T U M

I N S T I T U T F Ü R I N F O R M A T I K

Re-Engineering for Reuse: Integrating Reuse Techniques into the Reengineering Process

Panagiotis K. Linos, Sascha Molterer, Barbara Paech, Chris Salzmann

TUM-I9824
November 98

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-11-I9824-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Re-Engineering for Reuse: Integrating Reuse Techniques into the Reengineering Process *

Panagiotis K. Linos¶,Sascha Molterer†, Barbara Paech†, Chris Salzmann†

†Institut für Informatik
Technische Universität München
80290 München, F.R.G.

¶Department of Computer Science
Tennessee Technological University
Cookeville, TN 38505, USA

(molterer|paech|salzmann)@in.tum.de

linos@csc.tntech.edu

Abstract

In this report, we present an overview of the existing software re-engineering process and its related concepts. We also classify existing software reuse techniques and we propose how to integrate such techniques into the software re-engineering process by following a component-based approach. In addition, we demonstrate how our methodology can be applied on a client-server legacy system.

Keywords: Re-engineering, Reverse Engineering, Reuse, Components, Software Engineering.

1 Introduction

Nowadays, in order to manage the masses of information a modern software system contains it is essential to reuse as much existing software as possible. Biggerstaff refers to reuse as the reapplication of a variety of kinds of knowledge about one system to another in order to reduce the effort of development and maintenance of that other system [BP89]. Reusing software brought up concepts like object-oriented modeling and component-based modeling which present today some promising success stories. For instance, Sneed presents a software recycling methodology and tools for extracting objects from existing legacy systems [Sne96]. In addition, Canfora and Cimitile discuss various scavenging techniques for detecting reusable software components [CC95].

On the other hand, software re-engineering is the process of reverse engineering followed by forward engineering. It has been a promising effort toward efficient evolution of existing software systems. Some important work has been accomplished in this area. For instance, Rugaber and Wills present a research infrastructure for re-engineering [RW96] whereas Rajlich presents a methodology for software evolution [Raj97].

Moreover, various efforts have focused on how to customize the re-engineering process towards reusing existing software components [Sam97]. The term software salvaging has also been used in the literature to refer to the process of re-engineering systems with the intend of finding reusable components [Arn92]. An example of such an effort is Galdiera's and Basili's component factory [BG91]. However, within that context, little effort has been given on how to efficiently incorporate modern reuse techniques in the re-engineering process.

*This paper originated in the project A3 of the Bayerischer Forschungsverbund Softwaretechnik (FORSOFT) and was supported by BMW and the Bayerische Foschungsstiftung

In this report we present a methodology of how to integrate specific reuse techniques into the re-engineering process with an emphasis on components. In order to demonstrate our approach, we discuss an example of re-engineering a client server legacy system with the intend of reusing software components.

The remainder of this report is organized as follows: In the next section we describe the existing state of software re-engineering and its major targets. The third section will describe existing reuse techniques and classify them with respect to the re-engineering process. Following we introduce the integration of those reuse techniques into the common re-engineering process and explain its benefits. We illustrate this approach by an example of re-engineering a client server legacy system. Finally, we present our conclusions in the last section.

2 The Status-Quo of Re-engineering

Re-Engineering is the general term for activities during corrective, adaptive, perfective or preventive software maintenance. Corrective software maintenance aims for diagnosis and correction of errors, for example, for the Y2K affected applications. Adaptive software maintenance intents to maintain proper interfaces within a changing environment. Perfective software maintenance satisfies users requests and preventive software maintenance improves future maintability and reliability. Specific tasks in a re-engineering process are for example, when a legacy complex software system is to be salvaged, when someone else's complex software must be understood and restructured or when the design of a complex software system needs to be recovered (i.e. reverse-engineered).

In this section we review the status-quo of re-engineering. Therefore, we first explain the basic terminology. This terminology is used to characterize the process of re-engineering. Then we shortly discuss Re-engineering tools and caveats.

2.1 Software Re-Engineering Terminology

In this section, we attempt to put existing software re-engineering terms in some perspective. First of all, the term *software* is used in this report to indicate source code, documentation and any related data [Pre97]. Specifically, the documentation part may include various specifications and designs of the existing system. The data part may include various related data such as input and output test data. The source code may include any programs written using one or more programming languages and integrated in a single system as well as any dependency files (e.g. make) that are needed to compile and run the system.

Today, there is no commonly accepted definition for the term *software re-engineering* and the related terminology is not standardized. However, there exist various valid definitions of software re-engineering that represent different point of views and perspectives. For instance, Chikofsky and Cross define it as "the examination and alteration of a subject system to reconstitute it in a new form and subsequent implementation of that form in their landmark paper [CC90]. More recently, Arnold defines software re-engineering as "an activity that improves one's understanding of software, or, prepares or improves the software itself for maintainability, reusability or evolvability" in his one-volume guide to the re-engineering literature [Arn92].

In this volume Arnold presents various re-engineering related technologies [Arn92]. We describe these technologies in the following with a focus on their transformational activities. Since transformation of information is a core notion within software re-engineering, we wish to extend Arnold's approach by considering transformation as an essential activity of re-engineering. Figure 1 shows the transformations applied to the existing code.

Next, we describe each related re-engineering technology and some related pointers to the literature are given for further study.

Software Restructuring Restructuring efforts have appeared as early as the mid sixties [BJ66] and they refer to any activities that focus on making the existing software easier to understand and eventually easier to change. According to Yourdon, restructuring refers to the reorganization of the control structure of the existing source code so that it conforms to the rules of structured programming [You89]. For instance, one can transform a

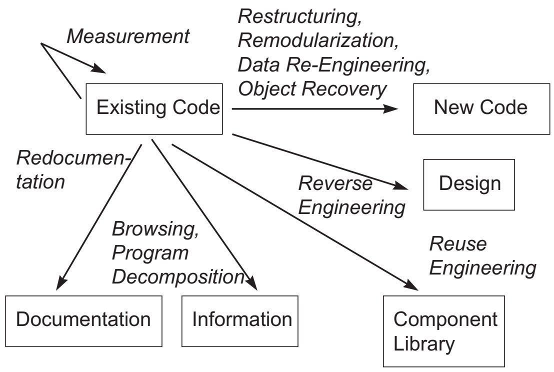

Figure 1: Re-engineering Techniques

spaghetthi-code-based system into new software that uses only one-entry-one-exit programming constructs (e.g. sequence-decision and iteration). Thus the transformation goes form source code to source code.

Arnold refers to restructuring as an activity that makes the source code more readable and understandable. Such activities may include indentation of statements, sorting of identifiers, use self explanatory variable names etc. [Arn89]. Restructuring has been fully automated and there are many tools to support its activities. Actually, the first automated restructures opened the way toward other re-engineering tools. More information on restructuring can be found at [GB88b].

Software Remodularization This term refers to any activities related to the re-organization of the module structure of an existing system. During remodularization one targets high cohesion and low coupling among the modules that comprise the existing system. Such efforts lead to an easier to maintain software system. The transformation process starts with the existing module structure and ends with an improved such structure. More literature covering this area can found at [Sch91] and [SJ88].

Data Re-Engineering Database schemas can be restructured, data dictionaries can be reorganized and existing data records can be updated. These activities are usually done when a transformation from one database system to another takes place. For instance, when one wishes to transform a relational to an object-oriented database system. Related references can be found at [RDW89].

Object Recovery The original software is transformed into objects with various relationships among these objects. Recent work attempts to convert existing software written in an non-object-oriented way into an object-oriented one. There is a lot of attention given to this area recently due to the popularity of the object-oriented technology. For instance, several efforts have been made to convert C to C++ programs [BL91], [Jac91], [DK91] and [Byr91].

Software Redocumentation This term refers to a serious effort of creating and updating documentation about software. Any knowledge found in the existing documentation, source code or specifications is transformed in an updated documentation. Therefore, the transformation process here starts from existing software including source code, existing documentation and related data and produces an up-to-date form of documentation that includes both external and internal documentation (i.e. comments in the code). The documentation usually is a combination of text, diagrams, tables and/or any related figures. Thus, it could be textual or graphical. Redocumentation efforts have started as early as the 70's and are described in various reports including [HKPS78], [Sne84] and [GN81].

Software Browsing The process of navigating within large amounts of information related to an existing software system is known as browsing. Today, there are many hypertext-based tools that allow for different textual views of software. Cross-referencing of such information are examples of software browsers. Moreover, there exist many graphical browsers today that transform existing textual information about software into manageable pictorial representations [Lin93b]. Thus, information about software is stored in a database and presented in various forms. There is a lot of information in this area since simple text editors were the first means of browsing through source code for better understanding. Some literature includes [RDLK90], [Lin93a] and [Rei88].

Program Decomposition It refers to a collection of activities involved in the process of breaking down an existing program into entities and relationships. The simplest example of program decomposers and tokenizers are UNIX tools such as lex, yacc and bison. These entities and relationships are stored into a database, which are used to facilitate further program analysis, measurement and/or statistical evaluation. This is a tedious and time consuming task and therefore many automatic tools exist that produce databases with program dependency information. Such tools include [Lin96]. Also, interesting work has been done in this area that helps graphically visualize statistical information about software usage [Eic98].

Program Comprehension The area of program comprehension, known also as software understanding, deals with the human side of software engineering. Specifically, it deals with ways of facilitating the process of understanding existing complex software. The main goal here is to comprehend the internal structure and overall design of an existing software system. This is a very expensive and difficult process for many reasons. A lot of work has been done in this area at different levels and granularities such as programming in the large versus programming the small. Specifically, efforts to recover the programmer's mental models have appeared as early as the 60's [Pen87]. Recent work includes cognitive model for large-scale software systems [vMV95]. Also, many tools exist for understanding existing code [Lin94], [MK88], [Til98] and [Big89]. There is an on-going international workshop devoted to program comprehension research and practices.

Reverse Engineering The existing information of a software system is transformed into a design view. For instance, we can recover structure charts and/or data-flow diagrams from existing source code. This effort is usually done in order to recover the original design of the system. The paper by Chikofsky and Cross is a good reference point [CC90]. There is an on-going working conference on reverse engineering.

Software Reuse Engineering Any modifications of an existing software system with an intend of making it more reusable is known as reuse engineering. Specifically, reusable software components of the existing system are detected, extracted and stored in a repository for reuse. Various efforts have been made in this area including the detection of reusable components and/or specific methods/techniques to find such components. The transformation here takes place from the existing software towards a database/library of reusable components. More information can be found in [Arn91], [GB88a], [RE90] and [BB90]. This technique is investigated in more detail in the following section.

The following two related re-engineering techniques do not transform the code, but are used to re-engineer the context of the software system (Business Process Re-Engineering) and to collect information about the quality of the code (Software Measurement).

Software Measurement This term refers to any activities related to measuring the quality of an existing software system. The term quality refers to the degree of meeting the original requirement specifications. There is a lot of work on establishing metrics for software [CBOR88], [McC76], [Zus93]. Also, there is a lot of work on program slicing [Wei88] and many others [HMKD82], [RU89]. No transformation of information is done here. There is an on-going workshop on software metrics.

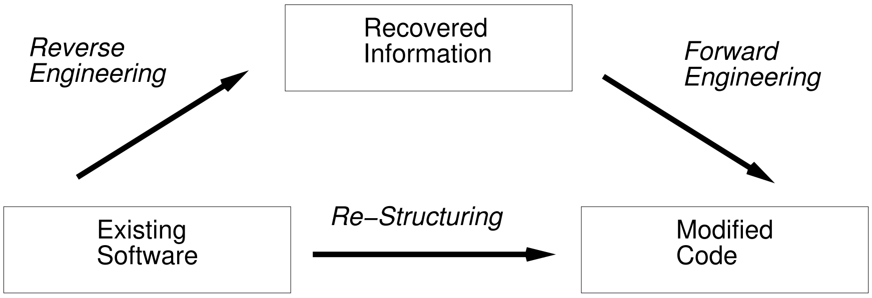

Figure 2: The basic Re-engineering Process

Business Process Re-Engineering This term refers to a collection of activities within an organization with the intention of improving the overall process of doing business. Within that context the existing software systems can be transformed into more efficient ones. This is a general approach towards doing business "right" within an organization with respect to the company's software systems. More literature in this area can be found at [Ham90], [DS90].

2.2 The Software Re-Engineering Process

Today, there is no standard definition of the software re-engineering process. A few efforts have been made in order to establish a life cycle for the re-engineering process. Several authors refer to software re-engineering as the process of reverse engineering followed by forward engineering [CI90] [Byr91] [Lin93a]. With in this definition, reverse engineering is viewed as the process of recovering design information from existing source code, whereas, forward engineering refers to the traditional software development life cycle.

Another author establishes three steps to describe the process of software re-engineering [Ulr90]. These steps include the inventory analysis followed by the positioning and transformation steps. In the first step, an information base is established that includes various components of the system. In the second step, the quality of the existing software is improved and finally in the last step, a new architecture is created from the existing system.

Figure 2 shows a generalized version of the two step-process. Software re-engineering begins with an effort toward understanding an existing software system. During this effort the design of the system is recovered from its source code. Related activities include browsing, static/dynamic program dependency analysis, program comprehension, detecting, extracting and storing design information etc. In the next step, the software re-engineering process includes a collection of activities that are performed in order to actually transform the existing software into a different, and easier to maintain form. Related activities include decomposition , restructuring, remodularization, redocumentation, reuse engineering, data reengineering, etc. This part involves a forward engineering strategy toward an improved version of the original system. It also focuses on improving the qualities of the existing software with respect to its evolvability and ease of change. Thus, it could be easier for software engineering to maintain.

2.3 Software Re-Engineering Tools

In this section, we outline what software re-engineering tools are and how they fit into the software re-engineering process. Software re-engineering tools (SRETs) are software tools that facilitate the process of re-engineering the structure/architecture and/or functionality/behavior of software systems with the intend of improving their understand-ability, maintainability and evolvability. It is estimated that within the next 10 years, we will be spending some trillions of dollars on maintaining and up-grading existing software if we continue to follow the same techniques known today. Software maintenance is a complex and expensive task due the rapidly paradigm shifting nature of the software technology. Thus, the need for software tools that facilitate the process of re-engineering software systems is compelling.

SRET class	Focus
Domain-based	specific application domain
Language-based	specific programming language and its problems
Paradigm-based	specific programming language and its problems
Dependency-Analysis based	maintain static/dynamic software dependencies
Hypertext-based	use of hypertext techniques to understand and re-engineer structural/behavioral aspects of programs
Program-animation based	use of animation to understand the behavior of a program

Table 1: A taxonomy of Software Re-Engineering Tools

Some criteria can be specified which characterizes software tools as SRETs. Tools that provide mechanisms to detect, extract, certify, qualify and store reusable software components from existing systems, tools that create and manage a repository of architectural and/or behavioral information about reusable software components and tools that provide efficient search engines for retrieving reusable components from the repository are SRETs. Furthermore tools that provide transformations for going from one kind of component representation to another, tools that provide partitioning techniques for decomposing large software components into smaller manageable pieces and tools that provide abstraction mechanisms for constructing higher granularity level components are considered as SRETs. An finally tools that maintain consistent and up-to-date documentation of software components and tools that provide presentation models for visualizing information about software components fall in the SRET category.

Table 1 shows a taxonomy of SRETs and the focus of each SRET class.

2.4 Re-Engineering Caveats

Software re-engineering is not a panacea, and therefore, it won't solve all the problems of the software maintenance crisis. It is a promising starting point towards establishing good standards for producing maintainable software. According to Arnold, there are several issues related to re-engineering that one should be aware when launching a re-engineering project [Arn92]. Re-engineering caveats include process-related risks such as the generation of very high costs due to manual re-engineering activities. Sometimes, the management may not be committed fully to an on-going re-engineering plan. In addition, various personnel related issues might arise such assistance from senior software engineering. Application related risks might include the lack of application experts. On the other hand, re-engineering technology related caveats might include issues such as when the recovered information is no longer useful or needed. It also possible to face problems when inadequate or unreliable tools are used in the project. Finally, if there is no global view and long term vision within the organization might create several problems with software re-engineering.

3 Techniques of Reuse

In this section we collect the basic reuse concepts. We focus on software parts and software description as units of reuse - in contrast to descriptions or parts of the environment or usage context of the software. First we introduce a taxonomy of reuse techniques. Then we apply this taxonomy to the most popular techniques, in particular to componentware which has received considerable attention lately.

The aim of this section is to classify the reuse techniques so they can be integrated into the Factory of the re-engineering for reuse process by the classified aspects (e.g. encapsulation) in Section 4.3.

3.1 A Taxonomy of Reuse Techniques

In this work we rely on several surveys of reuse techniques. Most similar is [Kru92] who also classifies the techniques according to different aspects. The main difference with this section is that we have incorporated the new developments since 1992, in particular the work accomplished on componentware. This also leads to slight changes in the aspects considered. [Kar95] provides a very comprehensive methodology for reuse, covering in particular issues of introducing and managing reuse in a company. Here, we don't deal with the organizational process of reuse, but concentrate on products and techniques for the technical reuse process.

In the following we distinguish the following aspects of reuse techniques :

- Level of Isolation
- Level of Specification
- Specialization
- Integration Mechanism
- Interface Description
- Aim

The *level of isolation* characterizes the reuse unit in terms of its relation to the complete software product. Here are single code elements (statements) the lowest level. The highest level offers a framework which offers a set of services without relying on other software parts. Patterns are also quite low, since they only describe projections of complete designs. A component can be self-contained, but often it will require services of other components to complete its functionality. The architecture describes also a complete software product, but on a very high level.

The *level of specification* characterizes the descriptions the developers are working with. These range from machine code to domain specific specifications. The same unit can be described on different level of specifications. In accordance with [Kru92] we call the lowest level the realization.

[Kru92] mixes the level of isolation and the level of specification into the abstraction aspect. There are some dependencies between them: for example code elements are not described on the specification level. However, for the more modern reuse techniques like components, different levels of specification are possible.

[Kru92] also distinguishes the hidden part, the variable part and the fixed part of an abstraction. The hidden part is only visible in the realization, while every specification can be divided into a fixed and a variable part. Because we allow several levels of specification of one unit, we do not stipulate a hidden part. There can be several abstractions of the same unit which specify different parts of the units, so no part needs to be hidden in all abstractions.

Usually, it is not possible to reuse the unit as such. Instead specialization, also called customization, of the unit is necessary. There are two kinds of such *specializations*:

- one is expected by the developers of the reusable unit. Therefore this customization can be done by changing the variable part of the unit. The variability can be achieved by parameterization, by inheritance, but also by configuration. In all cases, the realization has not to be changed.
- the other is not expected by the developers. It requires changes to the fixed part of the unit.

The first one is more interesting, of course. Reusable units are not stand-alone. They must be integrated with other reused units or with some newly build ones. Typically a *technique of integration* comes with the unit. This includes

- an integration environment which is the common glue between the different reused units. This environment typically consists of some mechanisms (technology) to integrate the units. They are based on a particular model of allowed interactions between the units. The mechanisms can be applied at development-time, build-time or run-time (see also the REBOOT project [Kar95]).
- an interface description of the reusable unit which makes explicit the external facets of the unit to be used by the other units.

Level of Isolation	**Level of Description for the (Re) User**	**Specialization Mechanisms**	**Integration Mechanisms**	**Interface Descriptions**	**Aim**
Code Elements	Machine Code	Parameter	Development time	Import/Export	Distribution Transparency
Pattern	Programming Languages	Configuration	Building time	Provided Capabilities	Explicit structure (property analysis)
Component	Specification	Refinement	Run-time	Include	High-Level-Specification
Architecture	Domain specific Specification	Constraints	Ad hoc	Provided Properties	

Table 2: The relevant aspects of reuse techniques

The *interface description* depends on the integration mechanism and vice versa.

It is important to also record the aims for which the techniques where developed. Most often, reuse is only one aim for the technique. Especially, the aims transparency of distribution and reuse are often quite heavily inter-twined. The former allows to combine distributed units in an integrated environment such that the distribution is not visible to the user. The latter aims to combine reusable units which need not to be distributed. We therefore distinguish

- aims in connection with the use of the software system (e.g. transparency), and
- aims in connection with the development of the software system (e.g. reuse).

For reuse to successful, the collection of reusable units and their retrieval must be organized. This is called *selection* in [Kru92]. [ZGWK97] gives an overview of relevant techniques. In this study we don't treat selection, since we focus on the application of units of reuse.

Table 2 lists again the reuse aspects which we will be looking at, together with the typical examples.

Domain specific specification applies to solution domain as well as to the problem domain. Refinement means adding details constructively, while constraints add details declaratively. Development time always means integration by the developers, while build time always means automatic support prior to run time. Provided Capabilities describe the data managed and the services provided to the environment. Provided Properties describe properties of the complete system not attributable to a single functionality.

3.2 Classifying Reuse Techniques

Table 3 classifies the major reuse techniques according to the aspects identified in the last section.

There are four major categories of reuse units:

1. **Domain specific architectures**, like frameworks and application generators, also try to capture experience. However, this is already embodied into code. Integration is not treated, since *DSA* are typically self-contained. The specialization mechanisms vary.
2. **Patterns** aim at reuse of experience. The description is therefore very comprehensive. Specialization is done by refinement, integration is done at development time and the interface is only described in terms of the provided properties.
3. **Programming and specification languages** aim at fine–grained specification, they typically use parameterization for specialization. Integration is done at build time or development time. Interface description is not used.
4. **Components** provide the most precise notion of interface. They do not use constraints for the specialization, since this is not constructive. Specification levels and integration and interface mechanisms vary.

Technique	**Level of Isolation**	**Level of Specification**	**Specialization**	**Integration Mechanism**	**Interface Description**	**Aim**
Architecture Components	Component	Component and Connectors, no fixed realization notion	Parameters	Integration of Components by Connectors	Capabilities	structure explicit for reuse, design decisions, analysis of properties
Modules	Component	Specification no fixed realization notion	Parameters	Build-time, programming language independent	Import / Export	Design and implementation structure, but language indep.
Classes	Component	Specification (in Programming language)	Refinement (Inheritance)	Build-time	Provided Capabilities	Reusable, data-centered
Code Components	Component	Programming language	Configuration	Build-time, programming language specific	Include	Facilitate installation
Execution Component	Component (Threads, Tasks)	Machine Code	None	Run-time system of the programming language.e.g. schedulers	None	Describe dynamic structures making use of the runtime environment, for analyzing run-time properties
Application Generator	Architecture	None - the architecture is hidden to the user	Constraints	None	None	High-level specification for prototyping and easy modifiability
Framework	Architecture	Programming language	Inheritance or Configuration	None	None	Reuse of architectures
Architectural Patterns	Pattern	Domain specific specification (conceptual architecture)	Configuration Refinement	Development-time	Provided capabilities	Capture architecture
Design Patterns	Pattern	Domain specific specification (design) and Programming language	Configuration Refinement	Development-time (combination)	Provided capabilities	Capture design experience
Idioms	Pattern	Programming language	None	Development-time	Provided capabilities	Capture implementation experience
High-level Programming Language Constructs	Code Elements	Programming language, realization by assembly language	Parameterized slots	Building-time (compiler)	None	Platform independence, understandability
Design and Code Scavenging	Code Elements (small parts or large parts with holes)	Programming language	None	None	None	Reuse which has not been foreseen by the developer
Very High-level Programming Language Constructs	Code Elements	Specification language, realization by assembly language	Parameters Templates	Building-time	None	General, executable specification
Transformation systems	Pattern (for transform. steps)	Specification	None	Development time (composition of transformations)		High-level spec. for prototyping and easy modifiability
Enterprise Componentware Components	Component	Specification language (Interface Description Language)	Parameter (not mandatory)	Build–time and/or run–time	Provided Capabilities	Distribution, transparency, reuse of components
Application Componentware Components	Component	Specification (in Programming language)	Parameter	Build–time	Provided capabilities and provided properties	Reuse of component
Compound Document Parts	Component	None	Scripting	Run-time	None	Inter-application transparency within one document

Table 3: Classification of some Major Reuse Techniques – see also [Kru92]

3.2.1 Software Architectures

Software Architecture is a very popular term, but its meaning is very vague. The aim is to capture the global structure of a software system design [Kru92]. As stated in [GS93], however, reuse of this global structure is only one aspect. According to [BDRe97], a formal foundation of the notion Software Architecture is needed.

Making this structure explicit also allows to make principled choices among design alter natives and is essential to the analysis and description of high-level system properties. As stated in [SNH95] also code generation from architectures is examined. Here we distinguish between two kinds of architecture reuse:

- reuse of the elements of the architectural structure. Here the architecture provides a framework for abstraction, integration and customization. As examples we treat the four different architectures identified by [SNH95].
- reuse of the architecture as the whole. Here the we look at domain specific architectures and frameworks.

Architectures as integration mechanisms for reuse units

Conceptual Components The conceptual architecture describes the software in terms of components and connectors. Components are specification units, but they need not be identified as such in the code. Therefore there is no clear notion of realization in this technique. This is even more true for the connectors. They capture protocols of interactions between the components.

Modules Module architectures describe the system in terms of modules and their export and import interfaces. Modules are functional components which already reflect implementation decisions. They are typically collected into layers to restrict import/export relations. Modules can be viewed as a realization notion for components and connectors, where both are realized through sets of modules. Again for modules, no clear notion of realization exists. They might be identified as such in the code, but need not be. Modules typically allow for specializtion by parameterization. For integration, modules make explicit their export interface which in turn can be used as imports by other modules. However, integrating modules from different sources, often leads to naming conflicts. Therefore, module interconnection languages have been developed [Kru92]. The compile time environment of the programming language fixes the mechanisms for integration.

[Kru92] discusses more general notions of source code components: ada packages and classes in object-oriented programming languages. Inheritance provides a special notion of specialization for classes. Instead of providing parameters in the original unit, the specialized unit takes attributes and operations from the original unit and extends them. This has some flavor of the design and code scavenging approach discussed below. The semantics of this notion has been an ongoing source of debate in the OO community. Classes also use a specific notion of interface. Classes provide services to be used by other classes. The called services operate on the data of the called class, while the imported parts of modules operate on the data of the importing module.

Code Components The code architecture reflects the choice of the programming language and the development environment. It aim s at facilitating system building, installation and configuration management. It can be viewed as a realization of the module architecture. The code units (files) realize modules. They themselves are organized into directories and libraries. The programming language code constitutes the abstraction specification (e.g. organized with include files), while its realization is the binary code. At this level not the reuse units themselves can be customized. Instead customization of the complete system consists of configuration. The mechanisms of integration are provided by the runtime system of the programming language.

Execution Components The execution architecture reflects the choice of the runtime environment, in particular wrt. performance, distribution and resource allocation. The abstractions used are thread, tasks, processes, address spaces and the like. The realization depends on the programming language. Typically there is no customization involved. The system model is fixed through the runtime environment. This also contains the mechanisms for integration of the units. Examples for such mechanisms are schedulers, load balancing.

Reuse of Architectures

Reused architectures are also called Domain Specific Software. They allow for reuse of the whole development process in a specific domain. The domain can be either characterized by the problem features (problem domain, e.g. avionics) or by the solution features (e.g. database). In [FS97] frameworks for system infrastructure , middleware integration and enterprise application are distinguished.

Application Generators Application generators allow to produce a self contained application from a very high level specification. Typical examples are 4GL for database applications, expert system generators and compiler generators. They aim at high-level specification for prototyping and easy modifiability. The unit of reuse is the global system architecture. However, this is hidden to the developer. The realization is a complete application. Specialization consists of constraints using domain specific concepts which are used as additional input for the generation process. Integration is often not necessary, since the resulting application is self contained. Sometimes , subsystems are generated which communicate through an abstract interface.

Frameworks Frameworks describe a domain specific, reusable architecture as a set of interdependent classes in an object- oriented language. According to [FS97], the primary benefits of application frameworks are modularity (through stable interfaces), reusability (through generic components) , extensibility (through hook methods) and inversion of control (through the framework's reactive dispatching mechanism). [Pre97] distinguishes White-Box-Frameworks from Black-Box-Frameworks. The distinction is due to two different ways of adapting a framework. In the first case the classes contain many abstract methods which have to be specialized. In the second case the framework already contains different subclasses which specialize the class with the abstract method. The user of the framework only selects the appropriate subclasses. Typically a White-Box-Framework matures to a Black-Box-Framework eventually. The customization of black-box-frameworks is much less error prone. Somewhat in between is the use of template methods. These methods encapsulate the variable parts of their body within so called hook methods. Hook methods are abstract methods which are specialized in subclasses. The level of abstraction is quite low, since the framework user needs to understand the programming language of the frame work. Integration is not relevant at first-sight , since the specialized framework is self contained. However , increasingly integration of different frameworks is necessary [FS97].

3.2.2 Patterns

Patterns aim at reusing experience. Patterns specify abstraction above the level of single classes or compone nts. They typically describe the constituent components, their responsibilities and relationships, and the ways in which they collaborate [BMR+96]. This is a particular kind of specification for structures. However, not a complete system structure is described, but only a view of the structure which is relevant for a particular problem (system property). Therefore specialization is achieved by refinement which means adding more detail. Sometimes also different variants are described within one pattern. Then specialization corresponds to configuration by choosing the adequate variant. Integration of different patterns has to be done at development time by combining the patterns to achieve combined properties. The interface of the pattern is described in terms of the properties it can achieve. There are three levels of patterns:

- architectural patterns describe conceptual architectures. In contrast to the architectural components the focus is on the structure, not on the individual components.
- design patterns describe views on the design relevant for a particular property
- idioms describe solution to programming language specific problems.

3.2.3 Programming and Specification Languages

[Kru92] emphasizes that also programming and specification languages incorporate reuse techniques.

Programming Languages

Here the unit of reuse is part of the code. The aim is in general platform independence.

High-Level Languages The abstractions offered by high-level languages are language constructs like case statements or while loops. The realization is given by assembly language code. The translation from specification to realization is automated by the compiler. Typically the high-level constructs allow for specialization by parameterized slots. The integration mechanisms are the syntax rules of the programming language.

Design and Code Scavenging While the reusable units of HLL and VHLL are designed as such by the language developers, design and code scavenging reuses parts which have not be designed for reused. The latter takes a small contiguous code fragment, and the former a big code fragment eliminating parts which are not needed or have to be changed. Thus there is no clear notion of abstraction and specialization, since there is no clear semantic relationship between the original unit and the specialized one. Also for integration no particular mechanism is provided.

Specification Languages

Specification languages are similar to programming languages as a reuse technique, since the unit of reuse is part of a language. In contrast to most programming languages, they aim at general non-operational abstractions. We do not include software schemas as in [Kru92], since they particularly aim at easy selection which we do not cover here.

Very High-Level Languages Very High-level languages such as SETL, PAISLey aim at executable specifications. The abstraction is chosen as high as possible without loss of generality, but allowing for easy description and modification. The reali zation is still assembly language. As in high-level languages specialization is made possible through paramet erized languages constructs and templates, and integration through syntax rules. In addition, very-high level language often aim at a simple declarative semantics.

Transformational Systems Transformational systems, similar to application generators, aim at high-level specification for prototyping and modifiability. In contrast to application generators, the intermediate results are reused within a step by step transformation process. This allows for reuse of the generated prototype, single transformation steps or the whole development history. There is no notion of specialization, since the transformation itself is reused. Of course, the transformation can be applied to different inputs. Integration of transformations means sequential composition. Often there are no semantic restrictions for this.

3.2.4 Componentware

Briefly, componentware are the technical means behind Component-Based Software Engineering (CBSE). With CBSE we mean the task of building, assembling and integrating software components. In this report, we distinguish between

- *Enterprise Componentware*,
- *Application Componentware* and

- *Compound Documents.*

For our classification of reuse techniques, we examine the items, the different componentware techniques deal with, namely the components, not the componentware itself. As for high-level languages the compiler is the background technique to translate between specification and realization, the componentware is the background technique to put components to work. Nevertheless, because of the impact on re-engineering for reuse, we'll picture how componentware works. Before we discuss the different componentware technologies, the next section will put some CBSE terms in some perspective.

The following definitions are neither comprehensive nor scientific founded. Actually, the development of componentware technologies started and happens in software companies like Sun, Microsoft or IBM and less in research.

A *software component* is a self-contained software building block that exposes its provided capabilities and properties by means of interfaces to its environment. Self-contains means that every component is an encapsulated, autonomous unit, adaptable only through its properties. Software components can be combined with other components and with newly written code to produce an application or another component. The way to use and customize a component distinguishes it from similar units like modules, libraries or classes. The difference is the strict separation between the provided capabilities, their realization and the components' build- and run-time environment. Module, libraries or classes are introduced to archive a similar separation. However, modules and classes are caught in the programming language they are written in. An interconnection between modules or classes written in different languages is not possible. A library could be used in different languages, even dynamically during run-time, but it depends on the operating system, for which it has been build. Furthermore, modules, classes and libraries don't provide a standardized way to customize their behavior. A software component can be customized to suit requirements of its environment through a standardized access to a set of properties without requiring access to the source code. These mechanisms make a component flexible to fit in different contexts to produce different application without having to change it and therefore a promising solution for building reusable software units.

A *component model* defines the basic architecture of a software component, specifying the structure of its interfaces and the mechanisms by which it interacts with its environment and with other components. The component model provides guidelines to create, implement and (re-)use components. A component model is necessary to make the software components self-contained. A connection between two component models is called a bridge.

Components can be as small as a simple GUI element like a button or as big as a complex application service like an account management function. The important aspect in adapting a component to build an application is not its size, it's the complexity of its interface, that is the manifold provided capabilities and properties. A button component has few capabilities and properties and will be easy integrated in many environments. An account management component has myriad of capabilities and properties and its integration will be probably as complex as writing it from scratch. With *component granularity*, we mean the different levels of the interface complexity.

In the next sections we will discuss the different componentware technologies and outline the applicability of their items, the components, as reuse technique.

Enterprise Componentware

Enterprise componentware is a middleware which provides support for a location transparent and language independent cooperation between distributed software components, which in this domain are called distributed objects. A predecessor of this technology is the Remote Procedure Call (RPC) middleware developed by Sun. The important features, an enterprise componentware has to provide are:

- mechanisms to locate and communicate to other components.
- standard interface description independent from specific programming languages and platforms.

For CBSE, the main goal of enterprise componentware is the integration of components on heterogeneous platforms. The components itself can be any unit of executable code as long as this unit provides and implements the

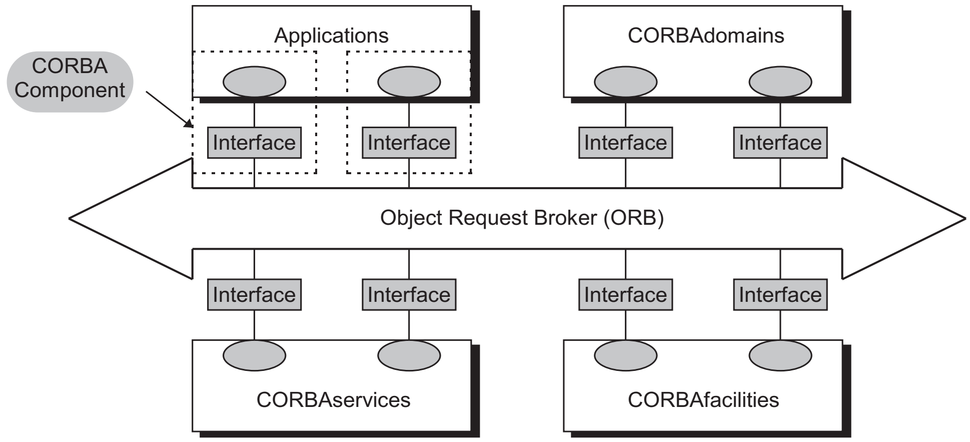

Figure 3: CORBA Object Management Architecture (OMA)

standard interface. The unit of reuse is the component specified by its interface which is written in a language and platform independent Interface Description Language (IDL). The code which realizes the component remains hidden for the (re-)user. The builder of a component - after writing the components' implementation - has to create the IDL description based on the functions that should be callable from the environment of the functions. Due to the fact that enterprise componentware is the middleware to realize distributed objects, the builder has no obligation to provide standardized properties for customization of a component. This means - in best case - that an enterprise componentware only provides its own way to specialize itself, that is, in most cases a less flexibility to (re-) use it in different environments. The mechanisms to integrate an enterprise component are provided either during building-time by compiling and binding the components' IDL in an application or during run-time by querying central instance to get the components' capabilities. The latter mechanism is more flexible but less simple to implement.

As example for an enterprise componentware, we introduce the Common Object Request Broker Architecture (CORBA). CORBA is a manufacturer independent standard for distributed objects by the Object Management Group (OMG) consortium [OMG92] and several companies offer implementations for this standard. The central component of a CORBA implementation is the Object Request Broker (ORB). The ORB provides a "component bus" for the cooperation between the (distributed) application of objects and provides the above features of an enterprise componentware.

Figure 3 shows the CORBA Object Management Architecture (OMA). The CORBAservices cover services to support work with distributed objects like persistence, naming or life cycle services. CORBAfacilities include services for applications like compound documents, system management and data interchange. CORBAdomains comprise domain-specific services, like financial services or health care. CORBAservices, CORBAfacilities and CORBAdomains are an extension to the enterprise componentware described above. Particularly the CORBAfaciltIes with bridges to other componentware products qualify the OMA as framework componentware.

Application Componentware

Application componentware facilitates the building and assembling of components and integrates them to applications. It provides mechanisms for encapsulation of implemented functionality behind standardized interfaces. The assembled components not only can be integrated to build applications, but also be used to provide their functionality through mechanisms of enterprise respectively document componentware. The unit of reuse is the component specified by its interface which is written in the programming language, the component is written in. In contrast to enterprise componentware, the main goal of application componentware the reuse of components in many applications written in one programming language, not the interoperability of components from different applications. Therefore, the specialization mechanism is standardized and the builder of a component has the obligation to provide standardized properties for customization of a component. The application componentware provides tools, the (re-) user can use to customize a component to fit the needs of the environment. The mechanisms to integrate components into an application are provided by the application componentware during build-time. The customization to fit into an specific environment is done during the development-time. Furthermore, the component behavior can be biased by the components' provided properties during run-time.

As an example for application componentware, this section introduces the JavaBeans framework from Sun. JavaBeans is an extension of the Java API [Fla97] with a component model framework. The components called beans are created like normal Java classes, but have to be compliant to the JavaBeans naming conventions. For example, the access functions to properties for customizing the beans must begin with *get* and *set*. Furthermore, the builder of a bean can add additional bean information with documentation, simple property editors or a visual bean customizer. The advantage of the naming convention approach is, that the (re-) user of a bean doesn't need to employ the JavaBean framework to communicate with a bean. The bean can be used in the same manner as ordinary Java classes. The disadvantage is that beans are not self-contained as described above. A JavaBean is only applicable within a Java environment. For a connection to other component models they need a bridge or wrapper and have to run in their own Java virtual machine run-time environment.

Compound Documents

The compound documents component model enables a document centered rather than an application centered approach on a desktop. That is, a document consists of several parts of different types. To manipulate the types, the user doesn't have to call the appropriate application itself, the compound document framework calls automatically the correct application component within the desktop. Even the editing takes place in the compound document itself and not in a separate window. One have to distinguish between to types of components and two different roles within compound documents model. First, there have to be components, which have to register to the desktop and claim their ability to handle - display, edit and store - one ore more types of document parts. These components are built and provided by software vendors. For the user of the compound documents, only the parts of the document a visible. So the unit of reuse is the hidden type handling component, which is only true for the document user. The builder of such a component reuses only the standardized mechanisms to write, distribute and register it. Because the component itself is hidden, the user works only with parts and no specification of the components. To change the behavior of a type handling component, some compound document models allow scripting, which means that beside the standard desktop integration, a user can adapt a type handling components in a document to his own requirements. The integration is done by the desktop at run-time of the compound document application. A disadvantage of the compound document model is its restriction to the configuration of a specific desktop. If a document has to be displayed on another desktop without the required type handling components, some parts are not accessible. These influences the exchange of compound documents between different users and different platforms.

In the following, we describe Object Linking and Embedding (OLE) from Microsoft as an example for a compound document standard. OLE 1.0 (1991) was a standard that enabled the creation and management of compound documents for the Windows operating system. It allows an embedding of objects in a document together with information about the format and the appropriate application. The aim of this version was to link and embed objects from one application into another and vice-versa. OLE 2.0 (1993) broaden this aim to an multi-purpose plug-in model for component oriented applications. That is, the compound document objects, the type handling components, can be plugged in to an application to extend the functionality without requiring changes. These components are called controls. Until now, OLE has evolved into a bunch of different services for different purposes. OLE consists of two major elements: the Component Object Model (COM), which is the underlying architecture, and a wide range of OLE services that enable software integration [Bro95]. The OLE services are:

- Services for application integration, which include Object Linking and Embedding, Visual Editing and Drag-and-Drop.
- Services for developers to customize standard applications, like OLE Automation and OLE Controls.
- Services for cooperation between different applications, like OLE Messaging, OLE DB and Distributed Component Object Model (DCOM) services.

	Enterprise Componentware Component	Application Componentware Component	Compound document type handling Component
encapsulated	*yes*	*yes*	*yes*
independent	*yes*	*no*	*no*
autonomous	*yes*	*no*	*no*
standardized properties	*no*	*yes*	*yes*

Table 4: A comparison of componentware components

OLE is part of the Windows operating system. The disadvantage of this is the deficiency in scaling in a heterogenious environment from one desktop to many heterogenious desktops in an enterprise. Due to the absence of OLE in other operating systems, the exchange of compound documents on the one hand and the reusability of type handling components on the other is limited to the Windows world. Due to the fact that the major part of OLE relying on Windows API functions, the task to provide OLE for other systems is not easy.

Componentware Technologies compared

In the beginning of section 3.2.4 a definition of an ideal software component was given. The components of the presented componentware technologies don't fulfill these definition completely. In table 4 the components are compared to the ideal software component definition.

An application componentware component is not independent, because bound to the used programming language, not autonomous and it depends on the application in which it's integrated. A type handling component is not independent, cause it's bound to a desktop system and it has no standardized properties, beside scripting.

4 Re-engineering for Reuse

In the last two sections we have reviewed re-engineering and reuse techniques. Re-engineering aims at providing a new structure for an existing system. Reuse aims at quality improvement and effort reduction by using existing system parts in a new context. We have examined different units of reuse and their (interface) specification, specialization and integration. Current reuse research concentrates mainly on domain specific architectures and components.

In this section, we present requirements for appropriate reuse techniques. With this requirements, we choose appropriate reuse techniques and integrate them into the re-engineering for reuse process.

4.1 Objectives

Re-engineering for reuse (REfR) attempts to answer the following two questions:

- How do we re-engineer existing software systems so that they may be reused in the future?
- How do we extract and prepare (reusable) units from existing software so that they can be (re)used to build a new system ?

The first question refers to the evolution of software systems. By evolving a system the major part of the system is reused, only minor parts are removed or added. Regarding the second question the use of software units found in previous systems to construct new ones is considered to be a new paradigm toward improving software quality and toward increasing the productivity of software engineers.

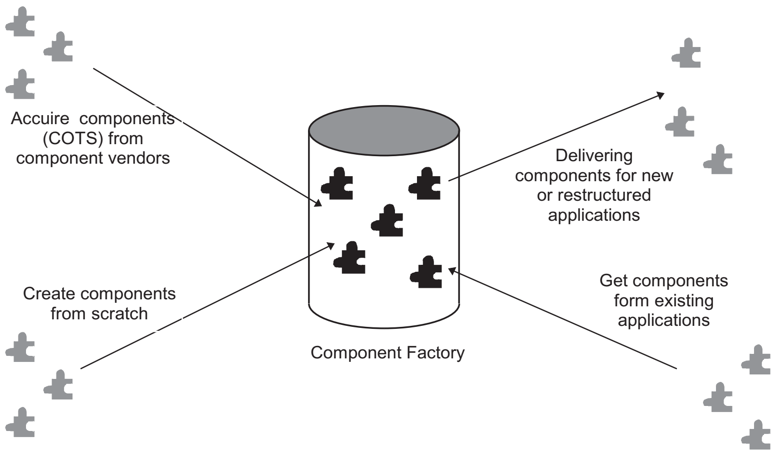

Figure 4: The Component Factory

4.2 The Process

In an effort to establish a model and a framework for the process of re-engineering for reuse, the idea of a "component factory" has been introduced in the late 80's [GB88b]. an organization responsible for developing and packaging reusable software components. The factory receives requests for reusable components from software engineers who are working on the traditional software development life cycle. When such requests are received the factory searches its repository of components to find and then customize, if necessary, the component needed. When the requested component cannot be found or it is too costly to customize, then the software factory will develop it from scratch, or builds it from more primitive existing components. After certification the component is released to the requestors. Because the efficiency of finding reusable components is very important within the context of the software factory, the repository must contain enough components to minimize the possibility of creating a component from scratch. Figure 4 shows a the component factory model.

There are two reengineering activities included in the factory: component identification and component qualification. The first phase can be fully automated whereas the second phase would need the intervention of a software engineer who has knowledge about the application domain, will assist with finding useful and interesting candidate components for reuse.

There is an overhead we have to pay for maintaining the component factory. Because of this overhead, in the short term, it is more expensive to develop reusable components rather than creating specialized programs. However, when we establish a large and well-organized repository of reusable components and provide an efficient search engine and an effective tailoring mechanism, then, in the long run, there will be clear economic benefit.

Figure 5 shows the framework of the re-engineering for reuse process. Specifically, it shows how the concepts of salvaging, reverse engineering and forward engineering fit within the process of re-engineering for reuse (compare

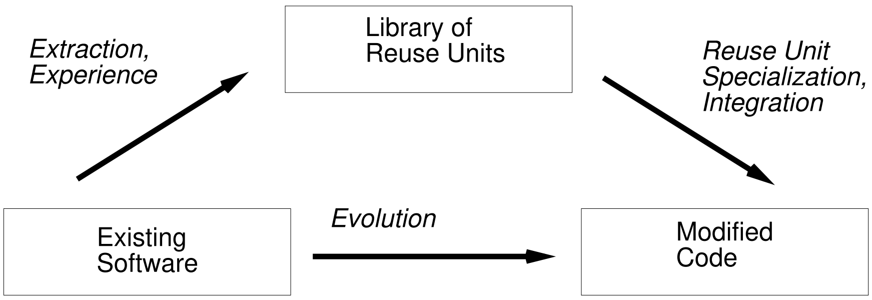

Figure 5: The Re-engineering for Reuse Process

	Corresponding Classification	Requirements	According techniques
Extraction	Level of Specification	Specification language	Modules Classes VHLL Architecture Components Architectural Patterns Design Patterns Transformation Systems Enterprise Componentware (Application Componentware)
Encapsulation	Level of Isolation	Modularization of large application systems	Architecture Components Modules Enterprise Componentware Application Componentware
Selection	not discussed		
Specialization	Specialization	at least Parameter	Architecture Components Modules Classes Application Generator Framework Architectural Patterns Design Patterns VHLL Enterprise Componentware Application Componentware
Integration	Integration Mechanism Interface Description	not Development-time Provided Capabilities	Classes Enterprise Componentware Application Componentware Architecture Components

Table 5: Accordance of Classified Reuse Techniques to the Requirements for Re-engineering for Reuse

with Figure 2). Reverse engineering is tailored to the extraction of reusable units. This extraction can be based on the source code, but it can also mean encapsulating experience with former software development projects. Development of reusable units always consists in packaging previous experience or existing code. Forward engineering is tailored to the selection, specialization and integration of components whereas evolution is viewed as the generalization of re-structuring.

4.3 Choosing Reuse Techniques for Re-engineering for Reuse

With regard to the above mentioned tasks during the re-engineering for reuse process, namely

- extraction,
- encapsulation,
- selection,
- specialization and
- integration,

we have to choose an appropriate reuse technique out of the ones classified in the last section.

Since we concentrate on the re-engineering aspect, again we do not deal with selection. The other tasks correspond closely with our classification: extraction is dependent on the specification level, encapsulation is dependent on the level of isolation. Specialization and Integration have been included as facets of our taxonomy.

For each task we choose specific requirements, due to the situation in our real re-engineering project: The technique should not be bound to a specific platform or programming language. Therefore, we choose specification

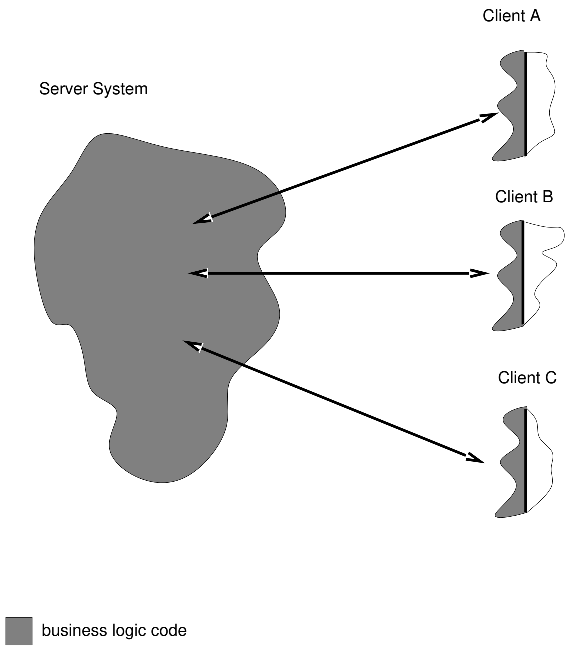

Figure 6: Initial Situation: A Client/Server Legacy System

language as the level of specification. The level of isolation should be as high as possible and particularly support the modularization of large application systems. We do not restrict the specialization mechanism, but paramaters should be possible at least. The integration should not be at development time to ease the burden of the designers. For the same reason, we require a clear notion of interface description.

The requirements on the tasks are presented in table 5 together with the techniques fulfilling the requirements according to our classification.

The two reuse techniques, which fullfill the requirements best are Enterprise Componentware and Application Componentware. They provide the best accordance to the requirements for re-engineering for reuse. In the next section, we'll use these techniques to illustate a re-engineering for reuse process for an example.

5 Integrating Reuse Techniques into the Re-engineering Process: An Example

In this section, we present an example of applying the re-engineering for reuse (REfR) methodology on a legacy object-oriented client/server software system. Our goal is to re-engineer this system for reuse, using componentware as a reuse technique. We first present some necessary definitions and some application-oriented assumptions.

We call an object-oriented client/server system a legacy system, if

- the classes and class hierarchies are not designed for reuse in different environments or there's no documentation, how to use the reusable classes
- the client is implemented as thick-client, that is, parts or all of the business logic code is on client-side and
- the client/server interactions don't use an common (and documented) interface.

If a business process changes, the business logic of a client/server system with thick-clients has to be changed at client and server side. Furthermore, if there is a need for another presentation type, the business logic has to be re-implemented. Figure 6 shows the initial situation.

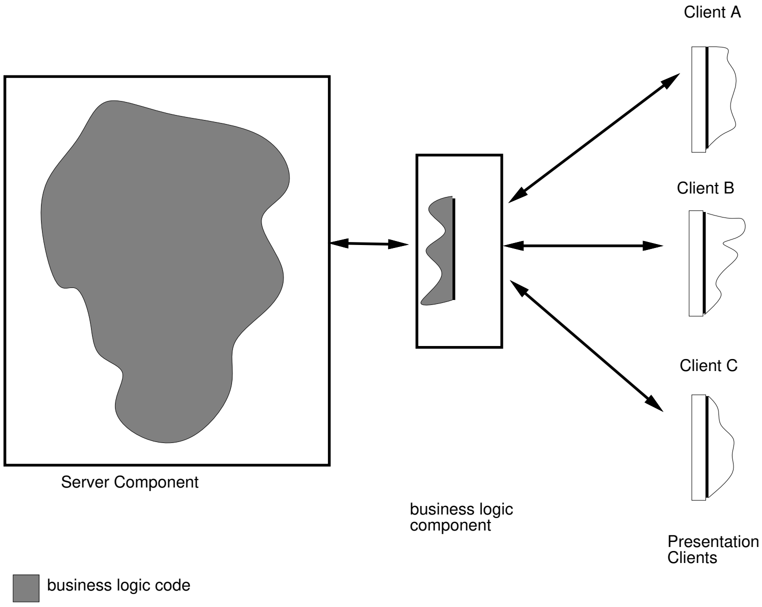

Figure 7: Encapsulation

We examine a client/server system with thin-clients, that is, all of the business logic is on server-side and only the presentation code is on client side.

This facilitates the adaptation of the system to a new business process or the generation of clients with other presentation types, e.g. a client based on Windows AWT and a client based on Motif for Unix Workstations. Due to the fact, that the business code is on server-side, another advantage of thin-clients is there reduced size and therefore for example a faster downloading time, if they are provided as Java applets over the Internet. If client and server do not interact using a common interface, it is difficult to change the servers' internal behavior and vice-versa. For example, if a client executes SQL statements to direct access server-side data, there's no way to change the used database without having to change the SQL statements within the clients.

Steps

In this section we describe the necessary steps needed to re-engineer the above client/server example. Figure 6 shows the initial situation: A client/server system with thick-clients (the business logic code is drawn grey) and no common interface (the access arrows are pointing within the server). Our goal is to re-engineer this system for reuse, using componentware as reuse technique.

In the *first step*, enterprise componentware is used to encapsulate client and server. The goal is to have a common interface for the clients to access the server. Furthermore, the business logic has to be removed from the clients and put to the server to have the advantages of the thin-client approach. The rectangles in Figure 7 are the enterprise componentware interfaces for clients and server. Having a common interface, the next step, componentization and integration, could be done independently for client and server. With a common interface and business logic it is possible to create and maintain different presentation clients. For example, an application implemented in C++, the encapsulated C++ server can be completed with a newly build Java applet as client.

In the *next step*, application componentware can be used to independently componentize the existing client and server into reusable components and remaining application specific code and integrate them with existing reusable components, which are replacing similar code, to a component-based application. The newly identified software components are checked into a enterprise-wide component factory, which serves as a means for storing, finding and maintaining such components. In Figure 8, the newly identified and created components are grey, the existing components out of the component factory are black. The former client business logic and the server are shown

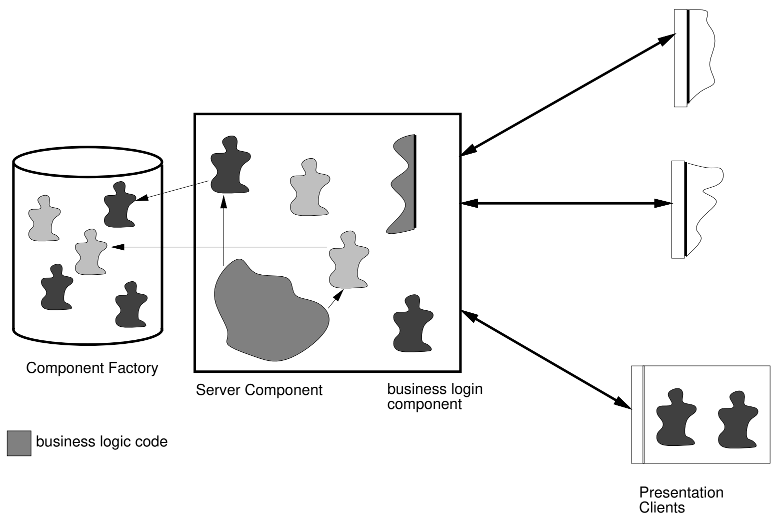

Figure 8: Componentization and Integration

as one object. This would be the ideal case. Normally, the removed business logic from the client has either to be rewritten and would be an object of its own or its functionality has to be written to the server component.

For re-engineering for reuse, step 2 is the most important and the most expendable. These expenses are only affordable, if a component factory enables the usage of the components enterprise-wide.

6 Conclusion

In this report, we studied the integration of reuse techniques into the re-engineering process and presented a methodology for re-engineering client/server legacy systems. To this end, we classified existing and more recent reuse techniques including application and enterprise componentware.

More specifically, we have argued in this report that from a re-engineering for reuse point of view, enterprise componentware is a way to encapsulate and connect applications at different stages of reusability. In addition, we made an attempt to show that application componentware can be used to extract reusable components out of existing systems and make them available by means of a component factory.

References

[Arn89] R. S. Arnold. Software restructuring. In *Proc. IEEE*. IEEE, April 1989.

[Arn91] R. S. Arnold. Risks of reengineering. In *Proc. Reverse Eng. Forum*, St. Louis, April 1991.

[Arn92] Robert Arnold, editor. *Software Reengineering*. IEEE Computer Society, 1992.

[BB90] J. W. Bailey and V. R. Basili. Software reclamation: Improving post-development reusability. In *Proc. Eighth Ann. Nat'l. Conf. On Ada Technology U. S. Army Communications Electronics Command, Fort Monmouth, N. J.*, pages 477–498, 1990.

[BDRe97] Manfred Broy, Ernst Denert, Klaus Renzel, and Monika Schmidt (edts.). Software architectures and design patterns in business applications. Technical Report TUM-I9746, Munich University of Technology, 1997.

[BG91] V.R. Basili and G. Galdiera. Identifying and qualifying reusable software components. *IEEE Computer*, pages 61–70, February 1991.

[Big89] Ted Biggerstuff. *Software Reusability*. ., 1989.

[BJ66] C. Bohm and G. Jacopini. Flow diagrams, turing machines, and languages with only two formation rules. *Comm. ACM*, 9(5):366–371, May 1966.

[BL91] P. T. Breuer and K. Lano. *Creating Specifications from Code: Reverse Engineering Technique*, volume 3 of *Software Maintenance: Research and Practice*, pages 145–162. ., 1991.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[BP89] Ted Biggerstaff and Alan Perlis, editors. *Software Reusability.* Addison-Wesley Pub Co, 1989.

[Bro95] Kraig Brockschmitt. *Inside OLE 2.* Microsoft Press, 1995.

[Byr91] E. J. Byrne. Software reverse engineering: A case study. In *Software-Practice and Experience.* ., 1991.

[CBOR88] V. Cote, P. Bourque, S. Oligny, and N. Rivard. Software metrics: An overview of recent results. *J.Systems and Software*, 8:121–131, 1988.

[CC90] E. Chikofsky and James Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7, 1990.

[CC95] Gerardo Canfora and Aniello Cimitile. Assesing modularization and code scavenging techniques. *Journal of Software Maintenace: Research and Practice*, 7:317–331, 1995.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 1990.

[DK91] M. F. Dunn and J. C. Knight. Software reuse in an industrial setting: A case study. In *Proc. 13th Int. Conf. On Software Eng.* IEEE, 1991.

[DS90] T. H. Davenport and J.E. Short. The new industrial engineering; information technology and business process redesign. In *Sloan Management Rev*, pages 11–27, Sommer 1990.

[Eic98] Steve Eick. Visualizing year 2000 program changes. In *6th International Workshop on Program Comprehension*, 1998.

[Fla97] David Flanagan. *JAVA in a Nutshell.* O'Reilly & Associates, Inc., 2nd edition, 1997.

[FS97] M. Fayad and D.C. Schmidt. Object-oriented application frameworks. *Communication of the ACM*, 40(10):32–38, 1997.

[GB88a] G. Galdiera and V. Basili. Identifying and qualifying reusable software components. *IEEE Software*, 1988.

[GB88b] G. Galdiera and V. R. Basili. Reusing existing software. Technical Report CS-TR-2116, Univ. of Maryland, College Park, 1988.

[GN81] R. L. Glass and R. A. Noiseux. *Software Maintenance Guidebook.* Prentice-Hall, N. J., 1981.

[GS93] D. Garlan and M. Shaw. An introduction to software architcture. In *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1993.

[Ham90] M. Hammer. Reengineering work: Don't automate, obliterate. In *Harvard Business Rev*, pages 104–112, Juli-August 1990.

[HKPS78] K. Heninger, J. Kallander, D. Parnas, and J. Shore. Software requirements for the A-7E aircraft. Technical report, NRL Memorandum Report 3876, 1978.

[HMKD82] W. Harrison, K. Magel, R. Kluczny, and A. DeKock. Applying software complexity metrics to software maintenance. *Computer*, 15(9):65–79, September 1982.

[Jac91] I. Jacobson. Re-engineering of old systems to an object-oriented architecture. In *Proc. OOPSLA*, pages 340–350. ACM, 1991.

[Kar95] Even-André Karlsson, editor. *Software Reuse – A Holistic Approach.* John Wiley & Son, 1995.

[Kru92] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[Lin93a] P. Linos. CARE: An environment for understanding and re-engineering C programs. In *Proceedings of IEEE Conference on Software Maintenance*, 1993.

[Lin93b] P. Linos. Facilitating the comprehension of C programs: An experimental study. In *Proceedings of the 2nd Workshop on Program Comprehension*, 1993.

[Lin94] P. Linos. Visualizing program dependencies. *Software-Practice and Experience Journal*, 1994.

[Lin96] P. Linos. A tool for maintaining hybrid C++ programs. *Journal of Software Maintenance*, 1996.

[McC76] T. McCabe. A complexity metric. *IEEE Trans. On Software Eng.*, SE-2(2), December 1976.

[MK88] H. A. Muller and K. Klashinsky. Rigi: A system for programming-in-the-large. In *10th International Conference on Software Engineering*, 1988.

[OMG92] OMG. Object management architecture guide – revision 2.0, 1992.

[Pen87] N. Pennington. Comprehension studies in programming. In *Second Workshop on Empirical Studies of Programmers*, 1987.

[Pre97] Wolfgang Pree. *Komponentenbasierte Softwareentwicklung mit Frameworks*. dPunkt Verlag, 1997.

[Raj97] Vaclav Rajlich. MSE: A methodology for software evolution. *Software Maintenance: Research and Practice*, 9, 1997.

[RDLK90] V. Rajlich, N. Damaskinos, P. Linos, and W. Khorshid. VIFOR: A tool for software maintenance. *Software-Practice and Experience Journal*, 1990.

[RDW89] J. A. Ricketts, J. C. DelMonaco, and M. W. Weeks. Data reengineering for application systems. In *Proc. Conf. On Software Maintenance*. IEEE, 1989.

[RE90] R. G. Reynolds and J. C. Esteva. Learning to recognize reusable software by induction. Technical report, Wayne State Univ., Detroit, Mich., 1990.

[Rei88] S. P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, 1988.

[RU89] D. H. Rombach and B. T. Ulery. Improving software maintenance through measurement. *Proc. IEEE*, 77(4):581–595, April 1989.

[RW96] Spencer Rugaber and Linda Wills. Creating a research infrastructure for re-engineering. In *3rd Working Conference on Reverse Engineering (WCRE)*, 1996.

[Sam97] J. Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.

[Sch91] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. 13th Int. Conf. On Software Eng.*, pages 83–92. IEEE, May 1991.

[SJ88] H. M. Sneed and G. Jandrasics. Inverse transformation of software from code to specification. In *Proc. Conf. On Software Maintenance*. IEEE, 1988.

[Sne84] H. M. Sneed. Software renewal: A case study. *IEEE Software*, 1(3):56–63, July 1984.

[Sne96] Harry Sneed. Object-oriented COBOL recycling. In *3rd Working Conference on Reverse Engineering (WCRE)*, pages 169–178, 1996.

[SNH95] D. Soni, R. L. Nord, and C. Hofmeister. Software architecture in industrial applications. In *Proceedings of the International Conference on Software Engineering*, 1995.

[Til98] Scott Tilley. A reverse-engineering environment framework. Technical Report CMU/SEI-98-TR-005, Carnigee Mellon University, 1998.

[Ulr90] William M. Ulrich. Re-engineering: Defining an integrated migration framework. *CASE Trends Magazine*, 1990.

[vMV95] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 1995.

[Wei88] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 1988.

[You89] Edward Yourdon. *Modern Structured Analysis*. Englewood Cliffs, 1989.

[ZGWK97] A. Zendler, S. Gastinger, W.Hesse, and P. Kosiuczenko. *Advanced Concepts, Life Cycle Models and Tools for Object-Oriented Software Development*. Tectum Verlag, 1997.

[Zus93] Horst Zuse. Criteria for program comprehension derived from software complexity metrics. In *2nd International Workshop on Program Comprehension*, 1993.