MINING COMPONENTS FROM LEGACY SYSTEMS THROUGH REVERSE ENGINEERING

Z. Tang and H. Yang

Department of Computer Science
De Montfort University
England
hjy@dmu.ac.uk

W. C. Chu and Y. C. Pen

Department of Computer Science and Information Engineering,
TungHai University
Taichung, Taiwan, R.O.C.
chu@csie.thu.edu.tw

C. H. Chang

Department of Information Engineering,
Feng Chia University
Taichung, Taiwan, R.O.C.
chchang@soft.iecs.fcu.edu.tw

Abstract

Legacy systems are increasingly acknowledged as major problems for most large corporations. Re-engineering is probably the best way to solve the problem. A typical component-based re-engineering process is: to use reverse engineering methodology to expose components from the existing system, to use repository to store and manage the components, then to restructure the new system, and to integrate the new system with reusable generic components and new-produced components by forward engineering. In our approach reusable components are mined from legacy systems, and made potentially reusable. New systems can be made by the integration of both mined and newly build components. The problem to be studied is an efficient and feasible way to extract components from the legacy systems. In this paper, component is explicitly defined and a sound method is proposed, detailed algorism is described with a case study.

1. Introduction

Let's suppose a scenario first: an application has served the business needs for a company for 10 or 15 years. During that time it has been corrected, adapted, and enhanced many time. People approached this work with the best intentions. Now the application is unstable. It still works, but every time a change is attempted, unexpected and serious side effects occur. Yet the application must continue to evolve. What to do?

Unmaintainable software is not a new problem. In fact, the broadening emphasis on software re-engineering has been spawned by a software maintenance "iceberg" that has been building for more than three decades [20]. Legacy systems are increasingly acknowledged as major problems for most large corporations.

A legacy system is any system that significantly resists modification and evolution to meet new and constantly changing business requirements. They've been developed but cannot be readily modified to adapt to the constantly changing business requirements, therefore, they provide the greatest opportunity to lower costs and improve the business.

A couple of problems exist. The first is to keep the systems running. The business depends on them. The second is to modify them to meet current business needs. Modification inevitably requires replacement, but replacement will not work for large, mission-critical information systems. The common-sense solution to the legacy problem is migration.

Obviously we can't prevent future legacies because we can't anticipate future business requirements or technology advances. But if you design your target systems to be completely decomposable (i.e., composed of separate components for each separable function), you'll be able to modify those components that do not support current needs when the time comes.

As we all know, it's easier to solve a complex problem when you break it into manageable pieces, software can be divided into separately individual named and address-able components, and can be integrated by modifying, deleting the generic components and adding some new-produced components to satisfy specific problem requirements. A reusable software component is a collection of operations designed to aid programmers in the development of applications programs. The use of software components saves both time and money. This savings, along with assured accuracy and reliability, is the main reason for using components to build component-based applications and a relatively sophisticated component-based approach to sharing, collaboration, and software reuse.

It became of strategic importance to be able to reuse existing knowledge to enable new applications to be assembled quickly and reliably. To achieve this developers require greater support and guidance for decomposing applications into meaningful pieces, and explicit representation of the rules for assembling new applications from a mixture of new and existing pieces.

2. Component-based Software Re-engineering

The IEEE [17] has developed a comprehensive definition that Software engineering is:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. And (2) The study of approaches as in (1).

Software engineering is a discipline that integrate process, methods, and tools for the development of computer software.

Software Re-engineering is the examination and alteration of an existing subject system to re-constitute it in

a new form. This process encompasses a combination

of sub-processes such as retargeting, reverse engineering, restructuring and forward engineering [7].

Re-engineering helps an organization move away from reactive maintenance to active management of its production system portfolio. The purpose of re-engineering is both to position existing systems to take advantage of new technologies and to enable new development efforts to take advantage of reusing existing systems. Re-engineering has the potential to improve software productivity and quality across the entire life cycle.

Reverse Engineering is the process of analyzing a system in order to obtain and identify major system components and their inter-relationships and behaviors. It involves the extraction of higher level specifications from the original system [20].

Reverse engineering has the potential to improve software maintenance productivity significantly. Reverse engineering is used to maintain software systems, to aid system migrations and conversions, and to discover reusable software components. Reverse engineering extends the useful life and value of existing systems by converting them to newer software technologies, and by migrating them to other operating platforms.

Software **reuse** covers the whole process of identification, representation, retrieval, adaptation and integration of reusable software components. Programmers have reused ideas, objects, arguments, abstractions, and processes since the earliest days of computing, today, complex, high-quality computer-based system must be built in very short time periods. This demands a more organized approach to reuse. But software engineering still makes less use of reusable components than any other kind of engineering [22]. By reusing reusable components, system reliability is increased, development time (i.e. design and coding time, verification or testing time) is reduced, and standards can be implemented as reusable components (i.e. standards for fault-tolerance or correctness, standards for user interfaces) [25].

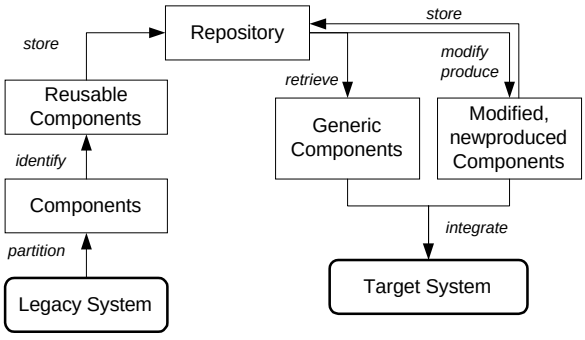

Figure 1: Process of Component-based Re-engineering

Component-based Development (CBD) is the industrialisation of the software development process based on assembly of prefabricated software components [21]. Two basic ideas underlie CBD. Firstly, that application development can be significantly improved if applications can be quickly assembled from pre-fabricated software components. Secondly, that an increasingly large collection of inter-operable software components will be made available to developers in both general and specialist catalogues. Manufacturing industries long ago learned the benefits of moving from custom development to assembly from pre-fabricated components. For modern manufacturing has evolved to exploit two crucial factors underlying today's market requirements: reduce cost and time-to-market by building from pre-built, ready-tested components, but add value and differentiation by rapid customisation to targeted customers.

3. Related Work

3.1 Various Definitions of Component

Various definitions are given out in many references.

Component is mentioned in the year of 1987 as "bits of software that can be replicated and, often with modifications, assembled repeatedly to form any number of applications" [8], in this definition, components are not regarded as off-the-shelf stuff, configuration should be considered rather than modification to make the future reuse of the components more flexible and adaptable.

And also, "A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction" [9], in the book namely Software Components with Ada. High cohesive and low coupling are the basic features of components, they should and must be included in the definition of components, and because of the variation of the levels of abstraction, but it is also important to mention the context in which a component can be used, which is missed in this definition.

Later on, O.Nierstrasz and D.Tsichritzis published a very abstract definition of components, which says "A software component is a static abstraction with plugs" [19], By "static", here means that a software component is a long-lived entity that can be stored in a software base, independently of the applications in which it has been used. By "abstraction", here means that a component puts a more or less opaque boundary around the software in encapsulates. "with plugs" here means that there are well-defined ways to interact and communicate with the component (parameters, ports, messages, etc.). This definition is a bit too abstract to be understood and over-mechanism.

Many more definitions came out from 1996, some companies also made their own definitions, like Meta Group's in 1998 [23], which says "Software components are de-fined as prefabricated, pretested, self contained, reusable software modules – bundles of data and procedures - that perform specific functions". Most of these definitions are made out for their own applications, which are not general enough to define a software component.

3.2 Related Projects

Many research works had been done in the area of reusable software components.

Software Compositions, for example, is a company

founded in 1989 to provide products (named Re-engineering Mentor) and services for Ada software development and maintenance organisations. The company's expertise in transformation and reuse technologies is applied to products and services for the re-engineering and reuse of Ada software [4].

FormaNet Technology Inc., another company, is dedicated to providing Java components. These components provide additional functionality to developers and speed development time. All of the components work with JDK 1.0.2. The utility products include the pre-built Java components (Byte code), the Java-Doc documentation for the components, example applets and the source code for the example applets [1].

The RECAST method [15] by the University of Durham was developed for ICL COBOL systems using IDMSX (Integrated Database Management Systems, Extended) and TPMS (Transaction Processing Management System). It provides a route for reverse engineering legacy COBOL systems into SSADM (Structured Systems Analysis and Design Method) logical specifications.

Another project namely IDENT [11, 12, 13, 14] is resulted from taking a number of techniques from the RE 2 project and constructing the techniques into a method using RECAST as a framework. It modifies, extends and integrates the two projects. The work is specifically geared towards large COBOL applications, consists of 10 steps.

Research works related to the component reuse and adaptation based on interface specifications and architectures, specification-based component retrieval, design representation for automating software component reuse were also explored in the early 90's by the Knowledge-Based Software Engineering Lab in the Department of Electrical and Computer Engineering and Computer Science, University of Cincinnati. They have produced a prototype component classification/retrieval system names REBOUND, and had extended the system with a formal model of architecture to support component adaptation and composition [16].

CATALYSIS by Trireme Company [24], is a development strategy for component-based design. Compliant with UML, Catalysis provides a set of design process patterns for: 1) Building software and business models from re-usable components; 2) Integrating legacy components with new development work; 3) Development or re-development from scratch; 4) Business process re-engineering; 5) Rigorous, robust design.

More recently, CBD products as EJB (Enterprise JavaBeans) [5], COM (Component Object Model) [6] and CORBA [3] have attracted many people's attention, with their relative language and platform-independent features, which make the creating and integrating of components more efficient.

There are also a number of software component repositories (such as the OAK Software Repository, it is a public service of Oakland University's Office of Computer and Information Services) offering many collections of computer software and information to Internet users free of charge [2].

extract					**newly-build**	**Store&**
code	**spec.**	**doc.**	**interface**	**design**		
RE MENTOR (Ada)					FORMANET	REBOUND
RE^2 (COBOL)	RE^2				ARACHNE	CAPS
RECAST (COBOL)	RECAST		RECAST		EJB	HELIOS
IDENT (COBOL)	IDENT		IDENT		COM	
RESTRUCT			RESTRUCT CIDER CATALYSIS	RESTRUCT DECODE	CORBA	
Other Research Works	Other Research Works					Other Research Works

Figure 2: Works done in the related area

4. Mining Components from Legacy Systems

4.1 Problem

Re-engineering covers both reverse engineering and forward engineering, a typical component-based software re-engineering process should contain several parts, namely: identification, classification, storing, retrieval, adaptation, composition.

More detailing, they are processed as below:

- Mining components from the legacy systems
- Wrap up the components with well-defined interface for future reuse
- Store the components in a software repository
- Build new reusable components if not available in the repository
- Make all the components off-the-shelf to meet specific user requirements
- Build the target systems by integrating the reusable components

When we consider the components of a software system, the following come to mind: program design documents, source code modules, object code modules, copy libraries, file descriptions, screen definitions, user manuals, etc. Functions, macros, procedures, templates and modules may all be valid examples of components [10], and component software may standardize interfaces and generic code for various kinds of software abstractions. Furthermore, components in a software may also be other entities than just software, namely specifications, documentation, test data, example applications, and so on. While most of the projects only concern one or two parts of them. And some others, i.e., the newly increased CBD, put more emphasis on composition, which is mainly of the forward engineering in the re-engineering process. Little attention have been paid in the reverse part, which is, mining the components from the existing systems.

What have been ignored is that, the existing systems are tested reliable, and domain specific, after being extracted

out, the components can be reused directly and efficiently.

4.2 Proposed Working Definition

The importance of a precise definition of what constitutes a **software component** and how to describe it have become a critical issue in the considerations about enhancements of the software development process in general and reuse of software pieces in particular, which helps to identify the components during system decomposition.

Components are larger than classes, can use multiple languages, include their own metadata, are assembled without programming, need to specify what they require to run.

Component systems are not invulnerable, the size of a component is inversely proportional to its match to any given requirement. Compared to objects, components are larger sized, physical entity instead of conceptual entity, support encapsulation, with defined interface. Components' strength is integration, so flexibility is key, and components are also highly scalable.

Thus, a definition of a component is clarified as follows: a **software component** is a coherent and configurable software package, independently of the applications in which it has been used, with well-defined interface in different context to interact and communicate with other components, to compose a larger system.

4.3 Proposed Method

Today, complex, high-quality computer-based systems are in need to be built in a very short time period. This strongly demands a more organised, more systematic approach to build software by reusing the software components.

Legacy systems are strongly in need of enhancement through re-engineering for the future reuse.

A great advantage of the extracted components is, they've already tested reliable in their history use. By borrowing an existing suitable software development method, which has been well developed, forward engineering therefore can be carried out easily in the process of building target systems. The extracted components are more domain specific than the newly built ones, and can be reused directly and efficiently.

The source code of a legacy system is first translated into CSL (Common Structural Language) through a "translator". The "universal translator" translates between a source/target language to/from RWSL (Re-engineering Wide Spectrum Language) (i.e., a COBOL-to-RWSL Translator [18]. This translator must be written for each source/target language and is simply a one-to-one mapping, to ensure semantics equivalence.

Five elements should be considered in, they are: code, specification, interface, design and documentation.

Source code is the most elementary part of a component, all the other elements are extracted out from code.

A component is more packaged than any old object. The assumption is that it will be used in many contexts unknown to its own designers. It should be robust in respect of abuse from other components, complaining rather than collapsing.

In addition to the executable code itself, there should be a specification documenting its behaviours unambiguously, using a suitable modelling and design notation. Since the average component will be used more than an individual product, it is, even more than usual, worth investing in good specification and design. The specification is essential because clients do not have access to the design, and should not have to waste time experimenting. A clear specification also tends to prolong the life of the designers' original vision, through many updates and enhancements [27, 28].

There are numerous undocumented programs. Maintaining a program that is un-documented (or poorly documented) is a costly task. The facts are: the original pro-grammars are gone; everyone that is around and knows something about the program doesn't want anything to do with it; the few comments that are in the code aren't necessarily correct (although they might be); and the small amount of documentation that exists (if any) is not necessarily correct or complete– it hasn't been updated for the las who-knows-how-many code updates.

Components are identified by their interface. Interface should be defined in different context to interact and communicate with other components.

The term black box conveys the idea of components whose internal workings are hidden, and so inaccessible, with the complementary notion that what is important about such a component are the ways in which it interacts with other components over some well-defined interface: its behavior.

What is important is how the components fit together, rather than how each performs its particular function. They should be functionally self-contained.

People know the systems through their design.

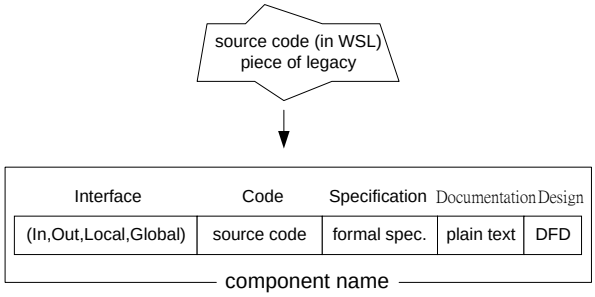

Figure 3: Decomposition

Finally, a well-structured repository is demanded to store all those elements for the future reuse, all associated software components could then be classified, stored, compared and retrieved, by software composition techniques.

4.4 Introduction to RA

Legacy systems usually have millions lines of code to maintain. However, not all of it can or should be restructured. Some programs have certain characteristics that will cause them to grow enormously in size if they are

restructured. Other programs need to be redesigned, not simply restructured. Re-engineering tools are developed to determine if and how to re-engineer existing programs.

The Re-engineering Assistant (RA) is the advanced version of MA (Maintainer's Assistant), which covers the aspects of reverse engineering, software maintenance, reuse and re-development. (as shown in Figure 4). The Re-engineering Assistant (RA) is an interactive software maintenance tool which helps the user to extract a specification from an existing source code program. It operates with WSL (or Wide Spectrum Language) which is a simple but very precise language, once the program is in WSL, it does not really matter in which language it was originally written. It is based on a program transformation system, in which a program is converted to a semantically equivalent form using proven transformations selected from a catalogue.

Some main functions of RA:

- Transforming: The Re-engineering Assistant can transform a WSL program. Transformation will produce a program which is functionally equivalent to – but ideally, much easier to understand or alter than - the original program. In addition to transforming a program, the user may make a conscious decision to edit it. The resulting program will not usually be equivalent to the original but should have been edited in such a way as to remove errors or to comply with changed requirements. The program is still guaranteed to be syntactically correct, nonetheless.
- Extracting Specification from Code: based on the construction of a wide spectrum language known as RWSL, a taxonomy of abstraction and a set of abstraction rules are developed, all of which enjoys a sound formal semantics, concentrates on engaging abstraction technology to extract formal specification from legacy source code.
- Metrics: Metrics may help to measure the progress made in optimising the program code and to measure the resulting quality of the program being transformed. A development of a classification of software metrics for reverse engineering is proposed and embedded in, which includes complexity measures, abstractness measures, object orientedness measures, economics measures and reusability measures.

Figure 4: Prototype of Re-engineering Assistant

5. Implementation

- CODE

Code can be obtained unchanged through RA.

- SPECIFICATION

Specifications are usually at different levels of abstraction, involving a process of crossing levels of abstraction, by adopting certain abstraction rules [29], specification can be represented as:

concrete → less abstract → more abstract

Abstraction rules are classified into two categories: elementary abstraction rules, rules to abstract source statements into logic formulae, which may be very redundant and specific; and further abstraction rules, which extract a more concise and abstract specification from the formulae through compositions and semantics weakening.

The formal definition of General Abstraction Rule is as follow:

$$S \geq LOG(S)$$

Where S denotes an uncomposite statement in source code, LOG gives the semantics definition of S in logical form.

- DOCUMENTATION

  The whole point is to make the program understandable by other people. Natural language is clearly a rich source of conceptual information. We propose the documentation in natural language, in the form of manual pages or comments, usually associated with the code.

- INTERFACE

  We can get the interface by the following steps:

- Cut out a procedure from the whole piece of code, loops or conditions should be cut as a whole.
- Identify the principle (dependent) data items and auxiliary (independent) data items, as both of them may change to each other in the process, a list is demanded to show the changes.
- Find out all the local variables and global variables.
- Present the interface in the form of below:

(In: var VarName, Out: var VarName, Local: var VarName, Global: var VarName)

- DESIGN

  We present the design by DFD (Data Flow Diagram), some main rules are listed is figure 5.

6. Conclusions

In a component-oriented approach, the activity of component engineering must be explicitly incorporated into the life cycle, and supported by the software process, the methods and the tools. Systematic rather than accidental software reuse requires an investment in component framework development and in software information management [26]. Component re-engineering can only be considered successful if the results are used to

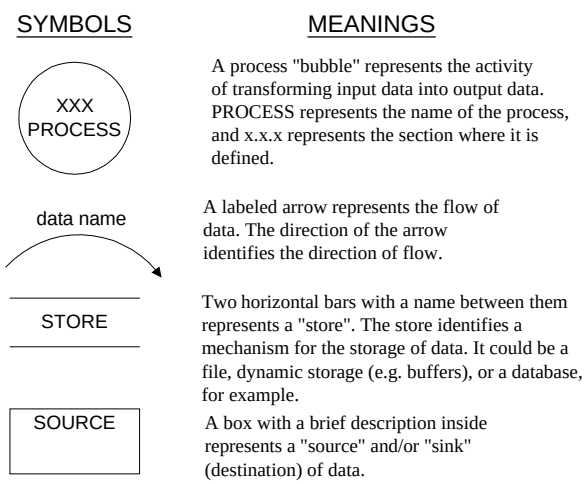

Figure 5: Data Flow Diagram

build more flexible applications.

Many research work have been explored in the area of software re-engineering, researches related to the components retrieval and adaptation were discussed in many ways, while few efficient and feasible ways could be found for understanding the existing system.

In conclusion, a sound systematic method of mining software components from legacy systems is proposed in this paper, through a clear definition and a feasible approach, which is believed has unbeatable advantage for the future reuse of the software components.

Thus, the value of it may be in its complete integrated, unified, domain specific, and documented support for components, and also support for the GUI, repository and providing a range of generic components as building blocks for the target systems.

References

[1] FormaNet Products - Java Components. http://www.formanet-tech.com/products.htm.

[2] OAK Software Repository. http://www.acs.oakland.edu/oak.html.

[3] Object Management Group Home Page. http://www.omg.org.

[4] Software Compositions - Ada Tools. http://www.swcomp.com/mentor.htm.

[5] Sun Microsystems. http://www.sun.com.

[6] Welcome to Microsoft's Homepage. http://www.microsoft.com.

[7] ARNOLD, R. S. Introduction: A Road Map Guide to Software Reengineering Technology. IEEE Computer Society Press (1994), 3–22.

[8] BIGGERSTAFF,T.,AND RICHTER, C. Reusability Framework, Assessment, and Direc-tions. IEEE Software 4, 2 (March 1987), 41–49.

[9] BOOCH,G. Software Components With Ada. Benjamin/Cummings, Menlo Park, CA, 1987.

[10] BRACHA,G. The Programming Language Jigsaw: Mixins, Modularity and Multiple In-heritance. PhD thesis, Department of Computer Science, University of Utah, March 1992.

[11] BURD, E., AND M.MUNRO. Enriching Program Comprehension for Software Reuse. In Proceedings of the International Workshop On Program Comprehension (IWPC'97) (1997), IEEE Press.

[12] BURD, E., AND MUNRO, M. Investigation the Maintenance Implications of the Replication of Code. In Proceedings of the International Conference On Software Maintenance (ICSM'97) (1997), IEEE Press.

[13] BURD, E., AND MUNRO, M. The Implications of Non-functional Requirements For the Reengineering of Legacy Code. In Proceedings of the 4th Working Conference On Reverse Engineering: WCRE'97 (October 1997), IEEE Press, Amsterdam, Netherlands.

[14] BURD, E., AND MUNRO, M. A Method for the Identification of Reusable Units Through the Reengineering of Legacy Code. The Journal of Software and Systems (1998).

[15] EDWARDS, H., MUNRO, M., AND WEST, R. The RECAST Method For Reverse Engineering. CCTA, NCC, Blackwell (1995).

[16] HTTP://WWW.ECECS.UC.EDU/JPENIX/KBSE/ INDEX2.HTML. The University of Cincinnati KBSE Homepage.

[17] IEEE Standard Collection: Software Engineering, 1993.

[18] KWIATKOWSKI, J., PUCHALSKI, I., AND YANG, H. Pre-processing COBOL Programs For Reverse Engineeering. In Proceedings of the International Conference On Software Maintenance (April 1998), Oxford, England.

[19] NIERSTRASZ, O., AND TSICHRITZIS,D.Object Oriented Software Composition. Prentice Hall, Englewood Cliffs, NJ, 1995.

[20] PRESSMAN, AND S., R. Software Engineering: A Practitioner's Approach, 4th ed. McGraw-Hill Book Company, 1997.

[21] SHORT, K. Component Based Development and Object Modeling. Stering Software CBD White Paper, Feburary 1997. Version 1.0.

[22] SIGFRIED,S. Understanding Object-Oriented Software Engineering. The Institute of Electrical and Electronics Engineers, Inc., New York, 1996.

[23] SZYPERSKI,C.Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998. ISBN-0-201-17888-5.

[24] TRIREME. Catalysis – Objects, Components and Frameworks With UML. http://www.trireme.com/ catalysis/, 1998.

[25] WAHLS, T. Software Engineering. In Course Notes for Comp 413 (1996).

[26] WEGNER, P. Capital-Intensive Software Technology. IEEE Software 1, 3 (July 1984).

[27] YANG, H. Formal Methods and Software Maintenance - Some Experience With the REFORM Project. In Position Paper, Workshop On Formal Methods (September 1994), Montery, USA.

[28] YANG, H., AND BENNETT, K. H. Acquaring Entity-Relationship Attribute Diagrams From Code and Data Through Program Transformation. In IEEE International Conference On Software Maintenance (ICSM'95) (October 1995), Nice, France.

[29] YANG, H., LIU, X., AND ZEDAN, H. Tackling the Abstraction Problem For Reverse Engineering in a System Re-engineering Approach. In IEEE International Conference On Sofware Maintenance (ICSM'98) (November 1998), Washinton D.C., USA.