JYVÄSKYLÄ STUDIES IN COMPUTING 39

Zheying Zhang

Model Component Reuse

Conceptual Foundations and Application in the Metamodeling-Based Systems Analysis and Design Environment

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella
julkisesti tarkastettavaksi Agora-rakennuksessa (Ag Aud. 3)
lokakuun 8. päivänä 2004 kello 14.30

Academic dissertation to be publicly discussed, by permission of
the the Faculty of Information Technology of the University of Jyväskylä,
in the Building Agora, Ag Aud. 3, on October 8, 2004 at 2.30 pm.

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2004

Model Component Reuse

Conceptual Foundations and Application in the Metamodeling-Based Systems Analysis and Design Environment

JYVÄSKYLÄ STUDIES IN COMPUTING 39

Zheying Zhang

Model Component Reuse

Conceptual Foundations and Application in the Metamodeling-Based Systems Analysis and Design Environment

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2004

Editors
Seppo Puuronen
Department of Computer Science and Information Systems, University of Jyväskylä
Pekka Olsbo, Marja-Leena Tynkkynen
Publishing Unit, University Library of Jyväskylä

URN:ISBN 9513919196
ISBN 951-39-1919-6 (PDF)

ISBN 951-39-1819-X (nid.)
ISSN 1456-5390

Jyväskylä University Printing House, Jyväskylä
and ER-Paino Ky, Lievestuore 2004

ABSTRACT

Zhang, Zheying
Model Component Reuse: Conceptual Foundations and Application in the Metamodeling-Based Systems Analysis and Design Environment
Jyväskylä: University of Jyväskylä, 2004, 76 p.
(Jyväskylä Studies in Computing
ISSN 1456-5390; 39)
ISBN 951-39-1919-6
Finnish Summary
Diss.

Component reuse is an emerging development paradigm that promises to accelerate information systems development (ISD) and to reduce costs by assembling systems from prefabricated components. Defining, designing, developing and deploying reusable component, however, is a complex process, which sets high requirements not only on methodical support for component structure and functionality, but also on the supporting development environment. The current component-based development approach and its supporting environment, however, lack the mechanism and functionality to support component reuse at every stage of the ISD process. The main objective of this thesis is to elaborate a theory, and hence strategies, that can systematically support component reuse in a metamodeling-based systems development environment – a metaCASE environment - which offers a great deal of potential in terms of software productivity and quality. The research reported here describes the characteristics of a metaCASE environment, develops a conceptual framework for different types of reuse of components on different granularities at different levels of ISD abstraction, builds the component model for a metaCASE environment, suggests strategies for component reuse, and empirically studies the impact of component deployment in systems analysis and design in MetaEdit+, an industry-strength metaCASE environment. The research follows a constructive research paradigm. A component-based reuse framework and a component model are designed to answer the needs of support for reuse in a metaCASE environment, which are further implemented and tested in MetaEdit+. In sum, component-based reuse in a metaCASE environment is a new research area and still in its infancy. The main contribution of this thesis is twofold: it offers conceptual frameworks which comprehensively depict the component model and its systematic reuse processes in a metaCASE environment, and an experimental design for quantitatively investigating the impact of component reuse in systems analysis and design.

Keywords: component reuse, component-based development, metaCASE environments, component model, component interface, component context

ACM Computing Review Categories

D.2.1 Software Engineering: Requirements/Specifications: *Languages, Methodologies, Tools*
D.2.2 Software Engineering: Design Tools and Techniques: *Computer-aided software engineering (CASE)*
D.2.10 Software Engineering: Design: *Methodologies*
D.2.13 Software Engineering: Reusable Software: *Domain engineering, Reusable libraries, Reuse models*

Author's address Zheying Zhang
Department of Computer Sciences
FIN-33014 University of Tampere
Finland
Email: Zheying.Zhang@cs.uta.fi

Supervisor Professor Kalle Lyytinen
Iris S. Wolstein Chair
Department of Information Systems
The Weatherhead School of Management
Case Western Reserve University
Cleveland, Ohio 44106-7235, USA
Email: kalle@po.cwru.edu

Reviewers Professor Sandeep Purao
School of Information Sciences and Technology
The Pennsylvania State University
State College, PA 16802, USA

Professor Jukka Paakki
Department of Computer Science
University of Helsinki
P.O.Box 26 (Teollisuuskatu 23), FIN-00014, Finland

Opponent Professor Pericles Loucopoulos
Department of Computation
University of Manchester Institute of Science and Technology
P.O. Box 88, Manchester, M60 1QD, United Kingdom

ACKNOWLEDGEMENTS

This research was supported by COMAS Graduate School, the INFWEST.IT program, the ITEA-funded Café project of the EU Eureka 2023 Programme, and the Department of Computer Science and Information Systems at the University of Jyväskylä. I extend my sincere gratitude and appreciation to the many people and organizations who made this thesis possible.

First and foremost, I am deeply indebted to my esteemed supervisor, Prof. Kalle Lyytinen, whose philosophical insights, continual support, and incomparable patience guided me throughout my study. I thank him for the opportunity to work in the MetaPHOR research group, and for his many constructive comments on my work. He was always available when I needed his advice. I especially appreciate the motivating, enthusiastic, and critical atmosphere he provided during the many discussions we had. It was my great pleasure to conduct this research under his supervision.

I would like to record my gratitude to the external reviewers of my thesis: Prof. Sandeep Purao from Pennsylvania State University, and Prof. Jukka Paakki from the University of Helsinki, and all the unknown reviewers of the papers. Their constructive comments and suggestions have helped me to improve my work.

My former colleagues from the MetaPHOR research group have supported me throughout my research work. The friendship of Janne Kaipala is much appreciated and has led to many interesting and amiable discussions relating to this research. I am also grateful to Matti Rossi, Steve Kelly, Juha-Pekka Tolvanen, Minna Koskinen, Jouni Huotari, Risto Pohjonen, and Pentti Marttiin who have all contributed with valuable comments and by creating favorable conditions for research. In particular, Matti, Steve, and Juha-Pekka introduced me into the exciting world of metamodelling, method engineering, and meta-CASE. Special thanks go to the other members of the research group, Kalle Korhonen, Matti Äijänen, and Mirja Pulkkinen, the students from the COMBO project, and the many people who participated in the laboratory experiment, for their support in the thesis work.

I have spent an unforgettable time in the Software Business Program. I wish to extend my thanks to Prof. Jukka Heikkilä (Jups), Prof. Timo Käkölä, Dr. Nazmun Nahar, Kai Vuolajärvi, Rauli Käppi, Anicet Yalaho, Jonna Kalermo, and other staff members who brought me to a broader but wonderful research area. I appreciate the fact that I have been able to work with you and thank you for your close friendship, valuable help, and joyful collaboration. My thanks go to Jups for helping with various aspects of the thesis finalization. I also own special thanks to Timo, who introduced me to the Software Business Program, and supported me in my research and teaching work.

I thank the lecturers, professors, and all my colleagues in the Department of Computer Science and Information Systems, in particular Prof. Seppo Puuronen, Dr. Samuli Pekkola, Dr. Eleni Berki, Dr. Marketta Niemelä, Prof. Jari

Veijalainen, Prof. Heikki Saastamoinen, Jari Rahikainen, Eija Ihanainen, Tapio Tammi, and other staff members and friends, for their valuable help, advice, encouragement, and patience. I am also grateful to Michael Freeman for his careful and patient proof-reading of my thesis.

I wish to extend special thanks to my parents, Chunfeng and Zhili, and to my parents-in-law, Kewei and Qiyun, for their love and unwavering support throughout my studies, and to my sister Lulu for providing me with various non-thesis-related problems to distract me. Lots of thanks go to my uncle's family, Zhiqi, Aiping, Chi, and He, who encouraged me to come to Finland and supplied me with endless wisdom and encouragement when I was confused and hesitant on this journey.

At last, but certainly not least, I would like to thank my husband, Yu, for his infinite wisdom, support, and encouragement. One of the best experiences that we lived through during this period was the birth of our son Jieming, who brought an additional and joyful dimension to our life mission.

Thank you, all.

Tampere
September 2004

FIGURES

FIGURE 1 Metamodel of the background concepts 16
FIGURE 2 Background of the study..... 17
FIGURE 3 Metamodeling and modeling in a metaCASE environment..... 24
FIGURE 4 CBD: moving from domain idea to finished product 36
FIGURE 5 DSM: moving from domain idea to finished product..... 40
FIGURE 6 Relationship between research questions and the research background..... 50
FIGURE 7 A multi-methodological approach to IS research 52
FIGURE 8 Contribution of each article..... 55

TABLES

TABLE 1 Comparison between reuse techniques..... 35
TABLE 2 Research questions and their treatment 55

LIST OF INCLUDED ARTICLES

Zhang, Z. & Lyytinen, K. 2001. A Framework for Component Reuse in a Metamodeling Based Software Development. Requirements Engineering Journal 6 (2), 116 – 131.

Zhang, Z. 2000. Defining Components in a MetaCASE Environment. In B. Wangler and L. Bergman (Eds.) Advanced Information Systems Engineering: 12th International Conference, CAiSE 2000, Stockholm, Sweden, June 2000, LNCS 1789, Heidelberg: Springer-Verlag, 340 –354.

Zhang, Z. & Rossi, M. 2002. Component Modeling for Systems Analysis and Design. ICSR7 2002 Workshop on Component-based Software Development Processes, April 15-19, 2002, Austin, Texas, USA.

Zhang, Z. & Kaipala, J. 2004. Component Context Specification and Representation in a MetaCASE Environment. To be submitted to Information and Software Technology for possible publication.
An early version was published in M. Khosrow-Pour (Ed.) Information Technology and Organizations: Trends, Issues, Challenges and Solutions, Proceedings of the 2003 Information Resources Management Association International Conference (IRMA2003), Philadelphia, PA, USA, May 18-21, 2003, Hershey, PA: Idea Group Publishing, 712 –715.

Zhang, Z. 2004. Component-based Reuse in Systems Analysis and Design: An Exploratory Study. To be published in the Proceedings of the 11th European Conference on Information Technology Evaluation, Royal Netherlands Academy of Arts and Sciences, Amsterdam, 11-12 November 2004.

CONTENTS

ABSTRACT
ACKNOWLEDGEMENTS
FIGURES AND TABLES
LIST OF INCLUDED ARTICLES

1 INTRODUCTION……..……11
1.1 Software Reuse Overview 11
1.2 Research Motivation..... 13
1.3 Conceptual Structure of the Study 15

2 BACKGROUND RESEARCH……..….19
2.1 Information Systems Development and Information Systems Development Methodology 19
2.1.1 Information Systems Development 19
2.1.2 Information Systems Development Method(ology).....21
2.2 Tool support for information systems development..... 22
2.2.1 CASE Tool.....22
2.2.2 Method Engineering and Metamodeling.....23
2.2.3 MetaCASE Tool and Environment 25

3 SOFTWARE REUSE -- A SILVER BULLET?..... 26
3.1 Ad hoc Reuse vs. Systematic Reuse 27
3.2 Component-Based Reuse..... 29
3.2.1 Component-Based Development 31
3.2.2 Component-Based Reuse Process and its Different Statuses 32
3.2.3 Summary.....33
3.3 Technical Support for Reuse 34
3.3.1 Component Infrastructure Technologies 36
3.3.2 Domain Analysis and Engineering 38
3.3.3 Domain-Specific Modeling.....39
3.3.4 Summary.....41
3.4 Current Tool Support..... 42
3.4.1 Tools Support for Reuse 42
3.4.2 Summary.....45

4 RESEARCH PROBLEM DEFINITION……..……..47
4.1 Technical Problems in Enabling Reuse..... 47
4.2 Research Problem Definition 49
4.3 Research Environment..... 50

5 RESEARCH METHODOLOGY……..……..51
5.1 Choice and Description of Methodologies..... 51

5.2 Application of the Methodology in this Research…….....52

6 SUMMARY OF THE ARTICLES……..……54
6.1 A Framework for Component Reuse in a Metamodeling-Based Software Development 57
6.2 Defining Components in a MetaCASE Environment..... 58
6.3 Component Modeling for Systems Analysis and Design..... 59
6.4 Component Context Specification and Representation in a MetaCASE Environment 60
6.5 Component-Based Reuse in Systems Analysis and Design: An Exploratory Study..... 61
6.6 About the Joint Articles 62

7 CONCLUSION…….……..…….63
7.1 Contribution of the Thesis..... 63
7.2 Limitations of this Study and Directions for Further Research..... 65

REFERENCES……67

YHTEENVETO (FINNISH SUMMARY)……...……76

1 INTRODUCTION

"We will see massive changes [in computer use] over the next few years, causing the initial personal computer revolution to pale into comparative insignificance." As predicted by 22 leaders in software development from academia, industry, and research laboratories 10 years ago (Gibbs 1994), software, together with information technology, is being applied to broader areas of application than ever before. These emerging areas of application are causing information systems development (ISD) to become market-driven. The practice of ISD is thereby being shaped by a varying set of demands originating out of a turbulent business environment. Accordingly, new requirements, new kinds of information systems (IS), and new information systems development methodologies (ISDMs) are being created. In order to gain competitive advantage, IS should be delivered in ways that respond to customers' needs and their timing. Although IS productivity has been steadily rising during the past 30 years (Yourdon 1992), it has not kept up with a rising demand for developing and managing more complex systems (Gibbs 1994), and maintaining existing systems (Mili *et al.* 1995). How to effectively develop new systems has become a perennial research topic in the IS community.

1.1 Software Reuse Overview

Software reuse (Krueger 1992), first introduced by McIlroy (1969) to solve the software crisis, offers great potential in terms of information systems productivity and quality. It has long been recognized that reuse can potentially deliver tremendous benefits. Exploiting reuse opportunities enables significant improvements in software productivity, quality and costs. Not only are there vast benefits to be gained from reuse, but there are also tremendous opportunities to increase reuse. Because IS typically are composed of similar parts (McClure 2001), the majority of systems can be built by assembling existing reusable components.

The idea of reuse is simple, while its execution is not. Although it has been proposed as a solution to the software crisis for decades, it has largely remained on the shelf of promising ideas. Over the last ten years, software reuse researchers and practitioners have learned that success with systematic reuse requires careful attention to be paid to both technical and non-technical issues (Griss 1995; Kim and Stohr 1998). These issues can be summarized from the lessons and findings obtained from empirical studies (Bowen 1992; Frakes and Isoda 1994; Lee and Litecky 1997; Jacobson *et al.* 1997; Rine and Sonneman 1998; Morisio *et al.* 2002). They mainly fall into three categories: organizational factors, technical factors, and human factors.

Long-term, top-down organizational support is a prerequisite for organization-wide systematic reuse (Zand and Samazadeh 1995; Jacobson *et al.* 1997; Kim and Stohr 1998; Rine and Sonneman 1998; Morisio *et al.* 2002). Instead of a stand-alone activity, a reuse "program" is a part of an organization's overall process improvement strategy, and it may require years of investment before it pays off. It is thereby important to clarify the motivation for reuse at the organization level. A clear top management vision and a commitment to introduce and sustain reuse enable managers and engineers to understand the rationale, expectations, and goals of reuse. The clarification of commitment and reuse roles enables construction of the technical support needed for reuse. Even without top management commitment, individuals can spontaneously incorporate reuse into their development activities on the basis of their skill, experience, knowledge, and attitude toward reuse. This *ad hoc* form of reuse, however, is not likely to lead to the overall benefits the organization seeks. To reap such benefits, software reuse support groups, which consist of representatives from each major project or application domain, can play an important coordinating role in the implementation of reuse programs in organizations (Kim and Stohr 1998).

Technical support is the key factor for successful reuse of software components. Without technique and tool support, a reuse project cannot be successful, because the reuse activities are always carried out with certain methods and tools (Kim and Stohr 1998). The technical factors concern reuse success in both process and product aspects. In the process viewpoint, it is important to adapt the systems development life cycle model to adequately address reuse activities, as explained in IEEE Standard 1517 (McClure 2001), unless software reuse is explicitly defined in the software life cycle processes, an organization will not be able to repeatedly exploit reuse opportunities in multiple software projects or software products. The adaptation covers both introducing reuse-specific processes (Kim and Stohr 1998; Morisio *et al.* 2002), i.e. component search, selection, adaptation, and integration, and modifying non-reuse process (Kim and Stohr 1998; Morisio *et al.* 2002), i.e. seamless integration of reuse activities into the traditional systems development process. In the product perspective, an explicit and uniform defini-

tion of reusable components is essential to systematic reuse. The organization has to establish compatible development environments, a well-designed architecture, and the proper form of components (Frakes and Isoda 1994; Jacobson *et al.* 1997; Kim and Stohr 1998;). With a uniform definition of components and a well-designed architecture, reusable components can be developed in line with the underlying principles of reusability and the architecture. Correspondingly, component definition tools, repository, search tools, and reuse metrics (Frakes and Terry 1996) are easy to implement for defining, storing, retrieving, integrating, and managing reusable components.

Human awareness, attitude, and capability of reuse are essential social factors for sustaining process change (Morisio *et al.* 2002). Engineers' skill in using software development techniques, experiences with the development environments, knowledge about problem domains (Lee and Litecky 1997), and attitudes toward reuse are important. They are considered by instituting a reuse culture, providing training, adhering to standards, and securing management commitment (Griss 1995). Training is a good way to promote reuse. Both managers and engineers need training to create an awareness, understanding and acceptance of reuse. Meanwhile, effective reuse requires active engagement in changes in technology and management in terms of component development and reuse. Incentives help the organizations to institutionalize reuse technology (Card and Comer 1994; Frakes and Isoda 1994). In particular, introducing reuse incentives at the early stage of a reuse program is necessary to encourage reuse practices.

These three aspects are indispensable and interdependent. No single aspect is either the major impediment to effective reuse or the most critical success factor. Reuse is a business issue that involves organizational change, technology transition, and individuals' subjective attitudes. In particular, organizational commitment is the root, and the technical support and human factors are the enablers. Without organizational support, neither developing a reuse mindset nor a methodical support for reusable component development will be worthwhile, and vice versa.

1.2 Research Motivation

Here, as throughout the thesis, we are concerned with the technical impetus to systematic reuse.

Technical support is an important and indispensable factor in promoting reuse practice. There has been a considerable amount of reuse research published over the last two decades (Kim and Stohr 1998; Zand *et al.* 1999; Biddle *et al.* 2003). Most of this research focuses on enabling technologies of software reuse, like identifying, classifying, retrieving, understanding, integrating, and maintaining reusable components. Consequently, a number of conceptual

frameworks for research on technical enabling of software reuse have been proposed in the literature. For example, Preito-Díaz and Freeman (1987) tackle the component retrieval problem and propose a faceted classification scheme based on reusability-related attributes and a selection mechanism as a partial solution to the software reuse problem. In addition, many similar studies have attempted to define reusable components and organize them into repositories, e.g. Maarek *et al.* (1991), Maiden and Sutcliffe (1992), Castano and Antonellis (1997), Zhang (2000a), Sugumaran and Storey (2003), etc. Moreover, Biggerstaff and Richter (1989) discuss software reuse approaches from the stand points of composition technologies and generation technologies, and emphasize the potential value of the reuse of design. Furthermore, Freeman (1983) and Krueger (1992) present diverse reuse approaches in terms of reusable artifacts and the way of reuse. Reusable artifacts vary from the programming code to high-level logic structure and design knowledge.

Diverse studies on the technical enabling of software reuse have been conducted. Although the generic objective of these studies is to enable the broad reuse of all types of information generated during the ISD process, most of the research has concentrated only on reuse within the single form of concrete reusable assets (Kim and Stohr 1998), and appears to be still at a formative stage (Kim and Stohr 1998). Besides implementing diverse techniques to support software reuse, research should seek both an approach and an environment to help users understand unfamiliar reusable components and decide whether to use them or not. Therefore, there is a need to combine the most effective techniques into existing ISD practices, and develop an integrated methodology that can be readily understood and adopted by a large community of system developers.

Traditional ISDMs do not explicitly support software reuse (Kim and Stohr 1998). The recently developed methodologies, such as domain engineering (Arango and Prieto-Diaz 1991), reverse engineering (Biggerstaff 1989; Müller *et al.* 2000), the object-oriented approach (Griss 1996), and component-based approach make it easier to take an advantage of software similarities and support reuse in the systems development process. However, due to the immature stage of technical support of reuse, most methodologies were not originally designed to support reuse. Existing methods such as OO methods do not incorporate key learning from the reuse community (Zand *et al.* 1999).

Meanwhile, current tools provide weak methodical support for reuse. The majority of computer aided software engineering (CASE) tools available only assist in building graphical models (Sodhi and Sodhi 1998), generating code, or reusing independent single resources in the implementation phase of the ISD process (Kim and Stohr 1998). There is no solution that can offer powerful mechanisms to model and organize different types of reusable components. CASE tools should be extended to provide mechanisms to support reuse in every phase of ISD.

Reuse is by no means an automatic by-product of following an ISDM. Instead of continuously constructing and illustrating new techniques and meth-

odologies to support efficient and effective ISD, we need to study the existing methods and tools known and used by software engineers, and try to institutionalize the concept of reuse into the whole systems development life cycle. It is distinct from the above-mentioned researches, our work aims at the improvement of systematic reuse by integrating existing component-based approaches into the systems development environment, more specifically, a metamodelling-based systems development environment - a metaCASE environment (Kelly 1997).

The aim of this thesis is to build up a systematic reuse framework in a metaCASE environment. We study the generic concepts, process, and techniques that can be used to incorporate the reuse concept into the existing metaCASE environment. In particular, we take advantage of the metamodeling feature of the metaCASE environment to specify the component model and deploy it in the process of both method engineering and ISD. As this is an empirical study, we incorporate the component model into the traditional OO design methods to study the impact of the model on reuse practice during the systems design process.

Work towards these goals improves the reuse support of the existing metaCASE environment, which in turn improves systematic reuse practices in the ISD process. It studies how reuse techniques are incorporated into the existing systems development methods and their supporting environment. In particular, the exploratory study of component-based reuse during the systems design process is a unique laboratory experiment. It demonstrates how the component-based approach can be integrated into the traditional OO design methods, allows a higher level of reuse, and shifts the reuse effort to a point much earlier in the systems development process. This saves design phase effort in addition to the normal benefits of reuse practice. Meanwhile, the laboratory experiment can be tailored and applied to other empirical studies in similar research areas.

1.3 Conceptual Structure of the Study

As we are concerned with systematic reuse support in the systems development environment, the main concepts involved in our study include systems, systems development methodologies, and tools that support systematic reuse. The essential concepts and their dependencies can be captured in the metamodel shown in FIGURE 1.

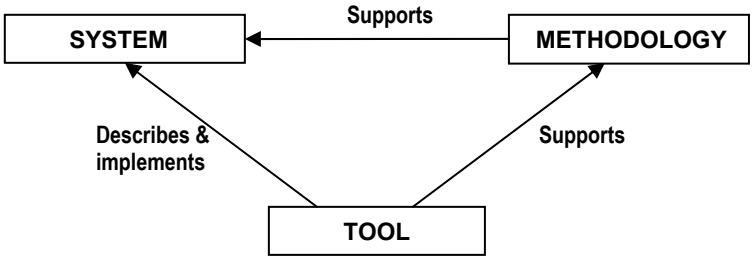

FIGURE 1 Metamodel of the background concepts

A *system* is a collection of components organized to fulfill defined purpose (Sage 1995). It consists of two essential features: the basic elements that make up the system, and tasks that the system must perform to fulfill the defined purpose. The concept of the system exists everywhere from a small artificial object with a specific function (e.g. pen, bike, etc.) to a large system with complex functionality (e.g. a library management system, an airspace system, etc.). Systems are developed by using a set of tools and by following a specific methodology. In the context of ISD, systems mainly refer to software-intensive information systems, which comprise hardware components, software components, and operational processes to accomplish a set of functions.

A *methodology* is a way of carrying out systems development. Merriam-Webster on-line dictionary (2004) defines it as "a body of methods, rules, and postulates employed by a discipline: a particular procedure or set of procedures". Accordingly, a methodology consists of a set of predefined processes. Each process encompasses many techniques and a notation to produce diagrams, documents and other deliverables. Examples of information systems development methodologies include structured systems analysis and design, object-oriented (OO) analysis and design, rapid application development, etc. They are embodied in a set of tools to support systems development.

A *tool* represents a device that aids in accomplishing a task. It embodies a (part of) methodology to support some aspects of the systems development process. As a tool provides an automated way of accomplishing a systems development task, it is sometimes regarded as one element of a methodology. A set of tools that are integrated to support the ISD process is called a CASE environment. Different CASE environments focus on different aspects of the development process and thus differ greatly in their functionality. Some provide toolsets which address the early stages of systems development, i.e. strategy, planning and analysis; some address physical design, programming and implementation stages (Avison and Fitzgerald 1995); others address mechanisms to define different modeling techniques (Koskinen 2000); yet others integrate the three into a single, fully integrated development and support facility. MetaEdit+ is an example of the integrated development environment (Kelly *et al.* 1996).

The three nodes of the metamodel correspond roughly to the basic concepts used in our study. Our ultimate goal is to find a way to develop a high quality information system in a time-saving and economical manner. In order to achieve this goal, we study the existing reuse techniques and the systems development methodologies, and incorporate the promising reuse techniques into the existing systems development tools.

The context of our study is depicted in the instantiation of the conceptual model in FIGURE 2. There are two layers of instantiation. The inner layer shows the instance of the metamodel in the field of ISD, its supporting methodologies and tools, and the outer layer further demonstrates the same concepts in the context of software reuse. As our study is based on a metamodelling-based systems development environment, the instances of tool distinguish between CASE tools and metaCASE tools. The CASE tool embodies a specific methodology or technique that supports ISD. The metaCASE tool specifies the systems development methodologies as different CASE tools which eventually support the process of ISD. The advantage of CASE tools and metaCASE tools for reuse is the existence of a repository of software-related artifacts that record domain knowledge and are linked together through all the stages of the ISD process (Karakostas 1989; Kim and Stohr 1998).

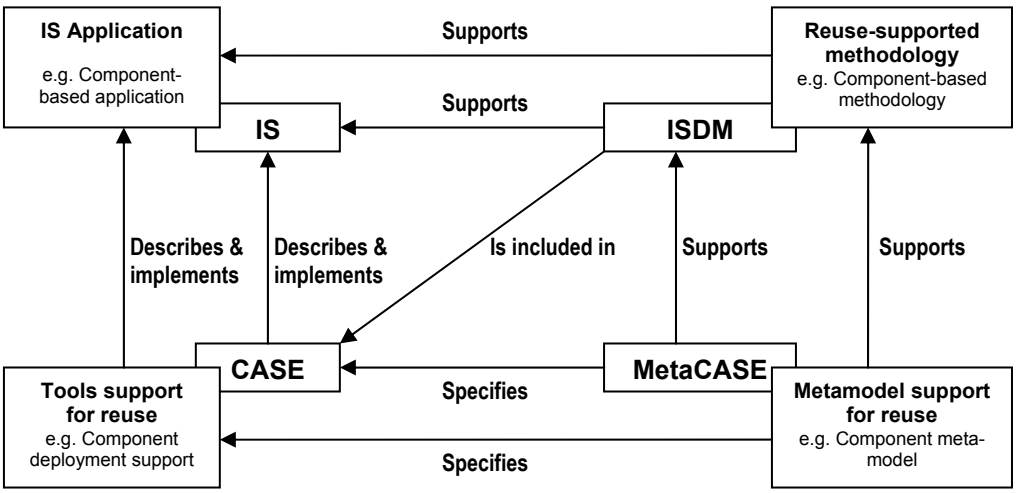

FIGURE 2 Background of the study

In the next section we will first look at the conceptual background within which the research takes place, i.e. the concepts in the inner layer in FIGURE 2. Next, in section 3 we examine the current situation of the methodology and the tools that support reuse, and explain problems of current reuse practice, i.e. the concepts in the outer layer in FIGURE 2. These motivate and provide our research problems, discussed in section 4. In section 5 we describe the research environment in which our work was carried out and the methodology that was applied in the research. Finally, a short summary of each enclosed paper is presented in

section 6, followed by a brief overall conclusion, a discussion of the limitations of the study and directions for future research in section 7.

2 BACKGROUND RESEARCH

In this section, we examine the inner layer of the research model (FIGURE 2) to propose basic definitions of the domain of the study. The basic concepts include ISD, ISDM, and tools support for ISD.

2.1 Information Systems Development and Information Systems Development Methodology

In the 70s, the importance of information technology began to be noticed (Galbraith 1977). As a major organizational problem-solver, information technology increases an organization's capacity to cope with external and internal complexity and improve its performance. The field of IS is premised on the centrality of information technology in everyday socio-economic life (Orlikowski and Iacono 2001). Its development consists of a wide variety of activities and processes which come together to create an information systems designed for a specific purposes. Generally, there are two major concerns within its community: the nature of ISD and the characteristics of methods to support ISD (Lyytinen 1987).

2.1.1 Information Systems Development

This section will elaborate a set of basic terms about information systems development (ISD) which we use throughout the thesis. By following the concept of ISD presented by Welke (1983), Lyytinen (1987), and Hirschheim *et al.* (1995), we describe the key features of ISD as a set of object system changes. Following Welke (1983) and Lyytinen (1987), we define ISD as follows:

Information systems development (ISD) *is a change process taken with respect to a number of object systems in set of environments by a development group to achieve or maintain some objectives held by some stakeholders.*

Seen in this light ISD is a multidimensional social change covering organizational knowledge, knowledge representation, linguistics, epistemology, technology, and so on (Lyytinen 1987; Hirschheim *et al.* 1991; Iivari *et al.* 1998). It is the change process covering the real world, conceptualizations of the real world, and descriptions of these conceptualizations, in order to represent target systems in a complete and unambiguous way.

> *Real world* – changes that is about to influence social behaviors and other arrangements.
> *Conceptualizations of the real world* – concepts that make sense of the phenomena in question, like the ideas about material flows, information flows, and their interactions.
> *Descriptions of conceptualizations* - descriptive languages such as a workflow notation, or UML notations.
> *Target systems* – achievements of change processes.

ISD is a web of technological, social, psychological and cultural phenomena. Here we focus on two essential concepts: object systems and the change process.

Object systems identify a target of change (Hirschheim *et al.* 1995). It consists of phenomena 'perceived' by the development group. In general, the development group can identify object systems in three principal perception schemes: structure, function and behavior (Iivari 1990). In the structure perspective, the object system is perceived as a set of static objects relevant to the real world in question, their relationships and attributes, etc. In the function perspective, the object system is perceived as a set of activities related to the real world, with input and output. In the behavior perspective, the object system is perceived as a set of changes of state over time. The perception can be represented in multiple ways: free-form text, semiformal notations such as graphical description, and formal mathematical notations (Hirschheim *et al.* 1995). In general, we call representations of perception *models*. The chosen form of a model depends primarily on the feature of the real world in question and its required degree of accuracy and formality.

A change process is an event in which phenomena, i.e. objects and their relationships in object systems, come into being as a result of the development groups' deliberate action (Hirschheim *et al.* 1995). It can be regarded as a modeling process consisting of a set of systems development activities to derive the representation of the object systems. The change process is enabled by combining techniques. A *technique* is a procedure, with a prescribed notation and rule, to perform the change processes (Brinkkemper 1996). For example, the modeling of a data flow and interviewing are techniques conducted by the development group. As mentioned above, object systems can be perceived from various aspects during ISD. This leads to the creation of a number of techniques for delivering the representation of the object systems from different perspectives. By using a certain technique, the development group perceives, defines and

communicates certain aspects of the object systems (Tolvanen 1998). For example, the perception of the static relationships of the objects can be represented in a class diagram, the process to complete a task performed by the object systems can be represented in a data flow diagram, and the behavior or state changes of certain objects can be represented in a state transition diagram.

Similar to the diversity of representation at forms of an object system, we use different terminologies to represent the concept of object systems and change processes: model to represent an object system, and ISD methods or techniques to represent the enablers of change processes. These terminologies and their concepts are used throughout the thesis.

2.1.2 Information Systems Development Method(ology)

ISD is featured as a change process undertaken with respect to a set of object systems. In order to enable and support the change process, techniques and methods are indispensable.

A *method* consists of a set of combined techniques to perform an ISD project. It states by whom, in what order, and in what way the combined techniques are used (Smolander *et al.* 1990) to achieve objectives held by stakeholders. There are different definitions of the concept of method. We follow the definition given by Brinkkemper (1996) and define a method as an approach to performing a systems development project.

*A **method** is based on a specific way of thinking, consisting of directions and rules, structured in a systematic way with corresponding development products.*

A *methodology* is a scientific theory of the systems development action. Different schools (see Olle *et al.* (1986), Brinkkemper (1990), Kumar and Welke (1992), Harmsen and Brinkkemper (1993), and Harmsen *et al.* (1994)) differentiate between concept of method and methodology differently. However, in this thesis, we will not distinguish methodology from method, and quote a definition of methodology that is similar to the definition of method. It is given by following the definition in Lyytinen (1987) and Hirschheim *et al.* (1995).

***Information systems development method(ology) (ISDM)** is an organized collection of concepts, beliefs, values, and normative principles (knowledge) supported by material resources.*

An ISDM is codified into a set of goal-oriented procedures that guide the work and cooperation of the various stakeholders involved in the change process to build the target systems. The systems development method generally denotes communicable, formalizable, and enactable knowledge about ISD, i.e. how to identify, specify, implement, and evaluate changes and accordingly organize the systems development process through the definition of the physical world (a way of thinking), the data model (a way of modeling), and the process model (a way of working). Accordingly, it can be used to identify problematic situa-

tions and object systems for change (Davis 1982), to generate and analyze the correctness of change actions (Olle *et al.* 1982), to assess and evaluate the effectiveness and efficiency of change actions (Kleijnen 1980), and to carry out and implement changes (Jackson 1975; Keen and Scott-Morton 1978). In general, a single method does not usually cover all aspects of systems development (Solvberg and Kung 1993; Lyytinen and Zhang 2000). To better support ISD, an organization normally has to reuse its knowledge of selected methods and tailor them to its requirements by expanding, combining, and constraining.

There is a large and confusing variety of ISDMs in existence (Avison and Fitzgerald 1995): some are similar and differentiated only for marketing purposes, and some are developed in-house and internal to individual organizations (Hardy *et al.* 1995; Russo and Wynekoop 1995). Examples of widely used methods in systems analysis and design include structured analysis and design methods (Yourdon 1989) and object-oriented methods, such as the unified modeling language (UML 1995).

2.2 Tool Support for Information Systems Development

In this section we present and elaborate on concepts and tools related to ISD tools.

2.2.1 CASE Tool

When an ISD method is supported by some instrument (a template, a questionnaire, or a computer program) this is called a development tool (Lyytinen *et al.* 1989). A development tool mainly supports a (part of) the development process by providing a set of functionalities such as abstraction of the object system into models, checking that models are consistent, converting results from one form of model and representation to another, and providing specifications for review (Olle *et al.* 1991). When a computer program is used as the instrument to support for the functionalities, we regard the tool as a computer aided systems/software engineering (CASE) tool.

Computer Aided Systems/Software Engineering (CASE) *is a disciplined approach to systems development in which computers are used to provide some automated support in analyzing, designing, implementing and maintaining information systems.*

CASE is a term that has been around for decades. In the early 1980s, CASE tools referred to stand-alone tools to help automate program diagramming and documentation. By the mid-1980s the capabilities of systems analysis and design diagramming tools had broadened to include automatic checks of designs. During this time, the importance of having an information repository, dictionary, or encyclopedia as the center of a CASE tool became more widely appreciated. Nowadays, a CASE tool can generally be applied to any system or collec-

tion of tools that helps automate the systems design and development process. Compilers, structured editors, source-code control systems, and modeling tools are all, strictly speaking, CASE tools. In particular, many current CASE tools are constructed to meet the needs of a specific application domain and support the application development within a domain.

When CASE tools are integrated into an environment to cover several development stages, including a model editor, document generator, code generator and repository, we called it a *CASE environment*. It is commonly used when coupled with a method that enables developers to abstract away from the source code to a level where the architecture and design become more apparent and easier to understand.

A CASE tool or environment automates time-consuming aspects of the systems development process including drawing diagrams, cross-checking of concepts across the system models, and generating system documents, code structure, and database schemas. There are several hundreds of CASE tools or CASE environments. Some well-known environments in industry are Rational Rose (Quatrani 1997), Axiom CASE suit (STGCASE 2003), and MetaEdit+ (MetaCASE 1999).

However, the problem with CASE tools is that the view of the development process has been hard-coded, and therefore cannot be changed or customized to include knowledge that is based upon information engineers' practical experience (Hofstede and Verhoef 1996; Kelly 1997). A CASE tool does not help "re-invent" anything, and only supports a fixed method, which cannot necessarily cater for the requirements of organizations in a rapidly changing market. The organization faces a continuous need to integrate other techniques, methods or tools (Hardy *et al.* 1995; Russo and Wynekoop 1995; Tolvanen 1998). In order to facilitate better ISD requirements, other techniques and tools are needed to support the flexible creation, modification, and reuse of ISDMs and tools in specific application domains.

2.2.2 Method Engineering and Metamodeling

The ISDM should be constructed to meet a particular ISD needs. Although ISDMs have proliferated in great numbers, a single method is hardly applied as it is originally defined. The empirical studies of method uses (Russo *et al.* 1996; Hardy *et al.* 1995) show that more than 80% of methods are always customized for local needs. Situations at an organization, project or individual level often cause changes in methods (Tolvanen 1998). CASE tools with hard-coded methods cater poorly for the change requirements of their supporting methods. There has been a tendency to construct methods on a project or organization basis, which is called method engineering.

Method engineering *is a discipline to design, construct, and adapt methods, techniques, and tools for systems development (Kumar and Welke 1992; Brinkkemper 1996).*

Method engineering is based upon the assumption that organizations are continually adapting and using ISD methods in their development endeavors. There are different views on the method engineering process. In Harmsen and Brinkkemper (1993), a method is viewed as a collection of method fragments saved in the method base. A method fragment can encompass products, processes, or tools. Method engineering is thus seen as a process of assembling a method from its different fragments. But on the other hand, in Brinkkemper (1990), a method can be abstracted as a conceptual model, called a metamodel. Method engineering is a metamodeling process to specify and integrate a method into a metamodel from the perspectives of concepts, properties, rules, and generators. In this thesis, we take the metamodel viewpoint as the pillar of method engineering.

Metamodels are conceptual models of methods. They conceptualize a method by selecting a specific set of concepts and representation perspectives. Generally, metamodels can be roughly divided into processes and product models. For example, the metamodel of a class diagram is a meta-data model for its static aspects, and a workflow can be meta-process modeled for its dynamic aspects.

Metamodeling in general is the modeling of the languages we use to model with (Brinkkemper 1990). In ISD it is the process of specifying a metamodel using a metamodeling language. A metamodel is the model of a method, namely a type of modeling language of ISD, which is further used in the modeling process to construct models of application systems. The process of metamodeling a method is similar to that of modeling a system (Kelly 1997), except that the object systems are different: a set of methods with rules and constraints versus an application system with requirements specification. The relationship between modeling and metamodeling is illustrated in FIGURE 3.

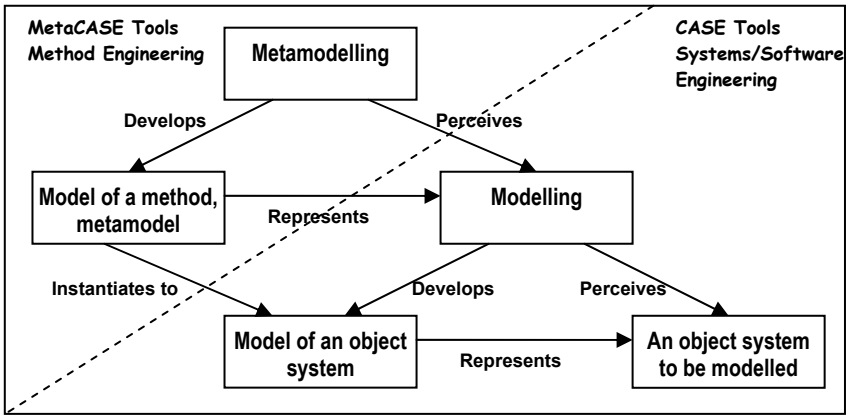

FIGURE 3 Metamodeling and modeling in a metaCASE environment (after Brinkkemper (1990))

In principle every ISD method can be used for some parts of metamodeling, even soft systems methodology (Checkland and Scholes 1990). The only difference is that some methods are more suitable for metamodeling than others, and there are some methods that have been developed specifically for metamodel-

ing, like UML (UML 1995). Generally, metamodeling enables a method to be implemented in a metaCASE tool.

2.2.3 MetaCASE Tool and Environment

In general, a CASE tool does not support the method engineering process. Although some CASE tools nowadays exhibit some method engineering functionality by allowing additions and minor cosmetic changes to their existing method support (Kelly 1997), a real method engineering supported tool allows a method to be completely built or changed. We regard such a type of tool as a metaCASE tool. Combined with a repository, a metaCASE tool should provide functionality to search and reuse existing conceptual definitions to continuously define methods "on-the-wing".

*A **metaCASE tool** is a software tool that supports the design and generation of CASE tools. It facilitates the design and specification of a method whose full and formal definition is not readily available.*

In this setting, the metaCASE tool is a support tool for the method engineer. It is therefore sometimes called Computer-Aided Method Engineering (CAME) tool (Harmsen and Brinkkemper 1993).

Generally, we call the system a metaCASE environment when it supports metamodeling within the same environment as modeling, and it produces the metamodel and inputs it to the metaCASE tools. That is to say that a metaCASE environment provides functionality for both method engineering and software (system) engineering. Since most ISDMs mainly provide support for modeling in systems analysis and design, we also call a metaCASE environment a metamodeling-based systems analysis and design environment. It will be discussed in the following chapters and is the central theme of this thesis.

A number of metaCASE tools and environments have been developed during the last decade, e.g. commercial products such as MetaEdit+ (MetaCASE Consulting), MethodMaker (Mark V), ToolBuilder (Sunderland/IPSYS/ Lincon); and research prototypes such as Meta View (Alberta) and metaGen (Paris). A brief overview of these tools is provided by Kelly (1997).

3 SOFTWARE REUSE – A SILVER BULLET?

Reuse in ISD starts from software salvage, which applies existing software and design artifacts in order to deliver new applications, or to maintain the old ones. Software reuse is the answer to the software crisis (Krueger 1992; Lee and Litecky 1997; McClure 2001). It reduces the development cost by building new systems from predefined artifacts rather than from scratch. Since the reusable assets have been proven in the previous systems and are less prone to error, quality and reliability are improved (Gibbs 1994).

Although the practice of reuse is regarded as a silver bullet to reduce cost and improve quality in the ISD process, it does not happen by accident. Reuse is not an automatic by-product of using new technologies. It is a long-term project requiring real commitment and strategic thinking (Card and Comer 1994). Reuse activities must be an inherent part of the life cycle process. Above all, the reuse strategy for a project, an application domain and the enterprise must be planned prior to development of the application if the benefits promised by reuse are to be achieved in practice.

The reuse community has also realized that reuse is not only a technical problem but it involves several other factors. ISD is a social process that involves actors in various social roles interacting in a variety ways (Hirschheim and Klein 1989; Hirschheim *et al.* 1991). The socio-technical nature of ISD implies that a successful reuse program not only needs technology support, but also changes in the managerial and cultural domains. Moreover, the success of reuse is related to several other socio-technical factors as well. These are very useful but do not themselves guarantee success. These factors are as follows.

Type of software production - A product family shares common features and offers natural condition for reuse (McClure 2001; Morisio *et al.* 2002).

Maturity level – If an organization has a mature software development method and process, it is easy to ensure that reuse is considered when appropriate (Card and Comer 1994; Morisio *et al.* 2002).

Ownership - An organization with its own processes can adapt an existing product development process by integrating reuse–specific processes as

needed, which makes it easier to provide technical support for reuse than in the case of a project whose processes are owned elsewhere, e.g. when sub-contracting is involved (Morisio *et al.* 2002).

Standardization – Besides the uniform definition of component and its reuse semantics on the organizational level, the spread of standardization policies across organizations might positively affect the rate of reuse (Banker and Kauffman 1991; Lee and Litecky 1997). The existence of standards will increase the reusability of components across organizational boundaries.

Furthermore, there are some complex but important issues to take into account when promoting reuse practice, such as legal questions (Frakes and Isoda 1994) related to the rights and responsibilities of providers and consumers of reusable components, and evaluation and cost-benefit analyses (Card and Comer 1994; Frakes and Isoda 1994) that decide the feasibility of systematic reuse. These issues are less of an impediment to reuse within an organization, but will multiply as reusable components cross organizational boundaries and reuse programs grow up.

Below we follow the outer layer of the research background model (FIGURE 2) to present the different methodologies (techniques) and tools that support reuse in systems development, and study ways of improving reuse practices in the ISD process.

3.1 Ad hoc Reuse vs. Systematic Reuse

Ad hoc reuse applies when there is no defined process for performing reuse. Reuse is an implicit byproduct of the ISD processes. That is to say an individual or a small development group can practice reuse without any proper document, process, and structured formal approach. With ad hoc reuse, the reusable parts are scavenged from previously built systems and applications for use as building blocks to construct new systems, applications, or enhancements (McClure 2001).

Besides ad hoc reuse, some literatures identify another type of reuse between ad hoc and systematic, called *opportunistic reuse*. Different from the unplanned practice of reuse, opportunistic reuse depends on the engineer to identify the needs and retrieve the needed reusable artifacts. However, there is still no standard process in place for guidance.

In ad hoc and opportunistic reuse processes, reuse is an afterthought. The reusable artifacts were probably not designed for reuse, which makes the obtained reuse potentiality leading to marginal gain. Often, they must be force-fitted into the target systems, which lead to compromise of the functional and non-functional requirements specification. Because of the disappointing results from ad hoc or opportunistic reuse experiences, the reuse community realized that reuse requires a broader application in systems development process. The processes must be applied at the organizational level for a family or related

software systems and applications, rather than only at the project level for an individual system or application (McClure 2001). In order to obtain more substantial gains, a systematic approach to reuse is necessary.

Systematic reuse *is the practice of reuse according to a well-defined repeatable process (McClure 2001, 42).*

In contrast to ad hoc and opportunistic reuse, the practice of systematic reuse forms an integral and explicit part of the development process. It encompasses a set of purposeful activities, like creation, management, support, and reuse of assets. Its repeatable feature requires a shift from a casual reuse case to a formalized organizational level of reuse, and a shift from crafting one system at a time to the use of engineering principles for entire families of systems (Frakes and Isoda 1994; Sodhi and Sodhi 1998).

Prior research (Gaffney and Durek 1989; Banker and Kauffman 1991; Basili *et al.* 1996) has focused on studies of systematic reuse of previous written code. As the concept of reuse is relaxed from referring to interchangeable source code parts (McIlroy 1969) to a broader reuse of work products (Freeman 1983) and a still broader concept of reusing artifacts associated with a software project including knowledge (Basili and Rombach 1988), systematic reuse, therefore, comes to involve more than just code. It involves organizing and encapsulating experience and setting up the mechanisms and organizational structures to support such a process. In particular, it becomes concerned with high-level life-cycle artifacts, like requirements, domain knowledge, analysis patterns, conceptual designs, architectural structures, and documentation (Biggerstaff and Perlis 1989; Frakes and Fox 1995; Karlsson 1995; Purao *et al.* 2003). At the organization level, systematic reuse makes it possible to identify which software artifacts have the greatest reuse potential and to plan for their reuse in up-and-coming development, maintenance, and enhancement projects. For example, Purao *et al.* (2003) define a group of related, generic objects with stereotypical attributes and behaviors as an analysis pattern, which can be reused to support conceptual design in different application domains.

Systematic reuse is not therefore limited to code salvage at the system implementation stage. Instead, it is interleaved at all phases of the development (Jacobson *et al.* 1997). Indeed, the earlier the reuse activity starts, the more benefits the organization may achieve. Reusable assets at the analysis and design stage are at a higher level of abstraction than code and less implementation-specific, and therefore more understandable and reusable. Also, since the earlier stages of a project (e.g. requirements, analysis, and design stages) are more expensive than coding (Boem 1987; Kotonya and Sommerville 1998), reusing earlier stage assets can potentially provide greater savings than reusing code. Moreover, reuse at the design level can lead to reuse at the code level. Information traceability tools help engineers to trace the desired information between different development stages.

Meanwhile, systematic reuse is domain-oriented (Frakes and Isoda 1994; Sodhi and Sodhi 1998). Within the same application domain, due to the com-

mon features existing between different systems, there are many more opportunities to reuse existing software assets. Consequently, there is a better chance of getting payback from the investment in designing and preparing for reusable assets in terms of domain models within the systems family (McClure 2001).

A systematic reuse program is a long-term investment: because application domains have to be analyzed and defined, reusable assets and their interfaces have to be defined and stored, and the reuse process has to be seamlessly integrated into the ISD process. A systematic reuse program provides the foundation for coordinating reuse efforts within and across software project teams. It raises the practice of reuse to the organizational level, which enables the highest possible stage of reuse when building new applications, thereby achieving the maximum possible reuse benefits (McClure 2001).

3.2 Component-Based Reuse

Software reuse is a process of building or assembling software applications and systems from previously developed software artifacts designed for reuse (McClure 2001). The key enabler of software reuse is the reusable asset. Without reusable assets, a software reuse program cannot be initiated, carried out, and completed successfully. Assets can be defined as requirements, design models, code, testing cases, documentation, standards, and plans. Different types of assets have different representational forms, and are used at different development stages. Since reuse occurs across similar systems or in widely different software systems, without any specification expressed in a commonly understandable way, it is difficult for a user to grasp the contents and features of every reusable asset. In order to facilitate systematic reuse, the reusable assets should be wrapped in an explicit interface to present the abstract conceptual information. We call an asset of this type component. Component is expected to be the primary driver of the dramatic changes about to take place in ISD (McClure 2001).

What constitutes a component? Many initial ideas in component-based reuse come from structured methods with modular design techniques, reuse-oriented approaches, and object-oriented analyses and designs, which support modular design, encapsulation, inheritance hierarchies or separated specification from an implementation. With the recent advent of component-based reuse approaches to building applications, the term component has come to mean many different things to different people.

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 to build applications.

It is worth noting, however, that a broader interpretation of component-based reuse technologies is also possible in which a component is not necessarily synonymous with executable software. Different from the component created in systems implementation, most components in systems analysis and design are models constructed using modeling languages such as UML. Similarly, a component may be wholly defined by a specification, documentation, or models at varying levels of abstraction and of different sizes from which code can be generated. For example, metamodels specifying a static perspective supported by a method, e.g. a class diagram in the UML method, can be regarded as a component at the metamodel level. Meanwhile, requirements specifications, designs, architectural descriptions, test suites, and so on are all components that are being reused.

Regardless of whether the narrower or broader interpretation of a component is used, Szyperski's (1998) definition emphasizes the importance of the interface and context specification of a component as well as its independent nature:

*A **software component** is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties (Szyperski 1998).*

The definition highlights the most common perspectives on what constitutes a component by focusing on three key words: interface, independent, and context.

Interface is the most important feature of a component. The interface encapsulates a component by offering an abstract description of the services (operations or functions) that it provides to its consumers. The interfaces of different types of components have different representation formats, but the information expressed should relate to the services or functionality provided by the component, and the semantic association with which the component communicates with other components. A well-defined interface enables a component to hide implementation details, to be fitted into software architecture, and to be replaced easily. Component can communicate with other components.

The independent feature can be thought of as inheriting characteristics and benefits from the interface feature. Because of the interface description, a component becomes an independent deliverable piece of functionality providing access to its service through the interface (Brown and Short 1998). Thus, the independent feature focuses on reuse and considers a component to be an implementation encapsulation boundary, which can be reused as a unit.

Meanwhile, the context dependencies ensure that components are not isolated independents. They represent different types of dependencies between components, which provides contextual information to their consumers, and better guides them to use them.

3.2.1 Component-Based Development

Component-based development (CBD) is the latest embodiment of systematic reuse (McClure 2001). It is an assembly approach to ISD in which software-intensive systems are constructed by means of assembling components. Some of these components may be harvested from existing systems, some may be retrieved from third parties, and some may be developed anew for the project at hand.

Component-based development (CBD) *is a software-intensive systems development paradigm where all aspects and phases of the development life cycle, including requirements analysis, design, construction, testing, deployment, and the supporting technical infrastructure are based on components.*

The strategy underlying CBD is to use predefined components and architecture to eliminate redesigning and rebuilding the same software structure again and again. Here developers consider applications not to be monolithic. Rather they view them as a set of separable, interacting sub-systems. Because of this underlying strategy, not only does the CBD approach imply reuse, it demands reuse to deliver significant gains in productivity, quality, and development speed (McClure 2001). Reuse lies at the heart of the CBD approach.

The benefits of exploiting CBD derive directly from the benefits of reuse. The CBD approach can provide very significant software productivity, quality, and cost improvements (McClure 2001). With the support of component technologies, developers can search for components from their previous project repository or the commercial standard component repository to reuse them, which is quicker than starting from scratching. The involvement of component at the systems analysis and design stages especially shortens the design-to-production life cycle. Productivity is improved by assembling existing components. Since these components have been well tested in their previous use and wrapped in "standard" interface, the system delivery time is reduced by less testing and the quality is increased by using pre-tested components with precise specifications.

CBD not only brings tremendous benefits, but also brings opportunities to increase reuse practice in ISD projects. Analysis of software and systems application has shown that there are many functions in common between different applications (Tracz 1988). It indicates that the major part of an application can be assembled from pre-defined components. The CBD approach facilitates componentization of the commonalities and hides the complexity for distribution purposes, which greatly promotes reuse in ISD.

In order to successfully apply the CBD approach, Kiely (1998) mentions that three main elements are needed in the component-based system infrastructure: a uniform design notation, a repository, and a standard interface. A uniform design notation provides an accepted way of describing components' functions and properties. For example, UML (1995) is a standard notation widely used in the industry. A repository is needed as a means of cataloguing

available components with descriptions for component retrieval. A standard CBD interface generalizes components that can be accessed by any applications.

3.2.2 Component-Based Reuse Process and its Different Statuses

A reuse program mainly considers two kinds of process: the component management process and the component reuse process. The reuse process is the primary process carried out during the ISD process.

By following the activities of asset management process specified in IEEE Standard 1517 (IEEE 1999), the activities of component management can be divided into three parts. The first two parts include preparation activities, e.g. part one is to create and review a component management plan, called process implementation. Part two is to define a component storage and retrieval mechanism. In detail, engineers define, implement, review, and maintain a component storage and retrieval mechanism, as well as a component classification scheme at this stage. The third part is to manage and control components, which includes activities such as evaluation, storage, classification, and retirement of components, as well as tracing reports and monitoring component changes. The management process is a support process that contributes to the success of a reuse project.

The component reuse process consists of a set of basic activities, such as component definition, selection, adaptation, integration, and maintenance. They are seamlessly integrated into the system development process. An example of CBD environment and component reuse activities will be discussed in the next section. In different ISD approaches, the integration of reuse activities might be slightly different. However, the basic activities are not changed. Accordingly, components in such a reuse process can have different statuses. We take the terms in the essential CBD activities framework suggested by Brown and Short (1998), and classify components into five groups: off-the-shelf, qualified, adapted, assembled, and updated components, which correspond to the five basic activities in the reuse process mentioned above: definition, selection, adaptation, integration, and maintenance.

Off-the-shelf components are components identified as being of potential interest. A process of investigation is required before selecting one as a reusable component. This group of components may come from a variety of local and remote sources, and also known as commercial off-the-shelf (COTS). In general, off-the-shelf components are the input of the component selection process. They are selected or involved at the requirements analysis stage. Because of uncertainty about the status of their potential reuse, little may be known about a component's characteristics at the requirements stage. The information available may be simply its name, its parameters, and some idea of its required operating environment. Many of the required and available interactions with other components may remain hidden.

Qualified components are candidate components selected from among the off-the-shelf components. They are the output of the component selection process. At this stage, possible sources of conflict and overlap among components

have been identified through a system designer's investigation. System designers have studied the component interfaces and identified the facts that are important to effective component assembly and evolution.

Adapted components are components that have been amended to address potential sources of conflict and services identified at the component selection process. They are the output of component adaptation, and are ready to be assembled in the new project. In general, the designers proceed on a trial-and-error basis to investigate the usefulness of the component in the new context.

Assembled components are components that have been integrated via some form of common infrastructure at the systems design or implementation stage. Once the system has been implemented and tested, it will be taken into use. Assembled status is thereby a relatively stable status; it will not change to the next status until the system or its sub-system is updated.

Updated components are components that have been updated by newer versions, or replaced by different components with similar behavior and interfaces. Components holding this status are mainly at the system maintenance stage. They are built when the previous ones cannot be easily integrated into the new project through adaptation. In principle a change to one component is easy to accomplish. However, changes may have extensive unforeseen repercussions on many other components in the system. The impacted components have to be identified by tracing different kinds of context dependencies, and corresponding changes have to be done as well.

These five statuses of components are extracted from the basic activities in the reuse process. Each status can be thought of as inheriting characteristics and benefits from the previous one, and as closer to reuse than the previous.

3.2.3 Summary

CBD as a vision and an approach has become the software concept *du jour*. It offers many exciting possibilities in terms of reducing application development costs, providing greater software reuse, and facilitating the maintenance and evolution of systems to meet new requirements. While it has not matured to the point where users can assemble applications freely, they are reshaping the ways groups design, build, deploy, maintain, and adapt enterprise applications. It is one tool in the bag, rather than a holistic and multifaceted approach (Allen 2002). To achieve the expected vision requires a number of hurdles to be overcome, such as deploying a "standard" interface specification language which is not only suited to all types of component, but also expressive enough to hide all properties that may lead to unanticipated interactions, searching for components from different resources, evaluating components, and maintaining component-based systems.

In current market, there have been a variety of CASE environments which (partly) support CBD. One of the CASE environments widely used in industry is Rational Rose (Quatrani 1997). A metaCASE environment that applies a component concept to support both method engineering and system engineering is difficult to find. MetaEdit+ is one example of an attempting to integrate the

concept of the component into a pre-exiting environment (Zhang 1997; Zhang 2000a; Zhang 2000b; Zhang and Lyytinen 2001; Zhang 2004; Zhang and Kaipala 2004).

3.3 Technical Support for Reuse

Among numerous software technologies, reuse has gained attention from academia, government, and industry over the last decade. Reuse has been recognized as a powerful means of potentially overcoming the software crisis (Tracz 1987; Basili 1989; Griss 1993). The number of technologies supporting software reuse has increased greatly. Different technologies have been proposed to achieve the goal of maximizing the reuse of basic components, of architectural design and even of software designers' experience in solving problems in specific contexts.

There are two principle types of techniques considered essential for reuse: techniques from the application developer viewpoint and techniques from the reusable asset developer viewpoint. The former concerns developing with the use of reusable assets (developing with reuse). These techniques mainly provide solutions for reuse at different stages of the ISD. The latter concerns developing reusable assets (developing for reuse). These techniques mainly provide solutions to developing and managing reusable assets. In general, we cannot strictly distinguish between these two types of techniques. A technique supporting reuse may consist of both developing for reuse and developing with reuse. For example, the product-line approach (SEI 2000) is a means of developing with reuse. At the same time, it provides the framework for building and managing a set of assets for reuse in a specific family of applications at the domain analysis stage. Because reuse is a process to leverage commonalities and variability between reusable assets, it is often domain-specific (Neighbors 1989).

Ezran *et al.* (1998) present and review the most relevant techniques in building and reusing software. Among them, *Object-oriented technology (OOT)* is regarded as an essential enabler for reuse (Judd *et al.* 1991). It provides methods and mechanisms for structuring models and program code to correspond to the objects found in the problem domain (Ezran *et al.* 1998). The designer identifies the main concepts in the application domain, and their responsibilities and relationships, and designs the application having in mind which objects can be reused. The OOT facilitates the reuse of code and higher levels of software artifacts to a limited degree. It is weak in describing the overall system structure and cannot bring about extensive reuse in ISD.

In addition, there is a set of techniques which have emerged from the OO world. They maximize the power of OOT in terms of reuse. The techniques are design patterns, application frameworks, and CBD. A *Design pattern* is a technique for documenting design solutions by describing the application context, the design problem, and a solution to the problem. The solution is customized

and implemented to solve the problem in a particular context (Gamma *et al.* 1995). It is an attempt to overcome the limitations of pure code reuse by emphasizing the importance of design reuse. It constitutes a promising attempt to shift the emphasis in software engineering away from component-based implementation towards component-based problem solving (Keller and Schauer 1998). *Application frameworks* and the *CBD* approach provide a framework which embodies a generic design, comprised of a set of cooperating classes/components that can be adapted to a variety of specific problems within a given domain (Cotter and Potel 1995). It combines code reuse with design reuse and offers specific services that are commonly used by a family of similar applications.

Furthermore, the concept of *software agents* has been proposed as an extension of the object model (Ezran *et al.* 1998). Compared with the object model or the component, software agents are integrated systems and their behavior is highly customizable, which enhance software reusability. In particular, the development of agent-based application frameworks is a promising approach to enhancing reuse.

Detailed descriptions and examples of reuse technologies are addressed in (Ezran *et al.* 1998). A discussion and comparison of the above methods is presented in Article 5. Therefore, we only provide a brief summary, as shown in TABLE 1.

TABLE 1 Comparison between reuse techniques (Ezran *et al.* 1998, 134)

Technology	Strength	Weakness
OOT	Enhances modularity and information hiding.	Requires significant modeling effort.
Design patterns	Facilitate retrieval of design solutions, provide guidelines for the development process.	Implementation from scratch.
Frameworks	Domain-specific semi-complete applications to be customized. Reuse of object model plus architecture.	Requires high expertise and deep understanding of the framework design.
Components	Domain-specific or technical. Development of an external market.	Not customizable.
Software agents	Highly customizable and adaptable, allow easy reconfiguration of complex systems.	Not yet mature and consolidated technology.

These techniques contain dependency on OOT. They offer partial (sometimes overlapping) views and solutions to better support for reuse in ISD. Among various kinds of reuse techniques, component-based reuse is the focus of this study. Below, we discuss several technologies that are closely related to the form of component-based reuse. These techniques support systematic reuse on the enterprise level.

3.3.1 Component Infrastructure Technologies

Although the CBD approach emphasizes the concept of the component throughout the development process, current CBD practices mainly concentrate on the code world where most reuse occurs, and component infrastructure development. The component infrastructure facilitates component reuse by providing capabilities and middleware solutions for connecting independent pieces of system functionality.

FIGURE 4 shows how developers move from an initial domain idea to a finished product in the CBD paradigm. Reuse occurs at each stage of the process, but the code stage provides most of the reuse facilities: the code component repository and component infrastructure with pre-defined component interfaces.

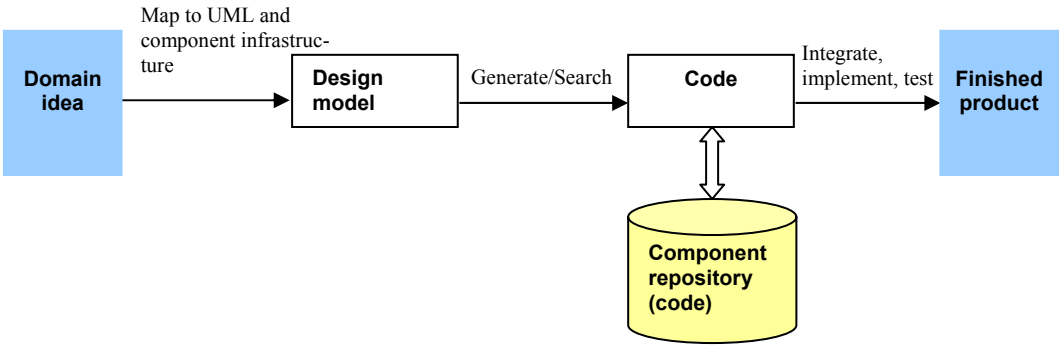

FIGURE 4 CBD: moving from domain idea to finished product

As in most ISD practices, developers first have to understand the problem domain and map the domain solutions onto design models that are represented in a specific ISDM notation, like the UML notation. Because the design models represent the implementation in code, the model developers have to take into account the component infrastructure defined in the project when mapping the domain solutions onto the design models.

A component infrastructure is responsible for proper execution of the design and implementation tasks on the project. A good infrastructure is a critical factor in the success of reuse and CBD (McClure 2001). In the world of component infrastructure technologies, there are currently different *de facto* infrastructures for software component specification, interoperability, and distributed computing. Currently, the three dominant infrastructure choices are:

Microsoft's Component Object Model (COM)/Distributed Component Object Model (DCOM) supports applications developed for Windows-based platforms using ActiveX components and tools (MSDN 2004).

SunSoft's JavaBeans specification defines a standard way to build Java applets, and takes advantage of the Java Virtual Machine (VM) to allow applications to be portable across any environments that support the Java VM (Sun 2004).

The Object Management Group's (OMG's) Object Management Architecture (OMA), specifies a Common Object Request Broker Architecture (CORBA) that uses an interface description language as the basis for brokering connections among applications written in different languages running on multiple distributed platforms (OMG 2002).

In addition, Microsoft's .NET platform represents the next stage in the evolution of COM. The .NET Platform essentially creates a component infrastructure for web middleware, using the component (software interchangeable part) as its basic building block and supplying these components with streamlined system and application services that integrate with the web (Hoagland 2003).

While these technologies are in many ways competitors, they share the same underlying characteristics. In particular, they provide the infrastructure required to manage and connect a disparate set of components operating on a distributed, heterogeneous platform. In fact, such technologies support access to remote services through both synchronous and asynchronous communication - in the presence of network failures - and with the variable network latency associated with internet- and intranet-based solutions. Large-scale, robust distributed business applications have been constructed using each of these infrastructure technologies in application domains as diverse as banking, telecommunications, and desktop publishing (Ezran *et al.* 1998).

Although these technologies have been successfully and widely used in industry, an obvious pitfall exists. The component infrastructures are based on the code world, and take the implementation stage as a starting-point to propagate reuse throughout the systems development life cycle, which may hamper reuse practice. Because the programming code is too abstract to understand, in order to better support code component reuse, other techniques like reverse engineering (Hall 1992) are applied to convert the information hidden in code to a higher level format, such as a design model. Thereby, design models are involved in the reuse process. Reuse is a practice that should be naturally and sequentially interleaved into the systems development process. In a natural manner, it should start from the systems analysis and design stage, instead of the code stage. In order to much more effectively propagate the reuse practice, the reuse strategy thereby should take the stage before the implementation stage as the starting-point to initiate and further perform reuse. As Neighbors (1980) stated the key to reusable software is captured in domain analysis in that it stresses the reusability of analysis and design, not code.

3.3.2 Domain Analysis and Engineering

Interest in domain analysis has increased significantly recently. It has been seen as the last of the technical stumbling blocks to achieve effective reuse. The definition of domain analysis formulated by Prieto-Diáz (Tracz 1991, 27) describes its purpose as follows:

Domain analysis *is a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems.*

Domain analysis is conducted as an iterative process that consists of four logical processes: identifying the domain, scoping the domain, analyzing the problem space, and designing the solution space (Sodhi and Sodhi 1998). It forms a systematic approach to the exploration of related systems to discover and exploit commonality, similarity, and variability. Through domain analysis, domain level requirements are analyzed, generic domain architecture is specified, and a set of assets specifying the commonality of related systems are identified and stored. Domain analysis is a part of the discipline of domain engineering.

Domain engineering *is a process creating an asset that can be managed and reused (Sodhi and Sodhi 1998, 61).*

Domain engineering is based on understanding the commonality and variability of systems in a given application domain. It consists of analysis and modeling of a domain, design of a generic architecture for a domain, implementing and leveraging reusable components that fit the architecture, and maintaining and evolving the domain and its components. Domain engineering uses business objectives and domain knowledge to create and standardize the architecture that the domain supports. It is notable that domain engineering is not a part of any one project – it cuts across all projects within the application domain, and forms a on-going effort to support multiple application engineering projects. Its goals are to identify, derive, organize, abstract, and represent the commonality and variability among assets within a particular domain. This further facilitates the reusability of a family of products within an organization's domain knowledge. Therefore, domain engineering has become the main trend of reuse in organizations.

Many companies and research institutes have moved away from developing software from scratch for each product and instead focused on the commonalities between the different products and capturing those in generic domain architectures and an associated set of reusable components. There are a number of reports on experiences of domain engineering practice, as well as studies on techniques and tools supporting domain engineering in an industry context. Brownsword and Clements (1996) report the experience of a Swedish naval defense contractor, CelsiusTech Systems AS, that has successfully

adapted domain models in building large, complex, software-intensive systems. Dikel *et al.* (1997) examine the success factors behind Nortel's newly created product-line architecture, an instance of domain models and architecture. Davis and Hawley (1994) describe the reuse capability that Boeing has developed as one of the prime contractors on the U.S. Advanced Research Projects Agency (ARPA) Software Technology for Adaptable, Reliable Systems (STARS) program. The underlying principle of the program is the separation of development into two separate life-cycle views: domain engineering and application engineering. During domain engineering, the Boeing/Navy STARS team developed a decision/question model over the application domain which supports the management and utilization of domain–specific reusable objects during application engineering. The research and case studies provide different solutions to domain-specific reuse. No matter which approaches are used, our view is that using the domain architecture model and associated components is an important way of identifying and managing commonality and variability in the product family. Consequently, it facilitates the institution of reuse and the chain reaction of reuse across different phases of development.

3.3.3 Domain-Specific Modeling

Domain-specific modeling (DSM) falls in line with the paradigm of domain engineering. However, as distinct from the technologies discussed above, SDM is based on the model world rather than the code world, leading to a direct mapping onto organizations' own domains in problem solving, design, and product implementation. It mainly supports reuse at the domain model construction stage. *Domain* is a problem space for a family of applications with similar requirements, a set of related systems with commonality. Examples of application domains include mobile phones, e-commerce platforms, point-of-sale systems etc. Due to the close relationship between applications within the same domain, the domain models can be reused many times within the family of applications. The concept of domain-specific modeling can be defined by adapting the description given by metaCASE Consulting (MetaCASE 2000, 4):

Domain-specific modeling (DSM) *is the process to understand the customer's requirements within the domain world and represent the requirements and possible solutions in the form of domain abstractions and semantics. It allows modelers to perceive themselves as working directly with domain concepts.*

FIGURE 5 shows how developers move from an initial domain idea to a finished product in the DSM approach.

The *domain model* captures in detail the domain-specific knowledge of the application in a form that leads to final implementation (Sodhi and Sodhi 1998). It relates directly to the application domain, and contains information about the domain abstractions and semantics. The domain model, therefore, is different from design models generated in traditional ISD paradigms. The design model is represented in a specific ISDM notation, like UML. The notation does not re-

late directly to the application domain but to the implementation, i.e. it visualizes the code (MetaCASE 2000). Hence, the domain idea and solutions must be mapped onto the design models representing the implementation in code, from which in general a relatively small percentage of the finished code can automatically be generated.

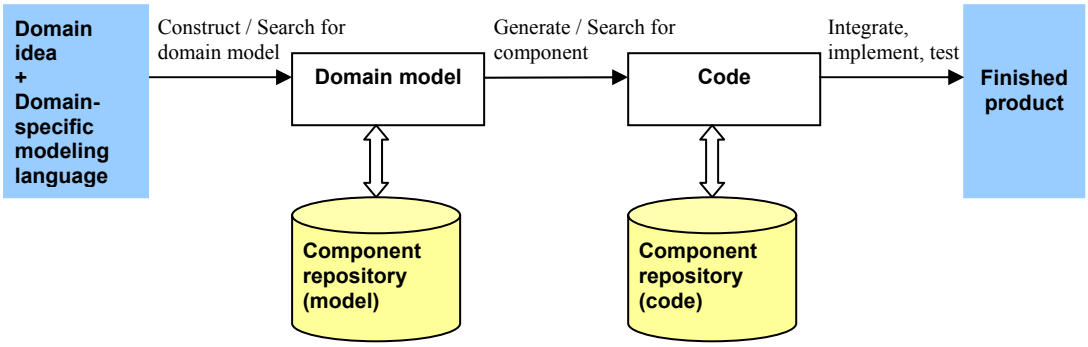

FIGURE 5 DSM: moving from domain idea to finished product

The difference between a domain model and a design model shows that, in the DSM approach, a great volume of information needed in the ISD process has been prescribed into the domain-specific modeling language. The information includes the architecture required to manage and connect a disparate set of domain models, syntax and semantics to construct a domain model and connect it to others, mappings from the domain models to the final implementation, and so on. Construction of the information belongs to the method engineering discipline, and will be done by method engineers, which reduces the burden on system developers. The domain-oriented method allows developers to concentrate on the required functionality and shift the focus from code to design. With the support of DSM, the developers can easily design systems within the same application domain by reusing pre-existing models. It improves product development speed significantly. Meanwhile, it is notable that the information is developed only once, and can be reused later in different projects.

The DSM approach has already been seen to work very effectively in a range of situations, most notably in embedded systems and product families (MetaCASE 2000). One example is the Nokia case study by MetaCASE Consulting (MetaCASE 1999). Nokia uses a domain-specific method to develop mobile phone software. A metaCASE tool (MetaEdit+ workbench) was used to model the concepts and rules of the mobile phone domain, its graphical notations, code generators and document generation templates. By using the domain-specific modeling approach, the benefits found in the case study are many: the domain-oriented method allowed developers to concentrate on the required functionality and to shift the focus from code to designs, results from code generation were more than fulfilled expectations, and the training time and cost

was reduced significantly. The result was a tenfold gain in productivity for the company (MetaCASE 1999).

3.3.4 Summary

There is a vital difference between an application's domain and its code (Jackson 1995). These are two different worlds, each with its own language, experts, ways of thinking etc (MetaCASE 2000). The technologies supporting reuse can therefore be divided into two groups according to their different views of reuse. Many reuse technologies focus on the code world, and take code reuse as a starting-point and the basis for reuse propagation, while recent research directly relates reuse to codified knowledge, such as the analysis pattern (Purao *et al.* 2003), the DSM approach (MetaCASE 2000), and the design pattern (Coad *et al.* 1995; Gamma *et al.* 1995; Purao 1998). These technologies attempt to exploit effective reuse in the form of codified generic knowledge or the application's domain world. They define groups of generic objects as analysis patterns to be reused in different domains (Purao *et al.* 2003), or package software engineering expertise with domain knowledge into domain models upon which more complex and more flexible systems designs can be built (Keller and Schauer 1998). These technologies take reuse of generic objects and domain models as the basis for influencing the remaining stages of systems development. They are distinguished from the concept of the traditional software reuse paradigm. Since generic objects and domain models represent high-level solutions to certain problems, they hide the detailed and complex implementation, and are easy for designers to understand and reuse. However, reuse of generic objects or design models requires that the environment provides a perfect mapping from generic knowledge to domain-specific knowledge or from the domain model components to the code components. Only in this way can the development team benefit from codified knowledge reuse during the systems development process.

Obviously, because of the overlap between different technologies, some techniques cannot work efficiently without combining other techniques. For example, a product-line approach (SEI 2000) follows the form of the domain engineering and CBD approach, which consists of the construction of software architecture for a specific domain and development of a complete set of reusable assets within the same domain. In the same manner, the middleware framework cannot support reuse if it is separated from the component repository.

Regardless of the diversity in the reuse technologies, the foundation on which reuse can be carried out is the collection of reusable assets. The management and retrieval of reusable assets/components thereby has captured the attention of the software reuse community (Prieto-Díaz and Freeman 1987; Maarek *et al.* 1991; Frakes and Pole 1994). Especially in the component-based reuse approach, libraries of components are necessary to achieve software reuse (Henninger 1997). Meanwhile, in terms of tool support and integration, there exists a fairly wide consensus that tools for reuse tasks should be integrated

seamlessly into CASE (Fischer 1987; Maiden and Sutcliffe 1992; Mili *et al.* 1995). Typical reuse functionalities like search should be available to developers, and should not distract them from their normal workflow. Broadly speaking, a reuse-oriented CASE environment should facilitate reuse activities at each stage of ISD.

3.4 Current Tool Support

Many software development organizations believe that investing in software reuse will improve their process productivity and product quality, and therefore are in the process of planning or developing a software reuse capability. Unfortunately, there is still little data available on the state-of-the-practice of utilizing or managing software reuse. The critical problem in today's practice of software reuse is a failure to develop the details necessary to support a valid software reuse infrastructure. Meanwhile, Morisio *et al.* (1999) report that the commitment of management, the existence of training, and effort to find a reuse approach that fits the context of the company are the keys to success. Although the management structure of an organization often determines the failure or success of reuse, such discussion is outside the scope of this thesis. Next, we present an overview of current reuse-supported tools and discuss the technical obstacles to reuse.

At present, there are many tools on the market claiming to support CBD and thereby reuse. Most of these tools support enterprise modeling, code generation, and round-trip engineering. The availability of these tools has led many application developers to consider CBD and reuse in developing large and distributed applications requiring robust operation. Below, based on the survey report of the COMBO project[1] we analyze some typical commercial tools. Because the COMBO project report (Hänninen *et al.* 2000) was finished in year 2000, the product information in the case of some tools has been updated in this thesis.

3.4.1 Tools Support for Reuse

Tools are examined in alphabetical order: after each tool name we give the web site for further information. Some tools are presented briefly due to inadequate product information. Meanwhile, the descriptions are not intended to cover each tool fully, but rather to illustrate various features that support reuse or CBD. The tools provide concrete examples of the current pervasive commercial or academic research in the field of reuse. After the tool descriptions, some important features will be summarized in a table.

1 The Combo project was a part of a development project of the Department of Computer Science and Information Systems in the University of Jyväskylä in term 1999-2000. The purpose of this course was to study tools for component management. For further information, please visit: http://projekti.it.jyu.fi/combowww/english/

MetaCASE Consulting: MetaEdit+ 3.0 (see: http://www.metacase.com)

MetaEdit+ is a metaCASE tool consisting of a method workbench for method specification and construction, and a full function CASE tool providing diagramming editors, browsers, generators, and multi-user support. MetaEdit+ has been successful as a (meta)CASE tool, with users numbering thousands (Kelly 1997). However, the current commercial version does not support component-based systematic reuse, although some ad hoc reuse can be done through property sharing. Also, reuse is introduced as the ability to include a design artifact from one graph in other graphs even in another graph from a different method, providing that the type of the design artifact is legal in the second method. This is made possible by the ability to reuse type-level components when defining the metamodel of a graph (Kelly 1997). Meanwhile, some facilities (like Info Tools) can be reused to support component management once the concept of a component is defined in this environment.

The metaCASE feature puts MetaEdit+ in a good position for reuse. The metaCASE functionality provides the flexibility needed to construct different ISDMs according to organizations' needs, which enables the domain-specific modeling approach or the product-line approach possible to be implemented and deployed within an organization that delivers a set of systems within the same application domain.

Introducing a component concept into MetaEdit+ that can systematically support reuse in a (meta)CASE environment is the goal of the next generation of MetaEdit+, as well as the objective of this thesis.

MicroTOOL: ObjectiF 5.0 (see: http://www.microtool.de/e_index.htm)

ObjectiF is an UML-based ISD tool where the main focus is on developing .NET applications. One promising feature offered by this tool is its round-trip engineering with ObjectiF and Visual Studio. That is to say every code window for a class in Visual Studio .NET gives engineers direct access to the corresponding ObjectiF class diagram, and vice versa: the designer can jump directly from an ObjectiF class diagram to Visual Studio .NET's code editor (MicroTOOL 2002). Components are depicted as packages in ObjectiF. A package contains a set of classes and their interfaces. A package diagram consists of packages and the relationship between packages, which forms the system architecture for system implementation and integration.

There is no specific emphasis on reuse in ObjectiF, but some functions provide good support for reuse. For example, the round-trip engineering eases the code understanding process, and thereby enhances reuse. The function of the import/export of packages provides a means of reuse of packages across different projects. However, the package management tool lacks specification, especially for package retrieval, which is a basic support mechanism for further package reuse.

Computer Associates: AllFusion Component Modeler 5.0 (Formerly available as: Paradigm Plus) (see: http://www3.ca.com/Solutions/Product.asp?ID=1003)

AllFusion Component Modeler is an UML modeling tool for visualizing, designing, and maintaining enterprise components for eBusiness. Through a common XML solution, it provides round-trip engineering, which keeps application design and implementation in sync through any number of code changes and design iterations (CA 2002). It also provides full XML support for model management. AllFusion Component Modeler also provides a flexible mechanism for integrated use with other enterprise modeling tools and enterprise integrated development environment.

AllFusion Component Modeler is a modeling tool rather than an enterprise integrated development environment. With XML support, the tool enables end-to-end integration with the AllFusion Modeling Suite and other technologies. It should be noticed that the modeling tool only supports UML notations, and there is no flexibility to revise the UML notations or construct a brand-new ISDM.

IBM: Rational Rose 98 (see: http://www.rational.com/products/rose/index.jsp)

Rational Rose provides solution to model-driven development with the UML. It is one of the leading visual modeling tools for object-oriented analysis, modeling, design, and construction through the application of UML notation. Rose 98 provides support for CBD including: component building, assembly, reuse, and component framework as well as a browser. Rose 98 expands the UML notation to support a modeling interface component called lollipop. Lollipops are an extension to the UML that represent COM components in application models. Rose 98 has additional specific functions to enable and simplify the assembling of components into complex applications or larger components. These capabilities help maximize the reuse of component models within the organization to build applications faster. Through its Frameworks browser, it also provides a cross-project component browsing capability.

UML representation of components is weak, and the tool does not specifically support the essential component modeling technique. As a result, the component interface cannot be defined with ideal precision (Brown and Barn 1998). Therefore, components in Rose 98 are limited to code, or executable components. Through component reverse engineering, developers can understand components represented by UML models. However, it would be improved if Rose 98 were to allow reuse already at the requirements analysis stage.

Select Business Solutions: Select Component Manager (see: http://www.selectbs. com/)

Select Component Manager is a part of Select Component Factory Version 4.4 released by Select Business Solutions. It is advertised as a highly scalable, active repository for software component publication, management, search and reuse. The tool is based on the code world in order to support CBD. It stores a wide range of components, including COM, CORBA, EJB, and wrapped legacy functions, and provides effective management of components, including version control, component interdependency management, access control, component files maintenance, and so on.

Select Component Factory is an integrated set of products for ISD. The environment provides support for software design (Select Component Architect), design review (Reviewer for Select Component Architect), service/component management (Select Component Manager), and code generation (Select Synchronizers). The software design tool provides support for design in UML notations.

3.4.2 Summary

After examining the tools that support reuse in different development phases, we can obtain some insights into the various ways in which current CASE tools support reuse.

It is easy to see that reuse is not the core technology supported by most of the tools, but each tool facilitates asset reuse in different ways. The most promising function offered by most of the tools (e.g. MicroTOOL's ObjectiF5.0, Computer Associates' AllFusion Component Modeler 5.0, and IBM's Rational Rose 98) is the support of round-trip engineering between the design models and the code. Through reverse engineering, the code component can be converted to a graphical diagram, which is more expressive and easy to understand, and thereby promotes code reuse. In addition, ObjectiF5.0 provides package import/export functions, which make it easy to reuse packages across projects. Select Component Manager, AllFusion Component Modeler 5.0, and MetaEdit+ provide solutions to component/model management, including version control, dependency management, component search, and component files maintenance, which are indispensable in the reuse process. Furthermore, some other functions promote and support reuse practice as well, like the design patterns provided by ObjectiF5.0, the support of system architecture design in ObjectiF5.0, Select Component Factory, Rational Rose 98 etc. Different tools provide support in their own ways. However, it is clear that no tools provide systematic support for reuse during the whole ISD process. Comparing the concept of a systematic reuse support environment with current reuse support tools, we can see that most of them provide incomplete support for reuse. Reuse is regarded as a side effect of different ISD technologies.

Meanwhile, we can easily see that the tools which (in)directly support re-use practice mainly facilitate the reuse of code. Although most tools emphasize

their model-driven development feature, they ignore the benefits of reuse at the design stage. No tools expand reuse practices from the code level to a higher level, e.g. the reuse of design models. Moreover, the DSM technique is not commonly supported in current tools. No tools support the construction of domain-specific modeling language, except MetaEdit+, which provides the meta-modelling feature. It is therefore difficult for them to support the reuse of domain models. Another factor is that current tools lack standards or uniform specification of components created at the systems analysis and design stage. XML can be regarded as a *de facto* interoperability model, which assures data exchange between design models. It has been used as a solution in current CASE tools; however, conversion support in CASE tools is not that widely successful.

4 RESEARCH PROBLEM DEFINITION

Having set out our conceptual framework for research, and examined the current state of the field and its issues, we now collate these issues and specify the research problem.

4.1 Technical Problems in Enabling Reuse

The main problem facing today's reuse practice is that it lacks support for reuse in the form of methodologies and supporting environments. No single technique, such as the technique for building and maintaining component libraries, is sufficient. As reuse is a common practice throughout the ISD process, an integrated methodology can provide proper and continuous guidelines and support for reuse practice in every phase of ISD. Meanwhile, development environments that promote efficient reuse are rare (Basili *et al.* 1992; Rine and Nada 1998), a problem which can also be traced back to the insufficient support of methodology. The problems can be specified in more detail by reference to the following perspectives:

P1: ***Limited understanding of components***
In general, the software industry limits the understanding of components in the source code or executable code. As seen in the survey of commercial tools supporting reuse, many of them only regard code as reusable assets, and ignore the reusable assets generated in other phases of ISD. This narrow understanding of components limits the scope of reuse. In theory, reusing assets generated at stages earlier than the implementation stage has greater potential leverage because of their greater expressive power. This can further trigger code reuse at the implementation stage. If we shift our focus from support tools to the state of the practice of CBD, a recent Cutter Consortium survey (Allen 2002), which gathered data on CBD from 118 companies from around the world, shows that, for many organizations, CBD is very much a programmer-related activity, with

70% reporting that programmers use GUI components. The survey also showed that the practice of component reuse in systems analysis and design had increased in at least half of the organizations polled. In order to incorporate component reuse into systems analysis and design, some research (Coad *et al.* 1995; Gamma *et al.* 1995; Fowler 1997) has been done regarding high-level component definition and reuse. However, we need an integrated methodology and a supporting environment which provides a means clearly to identify the various components that describe requirements, architecture, analysis, design, test, and implementation along the development chain. Clear identification underlies the ability either to reuse them or to allow them to be candidates for replacement.

P2: ***Insufficient methodical support for systematic reuse***
Current tools mainly provide *ad hoc* support for reuse practice. As summarized in the last section, most tools provide support for the CBD approach, which increases productivity and quality by reusing code component, while none take reuse as their mission. The tools lack support for systematic reuse which would help users understand the behavior of the asset, the context in which it was developed, and how it can be integrated into the application. The scope and benefits of reuse are thereby greatly reduced. In order to expand the benefits of reuse, an integrated methodology comprising both technical and non-technical issues of reuse should be integrated into the ISD environment (Jacobson *et al.* 1997). In particular, the methodology and its supporting tools should support the reuse process and reusable asset management, as specified in section 3.2.2.

P3: ***Inflexible support of modeling techniques in CASE tools***
Although modeling techniques do not directly affect reuse practice, their limited application domains decrease the number and the type of reusable assets. As can be seen in section 3.4, most CASE tools only provide "hard-coded" modeling techniques, like UML. Although UML has become the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, many organizations still need domain-specific techniques to describe the problem situation and the solutions to it. It is reported that 88% of organizations adapt the method-in-house (Hardy *et al.* 1995; Russo and Wynekoop 1995), and 38% of organizations have developed their own method (Hardy *et al.* 1995). It is difficult for a "hard-coded" CASE tool to provide a set of modeling techniques to cater for an organization's needs, which consequently limits the potential for reuse.

In sum, due to the insufficient methodical support for systematic reuse, the software industry at present cannot provide an ideal environment for the facilitation of reuse processes throughout the ISD process. To make up these drawbacks, one solution is to incorporate systematic reuse methodologies into systems development environments. The metaCASE environment is an appropriate environment for initiating such research, because it not only facilitates systematic reuse, but also provides flexible support for the specification of diverse ISDMs. Furthermore, due to its metamodeling capability, the metaCASE envi-

ronment can be specified and tailored to systems development in a specific application domain.

4.2 Research Problem Definition

Research into new methods and technologies continues unabated. Studies on how existing methods and technologies support reuse, however, are far too few (Zand *et al.* 1997). Existing methods such as OO patterns and CBD do not incorporate key learning from the reuse community (Zand *et al.* 1999). Reuse technology transfer seems to be very slow. Instead of endlessly building new methods and technologies, we need to study the more promising methods and processes that developers are using right now, work out ways of incorporating reuse technologies into the support methods, and find a method of evaluating the actual value of improvements.

Therefore, our research is aimed at developing a metaCASE environment which would support systematic reuse at different stages of ISD. On the basis of existing ISDMs and support tools, our goal is to incorporate reuse technology in the systems analysis and design environment at the component level. To achieve this goal we need a scientific approach to building and incorporating components into the systems analysis and design environment, as well as a scientific way of evaluating the approach. Our research tackles the main problems confronting existing support techniques for systematic reuse, as discussed in the previous sections. The research problem can thus be decomposed into the following three questions.

Q1: What is a generic conceptual framework that supports systematic reuse in a metaCASE environment?
Q2: What is a generic component model regardless of the semantics and the syntax design of component?
Q3: How does the conceptual framework and the component model support reuse in the ISD process, especially in the phases of systems analysis and design?

The questions are raised in line with the essential concepts contained in the model describing the research background. FIGURE 6 depicts the relationship between the research questions and the concepts addressed in this thesis. Our eventual objective is to develop IS in a time-saving, low cost, and high quality manner, as shown in the upper part of the figure. Hence, the research questions take into account the key issues regarding the technical support for systematic reuse, as shown in the lower part of the figure. In turn, the technical support for reuse will guide the ISD practice to achieve the eventual objective.

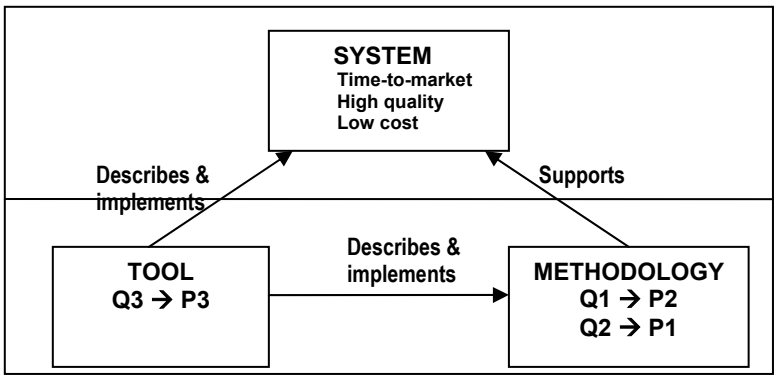

FIGURE 6 Relationship between research questions and the research background

In particular, Q1 reflects the problem of insufficient methodical support for systematic reuse. It inspires us to seek a possible direction to improve reuse practice. Answers to this question form the theoretical foundation of the whole research effort. Q2 is a follow-up question. It reflects the problem of our limited understanding of components, and indicates the need for additional research. Both questions focus on the methodology leading the research. Q3 is concerned with the flexible tool support. It raises the empirical question of how a component-based development methodology can be incorporated into a metaCASE environment, and how it influences the practice of reuse-based systems analysis and design.

4.3 Research Environment

This research has been carried out as part of the RAMSES research project funded by Tekes, the National Technology Agency, MetaCASE Consulting, and Nokia Mobile Phone (Korhonen *et al.* 2000). RAMSES is an acronym standing for Reuse in Advanced Method Support Environments. It is a three-year project that has been carried out in the Information Technology Research Institute at the University of Jyväskylä. Its goal is to enhance and develop automation and design support for component reuse in large-scale systems design environments. The project studies component reuse tools in an industry strength metaCASE environment called MetaEdit+. Current MetaEdit+ provides tools for environment management, model editing, a repository browser, and a method workbench. Although some tools in MetaEdit+, like the browser facility, support component reuse to some extent, systematic reuse support is insufficient. In particular, the component definition is not clear on either the type level or instance level, which hinders reuse. The weaknesses of the current system thus form the context for our study.

5 RESEARCH METHODOLOGY

Having identified the research problem and described the environment in which the research takes place, we can now move to selecting a research methodology that will direct and describe how we can address the research problems. The underlying theoretical approach is the information systems design theory (Walls *et al.* 1992).

5.1 Choice and Description of Methodologies

Different research methodologies are needed in the area of IS, including laboratory and field experiments, case studies, phenomenology methods, action research, and so on. The selection of a methodology depends on the nature of the research work to be carried out. Our studies on systematic component reuse in metamodeling-based systems analysis and design fall into the discipline of method engineering and software engineering. The research work includes constructive processes. We thereby consider systems development as one of our research methods. As pointed out in Nunamaker *et al.* (1991), systems development and other research methodologies are complementary. An integrated multi-dimensional approach will generate fruitful research results. The multi-methodological approach integrates four stages: observation, theory building, systems development, and experimentation.

Observation includes empirical methodologies such as case studies, field studies, and sample surveys that are unobtrusive research methods. It helps researchers to formulate specific theories and hypotheses to be tested.

Theory building includes development of new ideas and concepts, and construction of conceptual frameworks, new methods, or models. Theories can suggest hypotheses, guide the design of experiments, and conduct systematic observations.

Systems development is a constructive process consisting of stages like concept design, constructing the architecture of the system, prototyping, product

development, and technology transfer. Systems development is the hub of research that interacts with other research methodologies to form an integrated and dynamic research program.

Experimentation includes research strategies such as laboratory and field experiments. Experimentation is the process engaged in to validate/confirm the underlying theories. It leads to refining the theory or improving the system.

In our study, we reuse the multi-methodological approach suggested in (Nunamaker *et al.* 1991) as a general guideline and tailor it to our research context, as shown in FIGURE 7.

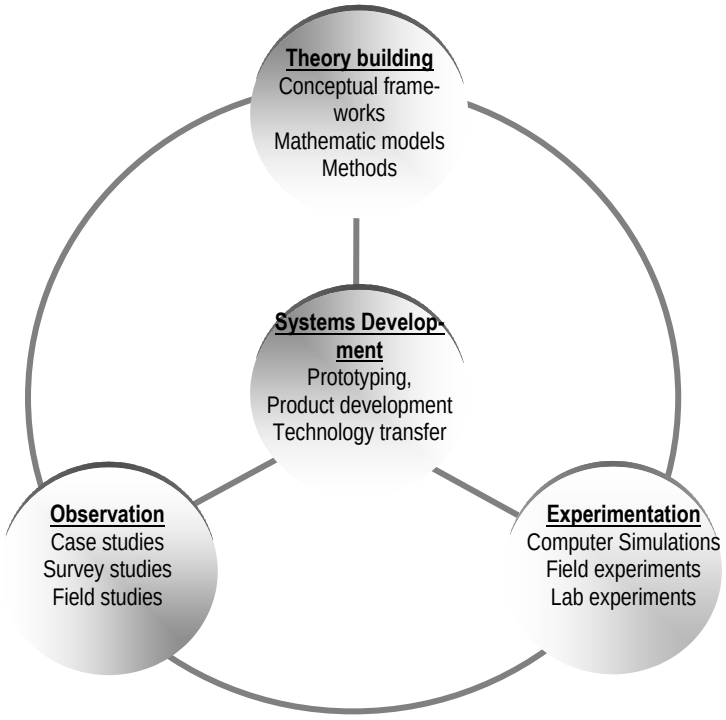

FIGURE 7 A multi-methodological approach to IS research (Nunamaker *et al*. 1991, 94)

5.2 Application of the Methodology in this Research

Observations were used at the beginning of our research work to obtain an overview of the state of the art of the support given to reuse in a systems development environment. By reviewing the literature, conducting interviews with CASE tool users and project managers, and surveying the tools that support reuse and CBD[2], we collected general ideas about the current situation of reuse support in the area of CASE and metaCASE tools. We also found problems, and posited a set of research questions to direct our subsequent research.

2 The interviews and survey were carried out by the COMBO student project.

Theory building is an important stage in our research design. It is necessary to obtain insights, evaluate the impact of the research, and then proceed to systems design. The development of a systems design theory that supports CBD forms the main contribution of the thesis. On the basis of the knowledge collected during the observation stage, we present a systematic reuse architecture in the metaCASE environment. The systematic reuse architecture, covering different stages in the ISD process, studies reuse possibilities and types of reuse from both the metamodeling (method construction) and modeling (systems development) aspects. The conceptual framework is intended to serve as a standard for practising systematic reuse in a metamodeling-based systems development environment. It identifies different types of reuse practice in both the product and method development life cycle; specifies a set of activities that enable implementation of the reuse practice; defines the component model, especially the model for wrapping up the design models as components; and proposes a hypertext-based component context representation approach to promoting communication between engineers.

Development is concerned with the systematic use of scientific knowledge directed toward the production of useful materials, devices, systems, or methods, including the design and development of prototypes and processes (Blake 1978). Systems development in our research can be thought of as a "proof-of-concept" approach. At this stage, a prototype of component deployment in systems analysis and design has been built over MetaEdit+ by metamodeling the component model and its repository, as well as developing tools facilitating component search and management. The prototype follows the principles envisioned at the theory building stage. They demonstrate the facilities for generating and managing components, and further support the reuse process.

Experimentation is carried out at the final stage of our research effort. We performed an initial study for a laboratory experiment designed to analyze the influence of component deployment in the phases of systems analysis and design. The experimentally tested hypotheses involved a number of quantitative, objective, and unobtrusive measures of the efficiency and effectiveness aspects of component deployment in systems analysis and design. The results reveal several statistically significant differences between the component-based systematic reuse approach and the normal object-oriented analysis and design approach, and thus serve as a preliminary test for validating and refining the conceptual frameworks for component-based reuse in a metamodeling-based systems development environment.

6 SUMMARY OF THE ARTICLES

In this section, I list the five articles that make up the body of the thesis, along with brief descriptions of the problems addressed and the results of each. The publication details and authors are listed for each article, and the division of work among the co-authors is described.

It should be noted that the thesis is made up of five separate articles published or submitted for publication. Some overlap is thus unavoidable. The repeated elements mainly concern the description of the research environment, e.g. the metaCASE environments and MetaEdit+.

Before presenting the details of each article, I briefly describe how the research questions, research methodologies, and thesis articles are organically integrated. TABLE 2 provides a brief summary of the research questions and their treatment.

FIGURE 8 illustrates the contents and the main contributions of each article. Together, the contributions can be seen as a systems development project.

This thesis is in two parts. Part 1 is an introduction to the thesis. It is regarded as the requirements stage in a systems development project. It studies the history and the state-of-the-art in ISD, ISDM, the reuse techniques, and the support tools on the basis of the literature review and survey (observation). It analyzes the current needs for the support of systematic reuse in an ISD environment, and reveals the limitations of current practice. The analysis generates some ideas for building a conceptual framework for systematic component reuse and provides our three research questions: a conceptual framework supporting reuse, a generic component model, and the application of the component model, on the basis of which the research methodologies are proposed.

Part 2 consists of 5 articles. Articles 1 and 2 provide a generic conceptual framework, which forms the design stage of a systems development project.

TABLE 2 Research questions and their treatment

Research question	Research methodology	Article and Contribution
Q1: Conceptual framework	Observation and theory building	Article 1 – A generic framework supporting systematic reuse in a metaCASE environment)
Q2: Component model	Observation and theory building	Article 2 – A generic component model in a metaCASE environment Article 3 – An extension of the component concept specification in the component model Article 4 – An extension from self-contained component reuse to components' contextual and hypertextual representations
Q3: Application of the conceptual framework and the component model	Prototyping and experimentation	Article 3 – Examples of the extended component concept specification Article 4 – A prototype of the hypertext model for component context representation and reuse in a high-level mobile phone design scenario Article 5 – A laboratory experiment to study component deployment in systems analysis and design

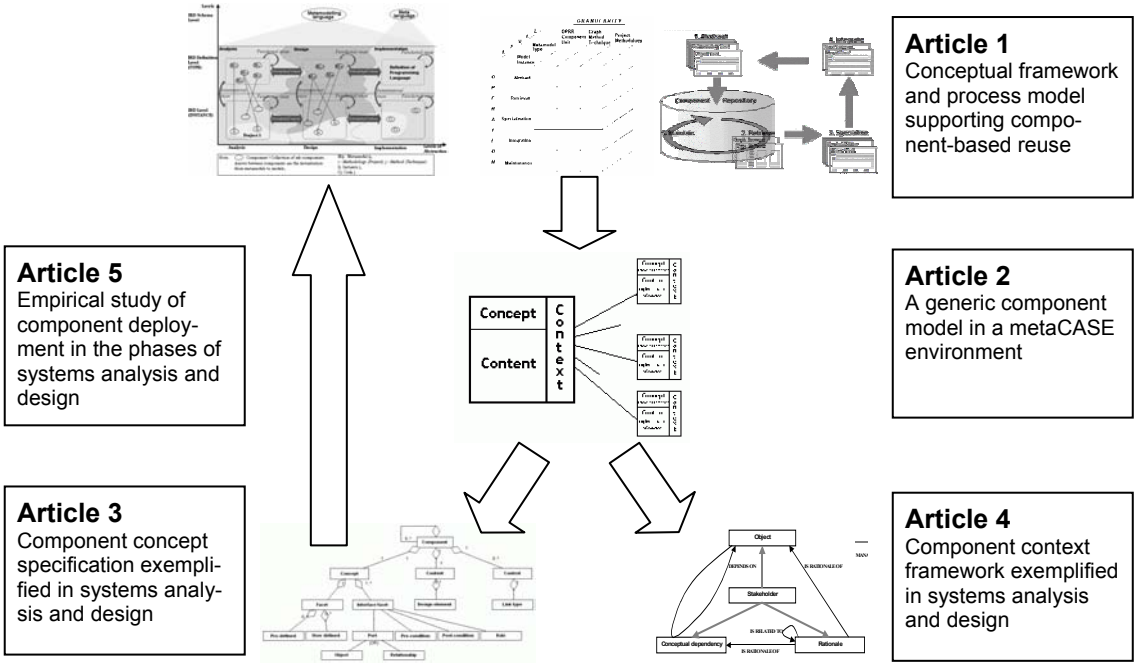

FIGURE 8 Contribution of each article

Article 1, A Framework for Component Reuse in a Metamodeling-Based Software Development (Zhang and Lyytinen 2001), is motivated by the needs of reuse support in a metamodeling-based systems development environment, as stated in the introduction to this thesis. This article thus builds theories to systematically support reuse in the processes of method specification and systems design by means of component support (theory building).

Article 2, Defining Components in a MetaCASE Environment (Zhang 2000a), follows the framework presented in the previous article, overviews the current component structure, and builds a generic component model for a metaCASE environment (theory building).

Articles 3 and 4 discuss the main elements in the component model: component content and component context. Each article provides more detailed specifications of the individual components on which the implementation of the components for systems analysis and design is based. It forms the implementation stage of a systems development project.

Article 3, Component Modeling for Systems Analysis and Design (Zhang and Rossi 2002), extends the conceptual framework from component representation and usage. Specifically, it specifies the component concept (interface) which can be deployed in the systems analysis and design phases. It also implements a prototype in our research environment, MetaEdit+ (prototyping), and shows the use of the component prototype on the systems design level.

Article 4, Component Context Specification and Representation in a MetaCASE Environment (Zhang and Kaipala 2004), builds the component context framework (theory building) and demonstrates its implementation (prototyping). The article shows how the component context prototype is unique and works differently in the systematic support of reuse: it provides a brief overview and comparison of the tools/techniques that facilitate component reuse with respect to corporate knowledge, such as the design rationale and diverse conceptual dependencies between components.

Article 5, Component-Based Reuse in Systems Analysis and Design: An Exploratory Study (Zhang 2004), wraps up the project by testing the implementations. It is an exploratory study of component deployment in systems design. This article not only demonstrates how the concept of the component is applied in systems analysis and design through its experimental design, but also partly[3]

3 We use the word "partly", because the experiment is an exploratory study of component deployment in systems analysis and design rather than a study of the entire conceptual frameworks of component-based reuse in the metamodeling based systems development environment.

verifies the conceptual component model and the systematic reuse architecture built in our previous research effort.

6.1 A Framework for Component Reuse in a Metamodeling-Based Software Development

Zheying Zhang and Kalle Lyytinen

Published in Requirements Engineering Journal, 6 (2), 2001, 116 – 131.
An early version was published in S. Brinkkemper, E. Lindencrona, and A. Solvberg (Eds.), Information Systems Engineering: State of the Art and Research Themes, London: Springer-Verlag, 107-122.

Systematic reuse is generally recognized as a key technology to improve the software process (Mili *et al.* 1995). However, current (meta)CASE tools are not powerful enough to offer systematic support for reusable asset abstraction, selection, adaptation, integration, and maintenance throughout the life cycle of a project. It is an impediment to the accelerating time-to-market demands of ISD. Moreover, even in the so-called reuse-supported environment, there is no clear distinction between different types of reuse situation, the diverse types of reusable information, and the corresponding support facilities. Therefore, we collected the activities needed in both the method specification and systems design process, and develop a holistic framework to systematically support the reuse process in a metaCASE environment.

The aim of the article was to suggest a component-based reuse framework that can address issues related to design artifact and method component reuse in the life cycle of systems development. In particular, the article seeks to demonstrate how reuse "ideas" can be implemented in an industry-strength environment, MetaEdit+. Our strategy for meeting these goals is the following. We first develop a general framework for metamodeling-based component reuse. This framework considers reuse from the perspectives of a systems development life cycle, modeling levels, reuse situation types, component granularity, and reuse activities. The framework is then used to analyze support functionality within a metaCASE environment, and to suggest how reuse activities can be integrated into method engineering processes and associated tasks of defining development processes and their technical facilitation.

The conceptual framework forms the theoretical precursor to studying and implementing extensive component reuse on the basis of a metamodeling concept. The framework differs from other suggested software reuse frameworks in that it introduces reuse based on metamodeling and requires support from a metaCASE environment. Meanwhile, it expands the reuse process by shifting the focus from reuse of code to reuse of domain knowledge and conceptual abstractions. Thus the framework not only focuses our attention on new

technical challenges posed by the software reuse process, but also acknowledges the organizational challenges implied by such an environment.

6.2 Defining Components in a MetaCASE Environment

Zheying Zhang

Published in B. Wangler and L. Bergman (Eds.) Advanced Information Systems Engineering: 12th International Conference, CAiSE 2000, Stockholm, Sweden, June 2000, LNCS 1789, Heidelberg: Springer-Verlag, 340 –354.

The current practice of reuse is based on the code world. Although a component can be considered as an artifact generated at any stage of the ISD process, reuse of different types of component is rarely supported in (meta)CASE tools. The study of components mainly focuses on the code level. Because metaCASE environments can be used for both methodology specifications and methodology supported systems design and implementation activities, they manipulate diverse artifacts. These artifacts and their embedded knowledge form good sources for reuse. It is feasible to introduce a generic component model into a metaCASE environment.

In this article, I extend the component reuse framework by defining a generic component model in the metaCASE environment. I first study the features of design artifacts, and the possibility of packaging them into components. After that, I define the component structure on the basis of three perspectives: concept, content, and context. The underlying principle is to offer a comprehensive description of the component concept, to extensively illustrate diverse dependencies between components and other objects, and to avoid information loss and repetition. Accordingly, a hierarchical facet-based schema is defined to represent the component concept. Different types of link are proposed to represent the conceptual relationship between components and the environment, including reuse dependency, the usage context, and the implementation context. A complete presentation of the component context can be found in Article 4 (Section 6.5). On the basis of the component structure definition, different artifacts, including the design artifacts in methodologies, can be used as components after the necessary re-specialization.

Although the recent wide-spread emergence of the component concept together with component-based reuse in systems development has meant that the component has attracted increased attention in industry as well as academia, component-based reuse in metaCASE environments is new. This is the first attempt to explicitly define a generic component model for diverse artifacts generated in a metaCASE environment during the systems development process. The next step in our research is to seek and prove the feasibility of the component model in the systems development process, especially in the systems analysis and design phases.

6.3 Component Modeling for Systems Analysis and Design

Zheying Zhang and Matti Rossi

The ICSR7 2002 Workshop on Component-based Software Development Processes, April 15-19, 2002, Austin, Texas, USA.

When any manufacturing process evolves to the point where it can be based on pre-built components and subassemblies, product quality, quantity, and speed of delivery soar. This principle applies equally to ISD, allowing unprecedented quality, speed of development, and highly effective change management. However, a fundamental change of mindset toward components is necessary to usher in the industrial era of ISD, although the component concept has attained maturing in manufacturing industry. There is no industry consensus on the definition of a component, aside from some agreement on the properties that a component is expected to have (McClure 2001), which hampers the propagation of component deployment. The component concept should be improved to form an integrated part of an ISDM that supports reuse-oriented development practices.

In the earlier article we sought to develop a generic component model for different types of components in the metamodeling-based systems development environment. In this article a set of detailed specifications are added to the component concept (interface). It follows the reuse pattern on the model level, as describe in the earlier conceptual framework in Article 1 (Section 6.1), and aims at a consistent and technology-independent component concept to support component-based reuse in ISD. We elaborate on the component concept (interface) specification to enhance the component model. The objects are the interim artifacts generated at the stage of systems analysis and design. We study how to provide effective and adequate information to component users. We first examine the pitfalls of the current component structure, and then expand the component concept by providing more practical information for component use, including the extensible component facet descriptor and interface elements.

It is distinct from other methodologies that support component-based development. Our focus is on the methodical support of component deployment. We thereby enhance the component concept specification for systems analysis and design in the context of a metaCASE environment. Based on the support of metamodeling techniques, the generic component model can be instantiated as a methodology-specific component model by combining the generic model with a specific ISDM, like SSAD (Yourdon 1989), UML (UML 1995), the in-house method (Tolvanen 1998), or domain-specific methodologies (MetaCASE 2000). The methodology-specific component model can thereafter be used to define components out of the design artifacts specified in the same methodology. The component prototype can be supported at repository and tool level within the metaCASE tool. It improves the design artifact reuse within a specific organiza-

tion and project. Since the component model is based on the original methodologies deployed in an organization, we believe that it is easy to adopt and deploy. The methodical support feature of the component model further enhances the possibilities of incorporating components and reuse practice into mainstream ISD practice already during the early stages of ISD.

6.4 Component Context Specification and Representation in a MetaCASE Environment

Zheying Zhang and Janne Kaipala

To be submitted to Information and Software Technology for possible publication.
An early version was published in M. Khosrow-Pour (Ed.) Information Technology and Organizations: Trends, Issues, Challenges and Solutions, Proceedings of the 2003 Information Resources Management Association International Conference (IRMA2003), Philadelphia, PA, USA, May 18-21, 2003, Hershey, PA: Idea Group Publishing, 712 –715.

The lack of design information forms a significant barrier for system developers to develop and reuse a component. In order to develop and reuse a component, we need multiple forms of contextual knowledge, which includes the domain description, the goals and strategies used to define the component, the arguments supporting decisions, and various dependencies among components being designed and implemented. This covers a very large volume of information. Some can be intuitively acquired from the interface description, but some is hidden behind the component definition and reuse process and difficult to retrieve. In practice, especially when the number of components goes up, it is excessively costly to collect and systemically manage all the contextual knowledge without a comprehensive framework and tool support.

The article addresses these deficiencies in the ISD environment, especially the CBD environment. It examines the current state of tool support for component reuse and the collection and management of contextual knowledge. The literature review reveals a rather narrow understanding of the concept of reuse depending on the adopted approach. Many tools provide support for reuse only on the code level, and lack support for a systematic process utilizing contextual information. The article seeks to extend current theory and practice with the development of the concept of the component context, describes the hypertext data model and its supporting tools, and exemplifies the representation of the component context by using the hypertext tools in MetaEdit+. The example illustrates how hypertext techniques assist system designers by creating conceptual dependencies, recording annotations, and capturing debates in their design activities.

Research into the component context and its hypertext implementation is unique in three ways.

First, we increase understanding of the component context from the perspectives of conceptual dependencies and rationales in the ISD process, especially in systems analysis and design phases. This is the first attempt at a study of how to enhance the reuse of design artifacts by reusing embodied contextual knowledge.

Second, we integrate the implementation of context representation into a metaCASE environment. The metaCASE environment provides flexible support for method construction, including specification of the appropriate component modeling concepts and notations in line with a specific methodology. To our knowledge, no tools yet exist that provide both metamodeling facilities and the hypertext tools for representing the component context.

Third, aside from those design artifacts which can be specified as reusable components, we figured out a way to "reuse" the corporate design knowledge embodied in any design artifacts or the design process through the presentation of the component context. Obviously, it is not easy to physically reuse the component context. It can only be reused to make inferences or draw generalizations after users have understood the concept.

By building the component context framework, the component model proposed in Article 2 (Section 6.2) is finally wrapped up. We exemplified the use of the component concept and context in different articles, but still need the empirical evidence to assess the efficiency and effectiveness of component deployment in ISD, especially at the stage of systems analysis and design. The empirical study is addressed in the following article.

6.5 Component-Based Reuse in Systems Analysis and Design: An Exploratory Study

Zheying Zhang

To be published in the Proceedings of the 11th European Conference on IT Evaluation, Royal Netherlands Academy of Arts and Sciences, Amsterdam, 11-12 November 2004.

The component is the kernel of the component-based approach to ISD. Besides providing implementation support, the component, theoretically, is the pillar of systems analysis and design in terms of concepts, processes, and methods. In order to expand the use of the concept of the component in the practice of ISD, a component-based approach should provide consistent and technology-independent component concepts and definitions, proper component modeling concepts and notations, and mechanisms for systems analysis and design in a component-oriented manner. Our preceding research had built the conceptual theory to tackle the above issues, an empirical study, however, was still needed to further verify and validate our preceding work.

In this article, I describe an exploratory study for an empirical experiment, which would compare the CBD approach in systems design with the traditional object-oriented systems design approach. The study serves as a preliminary test of the preceding research results, including the conceptual framework of component-based reuse and the component model. Our hypothesis was that component deployment in systems design would help to decrease design time and increase design quality in terms of completeness and correctness. The experiment tests the hypothesis by analyzing the empirical data quantitatively and qualitatively.

From a pilot group of 13 users, I obtained results that support the hypothesis: there exist statistically significant differences in design cost and quality between the CBD approach and the normal object-oriented design approach. The tentative results from this initial empirical study confirm the usefulness of component deployment in the early stages of the ISD process. Meanwhile, the evidence indicates that building the design architecture, defining components, and reusing components at the analysis and design stages are long-term strategies for systematic reuse in the ISD process, especially suitable for systems within the same application domain. Such an approach is much more beneficial to a large organization which delivers systems in one or several system families than a small one producing project level products for an individual system or application.

6.6 About the Joint Articles

My contribution in the joint articles (Article 1, 3 and 4) is as follows:

In the first article, A Framework for Component Reuse in a Metamodeling-Based Software Development, the division of work was largely even. I wrote the whole article as a draft. I was mainly responsible for building the architecture and framework for component reuse in a metaCASE environment (section 3 and 4), and describing the systems development process as five patterns (section 5).

In the third article, Component modeling for Systems Analysis and Design, I was responsible for introducing the component model (section 2), presenting the improved component structure in a metaCASE environment (section 3), illustrating the component interface elements and examples (section 5), and bringing together the article as a whole.

In the fourth article, Component Context Specification and Representation in a MetaCASE Environment, the division of work was largely even throughout. We collectively built the framework of the component context and the scenario to exemplify the use of the component context, but I was responsible for the conceptual dependency part in the framework building and the scenario design.

7 CONCLUSION

The contribution and limitations of this thesis are presented in this concluding section.

7.1 Contribution of the Thesis

The main contribution of this thesis lies in the exploration of the possibilities for developing the CBD approach in a metamodeling-based ISD environment. I propose remedies for the deficiencies of current CBD practices, and conduct an exploratory empirical study to partly verify and validate the proposed solutions. In particular, I examine the possibilities and need for extensive component reuse on the basis of metamodeling concepts and build a framework that systematically guides the evolution of metamodeling-based reuse. In order to better support reuse activities, I further identify current deficiencies in the deployment of the component concept, clarify the concept of the component by using an elaborate component model based on the concept of metamodeling, and conduct a laboratory experiment to empirically study the influence of component deployment at the analysis and design stages on the whole ISD process. Each of these studies is addressed in one or more articles, providing a deeper examination of the problems, directions for an overall solution, components of specific solutions, and examples of their applicability.

It is distinct from other reuse frameworks (see Biggerstaff and Richter (1989), Moore (1991), Krueger (1992), SRI (1995), and Liao *et al.* (1998)). Our component reuse framework introduces component reuse based on the concept of metamodeling and requires support from a metaCASE environment. Hence the reuse process consists of both a methodology construction process and a systems development process. Also, it is expanded by shifting the focus from simple code reuse to the reuse of artifacts generated throughout the ISD process. Reusable components are thus extended from the source code to any type of artifact generated during different phases of the ISD process. At the same

time, the framework classifies reusable components into varying granularity levels according to the data types and semantics defined by the metaCASE environment. Such a classification presents reusable components in a comprehensive manner and systematically supports component management. Furthermore, taking into account the character of reusable components and the context for reuse, the framework distinguishes type of reuse in terms of conceptual reuse, functional reuse, and instantiation reuse.

In addition, the component structure is defined by means of metamodeling. It expands a 3C model (Tracz 1990) to present components in three perspectives: concept (interface), content, and context. Our aim is to enhance the support of the component concept and representation of the component context in a metaCASE environment, so that user can easily understand the services embedded in the component and its usage. The main difference between our approach and most other approaches to components is that the deployment of component is supported at repository and tool level within the metaCASE tool. The component model is a generic model independent of any ISDMs and their supporting environments. By combining it with a specific ISDM by means of metamodeling, the component model is added to the metaCASE environment. Thereafter, the metaCASE environment facilitates component definition, storage, search, adaptation, integration, and maintenance. The hypertext prototype in MetaEdit+ provides the means for component context creation, representation, retrieval, and management. To our knowledge, no such environment yet exists that systematically supports component-based reuse. The deployment of the component model makes MetaEdit+ the first environment to allow component-based reuse practice in a metaCASE environment. Because this research was conducted by means of a generic view of the metaCASE environment, its results can be easily expanded and generalized to other metaCASE environments.

The applicability of the concept of component reuse in systems analysis and design is demonstrated and analyzed through a CBD approach in the laboratory experiment. The experimental evidence indicates that the CBD approach at the stages of systems analysis and design shortens design time and improves design quality. It forms a long-term strategy for systematic reuse in ISD process, especially for systems within the same application domain. Specifically, the metamodeling feature of the metaCASE environment facilitates the specification of the design architecture, component definition, reuse, and management-related activities, which are indispensable elements of the CBD approach. Thus, the CBD approach, empowered by its supporting metaCASE environment, systematically supports the reuse practice and improves design efficiency and effectiveness.

The contributions of this study are thus twofold. First, the conceptual framework built via the literature review, survey, and interview provides a generic blueprint of systematic reuse based on the concept of metamodeling. The component model and its prototypes implemented in MetaEdit+ by means of metamodeling support the systematic reuse process within the metaCASE environment. Second, the empirical study carried out in a laboratory experiment

validates our hypotheses of the applicability of the CBD approach, and further demonstrates that great benefits with respect to cost and quality can be gained by conducting the CBD approach in a metaCASE environment.

7.2 Limitations of this Study and Directions for Further Research

Due to the immature stage of the component-based reuse supported in the metaCASE environment, there are weaknesses in current study. The study lacks an extensive implementation of reuse tools that support systematic reuse in the metaCASE environment. Hence, the outcome of the conceptual framework is presented in part as a preliminary result of the empirical study in this thesis. When the prototype is finally implemented in industry settings, follow-up studies can be carried out in a set of field experiments, where professionals will be recruited as the experimental participants. At that time, our conceptual framework will be validated and strengthened, and then be of greater value at the level of industrial practice.

Meanwhile, because the starting point of our study was support for component reuse in the metamodeling-based development environment, our focus was to specify a generic component interface by means of metamodeling. The study lacks an overview of the current state of component interface specifications (interface definition language) for the code component as well as different techniques used in component interface specifications. Hence, there is no comparison between the component metamodel and other component interface definition techniques. This is a weakness of the study.

In view of the limitations of the thesis and our current research work, I propose that future studies in component reuse in MetaEdit+ should deal with four categories: further implementation and empirical studies on the component engine, formalization of the component semantics, further research on component-based reuse in the context of a specific domain, and componentization in the method engineering process in the metaCASE environment.

From the viewpoint of implementation, besides the definition, search, and hypertext-based component context definition and navigation tools, we will continue focusing on the exiting tools and seeking the necessary functionalities that better support component reuse and management. The reuse tools should include a complete component repository for management, and search and retrieval. It would also be instructive to carry out a laboratory or field experiment for the use of a reuse support function integrated within MetaEdit+. This would provide valuable information to validate the usefulness of reuse tools.

From the viewpoint of formalization, there are several approaches to component formalization. The possible mathematical domains are logic theory, set theory, and category theory (Hofstede and Proper 1997). Z notations (Spivey 1992), for example, can be used to express the semantics of component struc-

ture. Z is the underlying formal notation because of its maturity as a formal specification notation and its well-known underlying mathematical concepts, such as set theory and predicate logic. Through the formalization approach, we could study the semantic foundations of component structure and reuse support more closely, and establish a sound basis for sophisticated automated support for reuse processes. In particular, the approach would make rigorous analysis of semantic properties captured by the component model possible.

From the viewpoint of metamodeling, we have not conducted enough research on the deployment of the component concept in the metamodeling process. This is challenging because the practice of method engineering is not in as flourishing a state as the practice of software (systems) engineering. If there is no big demand for reuse in the metamodeling process, study of component-based reuse on the metamodeling level will lack justification support. However, it is an interesting topic and provides us a possible avenue for our future research.

One interesting path to follow up is to apply the metamodeling-based component reuse approach in a specific application domain. The results of our empirical study indicate that the metamodeling-based component reuse approach applied within a specific application domain is more beneficial than one that delivers different products in different application domains. Within the same application domain, the metaCASE environment does well in domain-specific modeling (MetaCASE 2000). By combining the CBD approach with a domain-specific modeling language, organizations can map the component model onto the domain-specific conceptual specifications and guidelines of the product life cycle that is being developed or acquired to establish its ability to support systematic reuse practice. We believe this to be one of the very few areas where significant contributions can be expected in addressing the productivity and quality problems of ISD.

REFERENCES

Allen, P. 2002. CBD Survey: The State of the Practice, Cutter Consortium. URL: http://www.cutter.com/research/2002/edge020305.html. Access date: May 20, 2003.

Arango, G. & Prieto-Diaz, R. 1991. Domain Analysis Concepts and Research Directions. Domain Analysis and Software Systems Modeling, Los Alamitos, CA: IEEE Computer Society Press.

Avison, D. E. & Fitzgerald, G. 1995. Information Systems Development: Methodologies, Techniques and Tools, Berkshire: McGraw-Hill International (UK) Limited.

Banker, R. D. & Kauffman, R. J. 1991. Reuse and Productivity in Integrated Computer-Aided Software Engineering: An Empirical Study. MIS Quarterly 15(3), 375 - 401.

Basili, V. R. 1989. Software Development: A Paradigm for the Future. In Proceedings of the Thirteenth Annual International Computer Science and Applications Conference, Los Alamitos, Calif.: IEEE Computer Society Press, 471 - 485.

Basili, V. R., Briand, L. C. & Melo, W. L. 1996. How Reuse Influences Productivity in Object-Oriented Systems. Communications of the ACM 39(10), 104 - 116.

Basili, V. R., Caldiera, G. & Cantone, G. 1992. A Reference Architecture for the Component Factory. ACM Transactions on Software Engineering and Methodology 1(1), 53 - 80.

Basili, V. R. & Rombach, H. D. 1988. Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment, Tech. Report CS-TR-2158 (UMIACS-TR-88.92), Department of Computer Science, University of Maryland.

Biddle, R., Martin, A. & Noble, J. 2003. No Name: Just Notes on Software Reuse. In Proceedings of the 18th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, New York: ACM Press, 240 - 260.

Biggerstaff, T. J. 1989. Design Recovery for Maintenance and Reuse. Computer 22(7), 36 - 49.

Biggerstaff, T. J. & Perlis, A. J. (Eds.) 1989. Introduction. In T. J. Biggerstaff and A. J. Perlis (Eds.) Software Reusability Volume 1: Concepts and Models, New York: ACM Press, XV - XXV.

Biggerstaff, T. J. & Richter, C. 1989. Reusability Framework, Assessment, and Directions. In T. J. Biggerstaff and A. J. Perlis (Eds.) Software Reusability Volume 1: Concepts and Models, New York: ACM Press, 1 - 17.

Blake, S. P. 1978. Managing for Responsive Research and Development, San Francisco: W. H. Freeman and Company.

Boem, B. 1987. Industrial Software Metrics Top 10 List. IEEE Software 4(5), 84-85.

Bowen, G. M. 1992. An Organized, Devoted, Project-Wide Reuse Effort. ACM Ada letters 12(1), 43 - 52.

Brinkkemper, J. 1990. Formalisation of Information Systems Modelling. University of Nijmegen. Thesis Publisher, Amsterdam. Ph.D. thesis.

Brinkkemper, S. 1996. Method Engineering: Engineering of Information Systems Development Methods and Tools. Information & Software Technology 38(6), 275 - 280.

Brown, A. W. & Barn, B. 1998. Fundamentals of Component-Based Development. The Fifth International Conference on Software Reuse Tutorial, URL: http://people.cs.vt.edu/~edwards/icsr5/tut-brown-barn.html. Access date: January 20, 2002.

Brown, A. W. & Short, K. 1998. On Components and Objects: The Foundations of Component-Based Software Development, Sterling Software. URL: http://www.cbdedge.com/cbdweb/relatedtopics/sast97-foot.htm. Access date: May 24, 2001.

Brownsword, L. & Clements, P. 1996. Case Study in Successful Product Line Development, Software Engineering Institute. URL: http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.016.html. Access date: March 4, 2002.

CA 2002. Managing EBusiness Development: AllFusion™ Component Modeler, Computer Associates. URL: http://www3.ca.com/Files/BrochuresAndDescriptions/af_comp_modeler.pdf. Access date: June 10, 2003.

Card, D. & Comer, E. 1994. Why do so Many Reuse Programs Fail? IEEE Software 11(5), 114 - 115.

Castano, S. & Antonellis, V. D. 1997. Engineering a Library of Reusable Conceptual Components. Information and Software Technology 39(2), 65 - 76.

Checkland, P. & Scholes, J. 1990. Soft Systems Methodology in Action, Chichester: Wiley.

Coad, P., North, D. & Mayfield, M. 1995. Object Models: Strategies, Patterns, & Applications, NJ: Prentice Hall.

Cotter, S. & Potel, M. 1995. Inside Talligent Technology, Reading, MA.: Addison-Wesley.

Davis, G. B. 1982. Strategies for Information Requirements Determination. IBM Systems Journal 21(1), 4 - 30.

Davis, M. J. & Hawley, H. G. 1994. Reuse of Software Process and Product Through Knowledge-based Adaptation. In W. B. Frakes (Ed.) Proceedings of the Third Conference on Software Reusability, Seattle, Washington, USA, IEEE Computer Society Press, 44 - 52.

Dikel, D., Kane, D., Ornburn, S., Loftus, W. & Wilson, J. 1997. Applying Software Product-Line Architecture. IEEE Computer 30(8), 49 - 55.

Ezran, M., Morisio, M. & Tully, C. 2000. Practical Software Reuse: The Essential Guide, ESSI Surprise Project book. Paris: Freelife Publ.

Fischer, G. 1987. Cognitive View of Reuse and Design. IEEE Software 4(4), 60 - 72.

Fowler, M. 1997. Analysis Patterns: Reusable Object Models, Menlo Park, Calif. : Addison-Wesley.

Frakes, W. B. & Christopher, J. F. 1996. Quality Improvement Using a Software Reuse Failure Model. IEEE Transactions on Software Engineering 22(4), 274 - 279.

Frakes, W. B. & Fox, C. J. 1995. Sixteen Questions about Software Reuse. Communication of ACM 38(6), 75 - 87.

Frakes, W. B. & Isoda, A. 1994. Success Factors of Systematic Reuse. IEEE Software 11(5), 14 - 19.

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

Frakes, W. B. & Terry C. 1996. Software Reuse: Metrics and Models. ACM Computing Surveys 28(2), 415 - 435.

Freeman, P. 1983. Reusable Software Engineering: Concepts and Research Directions. In ITT Proceedings of the Workshop on Reusability in Programming, ITT Programming, Stratford, CT, 129 - 137.

Gaffney, J. E. J. & Durek T. A. 1989. Software Reuse - Key to Enhanced Productivity: Some Quantitative Models. Information and Software Technology 31(5), 258 - 267.

Galbraith, J. 1977. Organization Design, Reading, Mass.: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. M. 1995. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA.: Addison-Wesley Publishing Company.

Gibbs, W. W. 1994. Software's Chronic Crisis. Scientific American 271(3), 86 - 95.

Griss, M. L. 1993. Software Reuse: From Library to Factory. IBM Systems Journal 32(4), 548 - 566.

Griss, M. L. 1995. Software Reuse: Objects and Frameworks are not Enough. Object Magazine, February, 77 - 87.

Griss, M. L. 1996. Systematic Object-Oriented Reuse - A Tale of Two Cultures. ACM SIGSOFT Software Engineering Notes 21(1), 50 - 52.

Hall, P. A. V. 1992. Software Reuse, Reverse Engineering and Reengineering. In P. A. V. Hall (Ed.) Software Reuse and Reverse Engineering in Practice. London: Chapman & Hall, 3 - 31.

Hardy, C. J., Thompson, J. B. & Edwards, H. M. 1995. The Use, Limitations, and Customization of Structured Systems Development Methods in the United Kingdom. Information and Software Technology 37(9), 467 - 477.

Harmsen, A. F. & Brinkkemper, S. 1993. Computer Aided Method Engineering Based on Existing MetaCASE Technology. In S. Brinkkemper and A. F. Harmsen (Eds.), Proceedings of 4th European Workshop on the Next Generation of CASE Tools (NGCT `93), Sorbonne, Paris, France, Memorandum Informatica, Holland: University of Twente.

Harmsen, A. F., Brinkkemper, S. & Han, O. 1994. Situational Method Engineering for Information System Project Approaches. In A. A. Verrijn-Stuart and T. W. Olle (Eds.) Methods and Associated Tools for the Information

Systems Life Cycle (A-55). Amsterdam: North-Holland Publishers, 169 - 194.

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

Hirschheim, R. & Klein, H. 1989. Four Paradigms of Information Systems Development. Communications of the ACM 32(10), 1199 - 1216.

Hirschheim, R., Klein, H. & Lyytinen, K. 1995. Information Systems Development and Data Modeling, Conceptual and Philosophical Foundations, Cambridge: Cambridge University Press.

Hirschheim, R., Klein, H. & Newman, M. 1991. Information Systems Development as Social Action. Omega 19(6), 587 - 608.

Hoagland, J. 2003. NET Platform as Component Infrastructure, Components Online. URL: http://www.components-online.com/NETPlatform/. Access date: May 20, 2003.

Hofstede, A. H. M. & Proper, H. A. 1997. How to Formalize it? Formalization Principles for Information Systems Development Methods. Brisbane, Faculty of Information Technology, Queensland University of Technology.

Hofstede, A. H. M. & Verhoef, T. F. 1996. Meta-CASE: Is the Game Worth the Candle. Information System Journal 6(1), 41 - 68.

Hänninen, S. K., Jansson, M., Manninen, A., Raunio, A. & Äijänen, M. 2000. COMBO Project: Final report. Department of Computer Science and Information Systems, University of Jyväskylä.

IEEE 1999. IEEE Std. 1517, Standard for Information Technology - Software Life Cycle Processes - Reuse Processes, Piscataway, NJ: IEEE.

Iivari, J. 1990. Hierarchical Spiral Model for Information System and Software Development Part 1: Theoretical Background. Information and Software Technology 32(6), 386 - 399.

Iivari, J., Hirschheim, R. & Klein, H. 1998. A Paradigmatic Analysis Contrasting Information Systems Development Approaches and Methodologies. Information Systems Research 9(2), 164 - 193.

Jackson, M. 1975. Principals of Program Design, London: Academic Press.

Jackson, M. A. 1995. Software Requirement & Specifications - A Lexicon of Practice, Principles and Prejudices, New York: Addison Wesley Professional.

Jacobson, I., Griss, M. & Jonsson, P. 1997. Software Reuse: Architecture Process and Organization for Business Success, New York: Addison-Wesley.

Judd, C. M., Smith, E. R. & Kidder, L. H. 1991. Research Methods in Social Relations, 6th ed., Fort Worth, TX: Harcourt Brace Jovanovich.

Karlsson, E. 1995. Software Reuse: A Holistic Approach. Hoboken, NJ: John Wiley & Sons.

Karakostas, V. 1989. Requirements for CASE Tools in Early Software Reuse. ACM SIGSOFT Software Engineering Notes 14(2), 39 - 41.

Keen, P. G. W. & Scott-Morton, M. S. 1978. Decision Support Systems: An Organizational Perspective, Reading, MA.: Adison-Wesley.

Keller, R. K. & Schauer, R. 1998. Design Components: Towards Software Composition at the Design Level. Proceedings of the 1998 International Conference on Software Engineering, Los Alamitos, Calif.: IEEE Computer Society, 302 - 311.

Kelly, S. 1997. Towards a Comprehensive MetaCASE and CAME Environment: Conceptual, Architectural, Functional and Usability Advances in MetaEdit+. Department of Computer Science and Information Systems, University of Jyväskylä. Ph.D Thesis.

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 and Y. Vassiliou (Eds.) Advanced Information Systems Engineering, LNCS 1080, Berlin: Springer-Verlag, 1 - 21.

Kiely, D. 1998. Are Components the Future of Software. IEEE Computer 31(2), 10 - 11.

Kim, Y. & Stohr, E. A. 1998. Software Reuse: Survey and Research Directions. Journal of Management Information Systems 14(4), 113 - 149.

Kleijnen, J. P. C. 1980. Computers and Profits - Qualifying Financial Benefits of Information Systems, Englewood Cliffs, N. J: Prentice-Hall.

Korhonen, K., Lyytinen, K., Tolvanen, J.-P. & Kaipala, J. 2000. What is RAMSES All About? Department of Computer Science and Information Systems, University of Jyväskylä.

Koskinen, M. 2000. Process Metamodelling: Conceptual Foundations and Application. Department of Computer Science and Information Systems. Jyväskylä, University of Jyväskylä. Ph.D. thesis.

Kotonya, G. & Sommerville, I. 1998. Requirements Engineering: Processes and Techniques, Chichester: John Wiley & Sons.

Krueger, C. W. 1992. Software Reuse. ACM Computing Surveys 24(2), 131 - 183.

Kumar, K. & Welke, R. J. 1992. Methodology Engineering: A Proposal for Situation-Specific Methodology Construction. In W. W. Cotterman and J. A. Senn (Eds.) Challenges and Strategies for Research in Systems Development. Washington: John Wiley & Sons Ltd., 257 - 269.

Lee, N. Y. & Litecky, C. R. 1997. An Empirical Study of Software Reuse with Special Attention to Ada. IEEE Transactions on Software Engineering 23(9), 537 - 549.

Liao, H., Chen, M. & Wang, F. 1998. A Domain-Independent Software Reuse Framework Based on a Hierarchical Thesaurus. Software Practice and Experience 28(8), 799 - 818.

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

Lyytinen, K., Smolander, K. & Tahvanainen, V.-P. 1989. Modelling CASE Environments in Systems Development. In Proceedings of the First International Conference on Advanced Information System Engineering, Kista, Sweden.

Lyytinen, K. & Zhang, Z. 2000. A Framework for Component Reuse in a MetaCASE Based Software Development. In S. Brinkkemper, E. Lindencrona and A. Solvberg (Eds.) Information Systems Engineering: State of the Art and Research Themes, London: Springer, 107 - 122.

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

Maiden, N. A. & Sutcliffe, A. G. 1992. Exploiting Reusable Specification Through Analogy. Communications of the ACM 35(4), 55 - 64.

McClure, C. 2001. Software Reuse: A Standards-Based Guide, Los Alamitos, Calif.: IEEE Computer Society.

McIlroy, D. 1969. Mass Produced Software Component. Software Engineering Concepts and Techniques: 1968 NATO Conference on Software Engineering, New York: Petrocelli/Charter.

Merriam-Webster on-line Dictionary. 2004. Merriam-Webster Incorporated. URL: http://webster.com/. Access date: May 19, 2004.

MetaCASE 1999. MetaEdit+ Revolutionized the Way NOKIA Develops Mobile Phone Software, MetaCASE Consulting. URL: http://www.metacase.com /papers/MetaEdit_in_Nokia.pdf. Access Date: September 18, 2003.

MetaCASE 2000. Domain-Specific Modelling: 10 Times Faster than UML, MetaCASE Consulting. URL: http://www.metacase.com/papers /Domain-specific_modeling_10X_faster_than_UML.pdf. Access date: September 18, 2003.

MicroTOOL 2002. Get to know ObjectiF, microTOOL GmbH. URL: http://download.microtool.de/mT/pdf/objectiF/01/quicktour.pdf. Access date: August 10, 2003.

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

Moore, J. M. 1991. Domain Analysis: Framework for Reuse. In R. Prieto-Díaz and G. Arango (Eds.) Domain Analysis and Software Systems Modeling, Los Alamitos, Calif.: IEEE Computer Society Press, 179 - 203.

Morisio, M., Ezran, M. & Tully, C. 1999. Introducing Reuse in Companies: A Survey of European Experiences. In Proceedings of the fifth symposium on Software reusability, New York: ACM Press, 3 - 9.

Morisio, M., Ezran, M. & Tully, C. 2002. Success and Failure Factors in Software Reuse. IEEE Transactions on Software Engineering 28(4), 340 - 357.

MSDN 2004. Component Object Model, Microsoft Corporation. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ dnanchor/html/componentobjectmodelanchor.asp. Access date: May 16, 2004.

Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M.-A., Tilley, S. R. & Wong, K. 2000. Reverse Engineering: A Roadmap. In A. Finkelstein (Ed.) The Future of Software Engineering, New York: ACM Press, 47 - 60.

Neighbors, J. M. 1980. Software Construction Using Components. Irvine, Department of Information and Computer Sciences, University of California.

Neighbors, J. N. 1989. Draco: A Method for Engineering Reusable Software Systems. In T. J. Biggerstaff and A. J. Perlis (Eds.) Software Reusability I -- Concepts and Models. New York: Addison Wesley, 295 - 319.

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

Olle, T. W., Hagelstein, J., MacDonald, I. G., Rolland, C., Sol, H. G., Van Assche, F. J. M. & Verrjin-Stuart, A. A. 1991. Information System Methodologies - A Framework for Understanding. Boston, MA: Addison-Wesley.

Olle, T. W., Sol, H. G. & Verrjin-Stuart, A. A. (Eds.) 1982. Information Systems Design Methodologies: A Comparative Review, Amsterdam: North-Holland.

Olle, T. W., Sol, H. G. & Verrjin-Stuart, A. A. (Eds.) 1986. Proceedings of the IFIP WG8.1 Working Conference on Comparative Review of Information Systems Design Methodologies. Amsterdam: North-Holland.

OMG 2002. CORBA Components 3.0 - Component Model chapter, Object Management Group, Inc. URL: http://www.omg.org/cgi-bin/doc?formal/02-06-69. Access date: May 16, 2004.

Orlikowski, W. J. & Iacono, C. S. 2001. Research Commentary: Desperately Seeking the "IT" in IT Research - A Call to Theorizing the IT Artifact. Information Systems Research 12(2), 121 - 134.

Prieto-Díaz, R. & Freeman, P. 1987. Classifying Software for Reusability. IEEE Software 4(1), 6 - 16.

Purao, S. 1998. APSARA: A Tool to Automate System Design via Intelligent Pattern Retrieval and Synthesis. DataBase for Advances in Information Systems 29(4), 45 - 57.

Purao, S., Storey, V. C. & Han, T. 2003. Improving Analysis Pattern Reuse in Conceptual Design: Augmenting Automated Processes with Supervised Learning. Information Systems Research 14(3), 269 - 290.

Quatrani, T. 1997. Visual Modeling with Rational Rose and UML, Reading Mass.: Addison Wesley Professional.

Rine, D. C. & Nada, N. 1998. Software Reuse Manufacturing Reference Model: Development and Validation, George Mason University. URL: http://www.gmu.edu/depts/survey/. Access date: December 5, 2003.

Rine, D. C. & Sonneman, R. M. 1998. Investments in Reusable Software: A Study of Software Reuse Investment Success Factors. Journal of Systems and Software 41(1): 17 - 32.

Russo, N. L., Hightower, R. & Pearson, J. M. 1996. The Failure of Methodologies to Meet the Needs of Current Development Environments. In Proceedings of the British Computer Society's Annual Conference on Information System Methodologies, 387 - 393.

Russo, N. L. & Wynekoop, J. L. 1995. The Use and Adaptation of Systems development Methodologies. In M. Khosrowpour (Ed.) Managing Information & Communications in a Changing Global Environment, Proceedings of the Information Resources Management Association International Conference, Hershey, PA: Idea Group Publishing, 162.

Sage, A. P. 1995. Systems Management for Information Technology and Software Engineering, New York: John Wiley & Sons.

SEI 2000. A Framework for Software Product Line Practice - Version 3, The Software Engineering Institute (SEI). URL: http://www.sei.cmu.edu/plp/frame_report/productLPAs.htm. Access date: August 4, 2003.

Smolander, K., Tahvanainen, V.-P. & Lyytinen, K. 1990. How to Combine Tools and Methods in Practice: A Field Study. In A. S. B. Steinholz and L. Bergman (Eds.) Proceedings of the 2nd Nordic Conference on Advanced Information Systems Engineering, Berlin: Springer-Verlag, 195 - 214.

Sodhi, J. & Sodhi, P. 1998. Software Reuse: Domain Analysis and Design Processes, New York: McGraw-Hill.

Solvberg, A. & Kung, D. C. 1993. Information Systems Engineering, Berlin: Springer-Verlag.

Spivey, J. M. 1992. The Z Notation: A Reference Manual, N. J.: Prentice Hall, Engelwood Cliffs.

SRI 1995. DOD Software Reuse Initiative Technology Roadmap (V2.2), Software Reuse Initiative, Department of Defense. URL: http://dii-sw.ncr.disa.mil/ReuseIC/pol-hist/Roadmap/Cover.html. Access date: May 5, 1999.

STGCASE 2003. Axiom Case Tools, STG, Inc. URL: http://www.stgcase.com/casetools/ index.html. Access date: September 13, 2003.

Sugumaran, V. & Storey, V. C. 2003. A Semantic-Based Approach to Component Retrieval. ACM SIGMIS Database 34(3), 8 - 24.

Sun 2004. Enterprise JavaBeans Technology, Sun Microsystems, Inc. URL: http://java.sun.com/products/ejb/. Access date: May 16, 2004.

Szyperski, C. 1998. Component Software: Beyond Object-Oriented Programming, New York: ACM Press.

Tolvanen, J.-P. 1998. Incremental Method Engineering with Modeling Tools: Theoretical principles and Empirical Evidence. Department of Computer Science and Information Systems. Jyväskylä, University of Jyväskylä. Ph.D. Thesis.

Tracz, W. 1987. Software reuse: Motivators and Inhibitors. In proceedings of COMPCON 87, Thirty-Second IEEE Computer Society International Conference, 358 -363.

Tracz, W. (Ed.) 1988. Software Reuse: Emerging Technology, New York: IEEE Press.

Tracz, W. 1990. Implementation Working Group Summary. In Reuse in Practice Workshop Summary.

Tracz, W. 1991. Domain Analysis Working Group Report: First International Workshop on Software Reusability. ACM SIGSOFT Software Engineering Notes 17(3), 27 - 34.

UML 1995. UML Modeling Language, Standard Software Notation: Resource Center URL: http://www.rational.com/uml/, Rational Software Corporation. Access date: January 10, 1998.

Walls, J. G., Widmeyer, G. R. & EI Sawy, O. A. 1992. Building an Information System Design Theory for Vigilant EIS. Information Systems Research 3(1), 36 - 59.

Welke, R. J. 1983. IS/DSS DBMS Support for Information Systems Development. In C. W. Holsapple and A. B. Whinston (Eds.) Data Base Management: Theory and Applications, 195 - 250.

Yourdon, E. 1989. Modern Structured Analysis, Englewood Cliffs, N. J.: Prentice-Hall.

Yourdon, E. 1992. Decline & Fall of the American Programmer, Englewood Cliffs, N. J.: Prentice-Hall.

Zand, M., Arango, G., Davis, M., Johnson, R., Poulin, J. S. & Watson, A. 1997. Reuse Research and Development: Is It on the Right Track. In Proceedings of Symposium on Software Reusability (SSR97), New York: ACM Press, 212 - 216.

Zand, M., Basili, V., Baxter, I., Griss, M., Karlsson, E.-A. & Perry, D. 1999. Reuse R&D: Gap between Theory and Practice. In Proceedings of the 1999 symposium on software reusability, New York: ACM Press, 172 - 177.

Zand, M. K. & Samazadeh, M. H. 1995. Software Reuse: Current Status and Trends. Journal of Systems and Software 30(3), 167 - 170.

Zhang, Z. 1997. Methodology Engineering Based Component Reuse in a MetaCASE Environment. Department of Mathematics, University of Jyväskylä. Master Thesis.

Zhang, Z. 2000a. Defining Components in a MetaCASE Environment. In B. Wangler and L. Bergman (Eds.) Advanced Information Systems Engineering: 12th International Conference, CAiSE 2000, LNCS 1789, Heidelberg: Springer-Verlag, 340 –354.

Zhang, Z. 2000b. Enhancing Component Reuse Using Search Techniques. In L. Svensson, U. Snis, C. Sørensen, H. Fägerlind, T. Lindroth, M. Magnusson and C. Östlund (Eds.) Proceedings of the 23rd Information System Research Seminar in Scandinavia, Laboratorium for Interaction Technology, University of Trollhättan Uddevalla, 523 - 535.

Zhang, Z. 2004. Component-Based Reuse in Systems Analysis and Design: An Exploratory Study. Manuscript.

Zhang, Z. & Kaipala, J. 2004. Component Context Specification and Representation in a MetaCASE Environment. An early version was published in M. Khosrow-Pour (Ed.) Information Technology and Organizations: Trends, Issues, Challenges and Solutions, Proceedings of the 2003 Information Resources Management Association International Conference (IRMA2003), Hershey, PA: Idea Group Publishing, 712 – 715.

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

Zhang, Z. & Rossi, M. 2002. Component Modelling for Systems Analysis and Design. The 7th International Conference on Software Reuse Workshop on Component-based Software Development Processes, Austin, Texas, USA.

YHTEENVETO (FINNISH SUMMARY)

Informaatioteknologian soveltaminen uusille sovellusalueille on aiheuttanut sen että tietojärjestelmien kehittäminen on muuttumassa yhä markkinavetoisemmaksi. Koska turbulentti liiketoimintaympäristö tuottaa jatkuvasti uusia vaatimuksia ja piirteitä kehitettäville järjestelmille, tulee järjestelmistä hyvin suuria ja vaikeasti muokattavia. Lisäksi kehitettyjen sovellusten ylläpito on vaikeaa, koska järjestelmät koostuvat useista toisistaan riippuvista piirteistä, mikä vaikeuttaa uusien toiminnallisuuksien lisäämistä olemassaoleviin järjestelmiin. Jotta näitä monimutkaisia järjestelmiä voitaisiin kehittää nopeasti, ohjelmistosuunnittelun -tutkimusyhteisö pyrkii kehittämään olemassa olevien järjestelmien ja komponenttien uudelleenkäyttöä. Koska tietojärjestelmiä kehitetään yleensä samanlaisista osista, voidaan huomattava osa järjestelmistä rakentaa jo olemassaolevista komponenteista.

Komponenttien uudelleenkäyttö -paradigma pyrkii nopeuttamaan järjestelmien kehitystä ja alentamaan kehittämiskustannuksia, koska järjestelmä kootaan olemassa olevista komponenteista. Uudelleenkäytettävien komponenttien määrittely, suunnittelu, kehittäminen ja sijoittaminen muodostaa monimutkaisen prosessin joka vaatii tuekseen menetelmän komponenttien määrittelemiseksi, sekä myös kehittämisympäristön, jossa komponentteja kehitetään ja uudelleenkäytetään. Nykyinen uudelleenkäytön tutkimus on keskittynyt toteutusvaiheen so. ohjelmakoodin uudelleenkäyttöön, ja uudelleenkäytön tuki aiempiin järjestelmänkehitysvaiheisiin on jäänyt pääosin huomiotta.

Tämän tutkimuksen tavoitteena on edelleenkehittää komponenttien uudelleenkäytön teoriaa sekä strategioita jotka tukevat komponenttien uudelleenkäyttöä metamallintamista tukevassa MetaEdit+ metaCASE-ympäristössä (CASE, tietokoneavusteinen systeemityö). Koska metaCASE-ympäristössä on tehokkaat mekanismit evoluution hallintaan, mallintamiseen, organisointiin sekä komponenttien uudelleenkäyttöön, voidaan olettaa että ne toimivat myös tulevaisuudessa. Tässä tutkimuksessa on kuvattu metaCASE-ympäristön erityispiirteet, kehitetty käsitekehys joka huomioi uudelleenkäyttöprosessin, komponenttien rakeisuusasteen, ja abstraktiotason. Tämän jälkeen tutkimuksessa esitetään komponenttimalli joka muodostuu komponentin liitännästä, sisällöstä ja kontekstista, sekä esitetään mahdollisia strategioita komponenttien uudelleenkäyttöön. Tutkimuksen empiirisessä osassa tutkitaan laboratorio-olosuhteissa uudelleenkäytön vaikuttavuutta järjestelmän analyysi- ja suunnitteluvaiheissa käyttäen MetaEdit+ ympäristöä. Tutkimusote on konstruktiivinen sisältäen havainnointia, teorian kehittämistä, järjestelmäkehitystä ja kokeiden suorittamista.

Komponenttiperustainen uudelleenkäyttö metaCASE-ympäristöissä on suhteellisen uusi ja kehittyvä tutkimusalue. Tämän tutkimuksen pääkontribuutio on kaksijakoinen: on kehitetty käsitteistö, joka auttaa kuvaamaan ja käyttämään komponentteja, sekä toisaalta on kokeellisesti tutkittu analyysi- ja suunnitelutason komponenttien uudelleenkäytön vaikuttavuutta.