary to classify components into different groups.

There are different types of components involved in the process of ISD, depending on the underlying philosophies that classify the components. For example, [Jacobson et al., 1997] take into account the stages of ISD and distinguish use case components from object components. The use case components are primarily used to capture the system requirements and express what the system should do, and include use cases, actors, and glossary variants; while the object (analysis and design) components are primarily used the express system structure and design at some levels of abstraction, and include all the models or software artifacts from the analysis stage to the implementation and test stage.

Additionally, [Brown and Short, 1998] examine the engineering activities involved in assembling component systems, and divide components into 5 categories: off-the-shelf component, qualified component, adapted component, assembled component, and updated component. Moreover, in the context of the system development lifecycle, we will classify components into requirement components that capture the system requirements, analysis components which shape the system architecture, design components that sort out the solutions, code components that implement the solutions, and test components that validate the systems; in the context of an

application domain, we can distinguish the components used in each specific application domain.

Although numerous component taxonomies exist, none of them classify components as a whole. Some taxonomies focus on the limited viewpoints for a specific purpose. For example, [Jacobson et al., 1997] emphasize the importance of the requirements specification stage, and identify use case components, although the object component is a big group that consists of components at the analysis, design, implementation, and test stages. Also, some taxonomies are subject to component usability evaluation and cannot help users to locate a component that provides specific services, e.g. the taxonomy based on the engineering activities [Brown and Short, 1998]. Therefore, before selecting taxonomy to classify components, we have to first consider the purpose in using the taxonomy.

In the visual language based system development environment, especially in the metaCASE environment, components are deployed not only in the ISD lifecycle, but also in the process of metamodeling methods (visual languages) to support ISD. Both ISD and the methods to support ISD are two important aspects studied by the information systems community [Lyytinen, 1987]. When analyzing components in such an environment, we have to take into account both engineering processes and classify components into three categories: code component, model component, and metamodel component. The discussion domains and the relationship between each component category are shown in Figure 1.

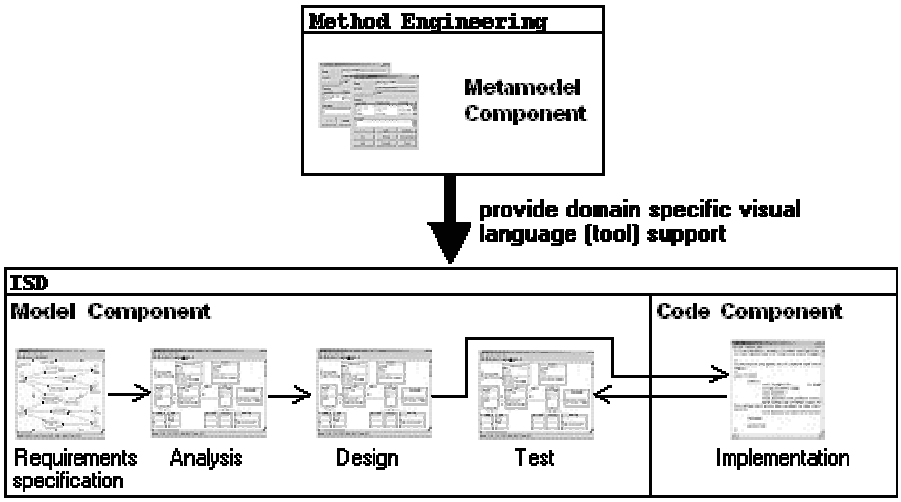

Figure 1 Component taxonomy in a metaCASE environment

2.1 Code component

A code component, or a so-called software component, is widely used in system implementation. Initially, it was viewed almost exclusively as a source code module. In recent years, the popular use of the term component has been with reference to binary code [Parrish et al., 2001]. Therefore, from the software perspective the notion

of a component has a wide interpretation. Examples are pieces of programming logic stored in libraries, executable code deployable on different target systems, and functional units performing business tasks. Each of these has similar characteristics: it provides a defined set of services and can be combined with other classes or components to build applications. In general, they have no associated analysis and design documentation. A code component is the most popular form of reuse in the field of software engineering because its effects on productivity are immediate.

In order to promote the use of code components, software and platform vendors have been touting their implementations of component frameworks. Since 1992, Microsoft's Visual Basic and its components have successfully created a substantial market [Eddon and Eddon, 1998]. In the enterprise arena, OMG's CORBA 2.0 followed in mid-1995. The growing popularity of distribution and the Internet led to new developments, including Microsoft's DCOM and ActiveX, and Sun's JavaBeans.

2.2 Model component

It is worth noting, however, that a component is not necessarily synonymous with executable software. It has been widely known that models generated at the system analysis and design stages can also be wrapped as components.

When building several related information systems, we create several different models. As we do this, we can reuse requirements specifications, use cases, classes, states, processes, and other facilities such as interfaces and test suites, rather than developing each model from scratch. However, we cannot just reuse any design artifact. Instead, the design artifacts intended for reuse must be elaborately designed and wrapped specially to be reusable components. Most model components are models constructed using visual modeling languages such as UML, rather than source code, templates, and executables. Similarly, a model component may be completely defined by a specification, documentation deliverables, or a model at varying levels of abstraction and of different size from which code can be generated.

However, current CASE tools normally take design artifacts as the design documentation associated to the code components rather than independent model components, which may hamper the diffusion of reuse activity at the early stage of ISD.

2.3 Metamodel component

Differently, metamodel component is not involved in the lifecycle of ISD, although it underlies the system design in a metaCASE environment. Metamodel components are the metamodels wrapped by interface description. They are conceptual models of a system development method to specify and construct methods from the perspectives of concepts, properties, rules, and generators. In a metaCASE environment, metamodels can be instantiated to represent the requirements and solutions of the application domains in forms of models or codes.

As one part of the component technology, the metamodel component is still an untouched research area, due to its limited application domains. However, with the increasing attention paid to component reuse during the system analysis and design stage, metamodel components should receive more attention in future research of component-based development in the metaCASE environment.

3 Component analysis

In this section, we will focus on component usability to analyze and distinguish between different types of components in the metamodelling based system development environment. Our discussion is presented from the perspectives of component structure, using environment, functionality, and reusability. In order to have a more vivid discussion, our comparison is based on a real environment, MetaEdit+, an industrial strength metaCASE environment for domain specific visual language based ISD. It is worth noting that although the features of components are discussed in the concrete environment, all comparison and discussion will be generic, and thereby can be applied to other metaCASE environments.

3.1 MetaEdit+

MetaEdit+ is a customizable CASE environment that supports both CASE and metaCASE functionality for multiple users within the same environment. It supports and integrates multiple methods using form-based tools and includes multiple editing tools for diagrams, matrices and tables which are used at the system design stage. It was developed in the MetaPHOR project, which had earlier developed the single user MetaEdit metaCASE tool [Kelly, et al., 1996]. The current research goal is to apply component definition at the system design stage [Zhang, 2000]. A prototype has been constructed and the feasibility evaluation of the component structure is being conducted [Korhonen et al., 2000].

3.2 Component analysis in MetaEdit+

The major difference between the three types of components is that they are composed from three different levels of information abstraction: the metamodelling level, where the domain specific visual languages are defined; the modeling level, where the requirements analysis and solutions are schemed out by using visual language; and the code level, where the expected services are programmed in the form of code or an executable binary segment. Due to their different information abstraction levels, the component structure, functionality, and reusability are distinct. Table 1 presents a schematic outline of the comparison. The detailed analysis is presented in the following sections. Since our discussion reflects the component use in MetaEdit+, all terms used here conform to the previous research papers [Lyytinen and Zhang 2000; Zhang 2000; Zhang and Lyytinen 2001].

		Metamodel component	Model component	Code component
Structure	**Component interface**	-	Faceted description	IDL (interface description language) or
	Representation	Form based specification	Diverse forms of representation, but mainly graph based	Code or binary executable segment
	Granularity	-	Graph level or component unit level	Graph level or project level
Functionality	**Input**	Specification of methodology (text-based)	Description of problems and requirements in a specific domain	Description of solution
	Output	Modeling languages/tools	Requirements and Solution specification	Services
Environment	**Domain**	Method engineering	System analysis and design, system test	System implementation
	Supporting tool	MetaCASE tools	CASE tools	CASE tools or CBS supporting environment
	User	Method engineer	Requirements analyzer system designer, system tester	Software engineer (analyzer, designer, and programmer)
Reusability	**Techniques to reuse**	White box	White box	White/gray/black box
	Reuse type	Conceptual /Functional /instantiation reuse	Functional/conceptual /instantiation reuse	Functional/conceptual reuse
Others	**Portability**	Less	To some extent	Most
	Maturity	Not yet	To be noticed	To some extent

Table 1 Comparison of three types of components in metamodelling based ISD

3.2.1 Structure

As a component, its service or functionality is provided through its interface. The interface is the most important part in the component structure. It summarizes services the component supports and describes how a client should interact with a component, but still hides underlying details. Thereby, the general information provided by a component interface is similar; however, to different types of components, the interface descriptions are distinct depending on the information abstraction level it represents and the supporting infrastructure.

The interface definitions of code components are diverse, depending on the implementation languages and the infrastructure in which they are involved. For example, in JavaBeans, there is a standard way to allow the user of a Bean to find out what operations it supports and how to call those operations. It is language dependent and simply provides some mechanisms to extract the interface description from the component, which allows applications to be portable across any machines supporting the Java VM. However, in CORBA, there is an Interface Description Language (IDL) to describe the component interface, which works as the basis for brokering connections among applications written in different languages running on different machines. It is thereby language and platform independent.

According to the reuse framework presented in [Zhang and Lyytinen, 2001], the component granularity is on the graph or project level, which means that the component provides a complete service independently. There are no code components on the unit level. Because the interface provides all the information a client can depend on, permitting the designer of the client to be unaware of implementations, it is unnecessary and impractical to decompose a code component into several smaller parts.

For model components, the contents are implemented by using the visual language. The interface structure and presentation is distinct in different CASE tools. In MetaEdit+, for example, the model can be represented in forms of a diagram, table, and matrix. The interfaces are specified based on a faceted schema, which is predefined by the method engineers and extensible by the component users. Each interface is composed of the port which is the "access point" through which the component interacts with the external, and the interface facet which defines the behavior of a port in terms of domain, role, interaction, pre-condition, and post-condition [Zhang and Rossi, 2001]. In other CASE tools, such as Rational Rose, the component interface is represented as a lollipop which is specified by an interface stereotype class. Due to the difference of underlying concepts and semantics provided by the supporting tools, components defined in Rational Rose cannot be reused in MetaEdit+, and vice versa. Generally, the granularity of a reusable model component is at the graph level. Sometimes, a unit level component, like a single class, can also be taken as the component and reused across projects.

So far, metamodel components are not that widely used. Current metaCASE tools seldom provide support for components in method engineering, and there is no well-defined structure for metamodel components as yet. In MetaEdit+, the metamodel of method concepts, their relationships, and associated rules are specified in their own specification tools and can be reused, but they are not wrapped as components.

3.2.2 Supporting environment and functionality

All components analyzed here are limited to the metaCASE environment, which includes both a metamodelling process for method specification and a modeling process for ISD. They are employed at different stages of ISD and play distinct roles, due to their distinct information abstract level.

Code components are specified to provide some services. A survey by [Zhang, 2001] shows that they are generally supported in CASE tools such as MetaEdit+, ObjectiF, Paradigm Plus, and Rose98. In these tools, their contents are generated automatically depending on the model component specification, and finally adjusted and implemented by programmers.

Model components are generated at the system analysis and design stages. They form a common reference frame for the parties involved in the development project and reflect part of the problem analysis, requirements determination process, and solutions [Solvberg and Kung, 1993]. Because CASE tools provide visual languages to present the conceptual models graphically, most of the model components are represented as diagrams specifying the system static structure, input-output transformations, and state changes over time. Meanwhile, model components can also specify the

procedures, stages, and data for system testing in forms of table, matrix, or structured text.

Metamodel components provide specifications of methods. In MetaEdit+, the well-defined metamodels form the visual language/tool to build model components in the later ISD projects. Because metamodels are a sort of high-level conceptual model of methods that are specified by the underlying conceptual data model provided by a specific metamodelling environment, their use is limited to the specific environment, which impedes their diffusion in information systems community.

3.2.3 Reusability

Component technology shifts our focus from new system development to the reuse and integration of existing component to perform new tasks. Reuse is one of the straightforward benefits gained from it. However, improving component reusability is a long-term goal, depending on the maturity of the component marketplace.

Code component reuse, so-called software reuse, has been researched since the emergence of the software crisis. Different software reuse methods have been formed. They are white box, gray box and black box reuse in terms of possibility to access and modify the component implementation [Ezran et al., 1998]. If users can find the exact code components required, the black box technique is applied during the reuse process; otherwise, they have to use the white/gray box technique to investigate and modify the details of the component implementation. Different from the code component users, users of model components and metamodel components are concerned more about the processes, transitions, and rules specified inside the components, and require the component content to be visible during the reuse process. Thereby, the white box technique is the main approach to reuse model components or metamodel components.

In the reuse framework presented by [Zhang and Lyytinen, 2001], there are three types of reuse: conceptual reuse, functional reuse, and instantiation reuse. Conceptual reuse, also known as horizontal reuse [Ezran et al., 1998], captures and thus “pipes” the domain knowledge through different layers of representations that are required to develop an application by starting from high level requirements. Functional reuse, also known as vertical reuse [Ezran et al., 1998], refers to the situations that exploit functional similarities across two or more application domains. Both types of reuse are general approaches in the reuse activity and can be applied to all types of components. However, instantiation reuse is subject to the reuse process that takes place in the metamodelling based system analysis and design process. The instantiation reuse exploits semantic similarities across the metamodel components and their instances, model components, to formalize garnered design knowledge for reuse. Thereby, it takes place between metamodel components and model components. Meanwhile, only experienced users who are acquainted with the metamodelling and the instantiation processes can accomplish the instantiation reuse process.

Current CASE tools that support component technology generally support code component reuse by providing reverse engineering functions for the components, classes, and class library. However, most of them ignore the reuse in the component based analysis and design process [Zhang, 2001]. To our knowledge, no previous

empirical studies of system design have specifically considered issues for component reuse, beyond the issue of reusing domain experience. Neither have they considered the reuse issue at the more abstract metamodel component level. For this study, we, thereby, sought to identify a suitable set of reusable metamodels that would have sufficient complexity to require some degree of abstraction in thinking about their reuse, and yet would be readily available for our purpose to develop component based metaCASE environment.

3.2.4 Others

Components are typically useful only within the context of a larger, unifying framework that provides structure and semantics. The deployment of components is limited to a specific metaCASE environment, due to its underlying semantics and rules to define and apply components. Although the category and principles discussed are general to any sort of metaCASE environments, it is difficult to use components across platforms, except code components. Therefore, if we evaluate the different types of component in term of portability, the code component is the most portable type. It can be applied in different environments. The model components have to be applied within a specific metaCASE environment, but can be used between different development projects or different application domains. The metamodel component may have the most limit of its portability due to its abstract feature to specify the concepts and rules of methodologies using the metamodelling language provided by the specific metaCASE tool.

4 Discussion and conclusions

Overall, we have sought to study and categorize components involved in the metamodelling based ISD by comparing them from the perspectives of structure, functionality, environment and reuse. The analysis and comparisons are summarized as follows:

- Code components are relatively developed and widely used not only in CASE tools, but also in environments that support component technology. Many CASE tools provide the reverse engineering functionality to support its reuse. Meanwhile, code components can be reused across platforms depending on the nature of the implementation and their interface specification.

- In order to expand benefits of reuse at the earlier stages of ISD, model components are concerned more and more by the information system communities. CASE tools start supporting the component technology at the system analysis and design stages. However, the deployment of model components is not as flexible as code components. They are generally limited to a specific CASE tool, but can be reused across projects.

- Research of metamodel components is still premature. Many metaCASE tools, like MetaEdit+, start to seek for the possibility to introduce the concept of components and the feasibility to convert or wrap metamodels up as components [Korhonen, Lyytinen et al. 2000; Zhang 2000]. However, how to define a metamodel component is still unclear. Because the conceptual specification of metamodels that later support ISD is very abstract, it is neither like a model component which is composed of the concrete descriptions of problems or

solutions based upon the real situations, nor like a code component which can be reused in a black box method through its concise interface description. It is difficult to define a metamodel interface description to specify its semantics and rules inside. When we try to reuse the metamodel component, method engineers have to dig into every detail about its abstract specifications. As a result, the use of metamodel components can be hardly recognized and accepted widely, unless there is a good solution for extracting information easily from the abstract metamodel specification. In other words, we need a mechanism to create context awareness of the component.

As such, we consider that our main results are, therefore, mostly in the nature of preliminary observations about the component technology applied in the metamodelling based ISD process, leading to new insight and new questions, as well as also creating the basis for more controlled component structure design in a metaCASE environment. There are obviously many other questions to be addressed in seeking to extend any conclusions from this study to the problems of reuse of components on a larger scale, especially in the context of knowledge presentation and understanding, and we plan to address some of these questions in future work.

While the results of this study show some interesting trends in the field of component technology, as indicated earlier, some research topics are still at the exploratory stage. However, we believe that the questions we have investigated are of considerable importance in the context of current trends in component technology, and, hence, merit further and wider investigation.

References

Brown, A. W. and K. Short (1998). On Components and Objects: The Foundations of Component-Based Software Development, Sterling Software.

Cheesman, J. and J. Daniels (2000). UML Components: a simple process for specifying component-based software, Addison-Wesley.

D'Souza, D. and A. Wills (1998). Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison Wesley.

Eddon, G. and H. Eddon (1998). Programming Components with Microsoft Visual Basic 6.0, Microsoft Press.

Ezran, M., M. Morisio, et al. (1998). Practical Software Reuse: the essential guide.

Jacobson, I., M. Griss, et al. (1997). Software reuse: architecture process and organization for business success. ACM Press, New York, Addison-Wesley.

Korhonen, K., K. Lyytinen, et al. (2000). What RAMSES is all about? Department of Computer Science and Information Systems, University of Jyväskylä.

Lyytinen, K. (1987). A Taxonomic Perspective of Information Systems Development: Theoretical Constructs and Recommendations. Critical Issues in Information Systems Research. R. J. B. Jr. and R. A. Hirschheim, John Wiley & Sons Ltd.: 3 - 41.

Lyytinen, K. and Z. Zhang (2000). A framework for component reuse in a metaCASE based software development. Information Systems Engineering: State of the Art and Research Themes. S. Brinkkemper, E. Lindencrona and A. Solvberg, Springer: 107 - 122.

Parrish, A., B. Dixon, et al. (2001). "A Conceptual Foundation for Component-Based Software Deployment." Journal of Systems and Software (to appear).

Sitaraman, M. and B. Weide (1994). "Special Feature: Component-Based Software Using RESOLVE." Software Engineering Notes 19(4): 21 - 67.

Solvberg, A. and D. C. Kung (1993). Information Systems Engineering. Berlin, Springer-Verlag.

Szyperski, C. (1998). Component Software: Beyond Object-Oriented Programming, Addison-Wesley.

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

Zhang, Z. (2001). Component-based Reuse in a MetaCASE Environment. Department of CS and IS. Jyväskylä, University of Jyväskylä: 121.

Zhang, Z. and K. Lyytinen (2001). "A Framework for Component Reuse in a Metamodelling based Software Development." Requirements Engineering Journal 6(2): 116 - 131.

Zhang, Z. and M. Rossi (2001). Using components for system analysis and design (working paper)