Applying Software Reuse to Improve Software Quality and Productivity

Sen-Tarng Lai
Telecommunication Laboratories, Chunghwa Telecom Co., Ltd.
9, Lane 74, Hsin-Yi Road, Sec. 4, Taipei, 106 **Taiwan**

Abstract

The cause of software crisis is that software quality and productivity can not meet the requirement of software market. In order to resolve the software crisis, there are many approaches to be proposed. Software reuse is one of the high potential approaches to improve software quality and productivity. Some high-tech countries (like USA, Japan) have proposed and studied the topics of software reuse in 1970 and got several results. In software development procedure, the products of detailed design and implement phases are simple, definite and suitable for reuse. In this paper, the necessary documents of reusable software component will be proposed, and how to select and evaluate the software components with high reuse potential and high quality will be discussed and recommended.
Keywords: software reuse, software quality, software metrics

1. Introduction

Computer hardware techniques progress very quick in the recent yreas. It is not only reduce the price of hardware devices greatly, but also extend the application domain of computer to each levels of human live. After computer universalizing, the software crisis becomes a major issue in the computer field. The cause of software crisis is that software quality and productivity can not meet the requirement of software market. Improving the software productivity and quality is only one solution to resolve the software crisis. For this, in order to reduce the threat of software crisis, applying the concrete and effective approaches for improves the software quality and productivity is a trend. Boheam is a expert of the software engineering economics. He has studied and discussed several approaches to improve software quality and productivity as follows: software tools, software methodology, work environment, education, management, personal incentives, and software reuse [1]. Based on the approaches, many factors that may affect the software quality and productivity can be identified. For each factor, it is necessary to study and discuss deeply. However, there are many software systems that have high function redundancy in the same application domain. According to study report of Capers Jones in 1986, he analyzed the function redundancy of three application domains and proposed the results as follows:

(1) Domain of system programs has over 50 % function redundancy.
(2) Domain of telecommunication programs has over 70 % function redundancy.

(3) Domain of application programs has over 75 % function redundancy.

Therefore, software reuse is one of the most concrete and effective approaches to improve software quality and productivity. It is worth surveying and studying deeply.

To study software reuse, at first, is necessary to understand the software documents and products that have high reuse value and potential. In [12], McClure suggests several possibilities to be a software component such as program code, design specifications, plans, documentation, expertise and experience, and any information used to create software and software documentation. In [8], Langergan and Grasso discussed the designs and code reuse in software development. A code component in a reuse library is likely to be of little value; however, the detailed design documents should be very valuable for the adaptation to new applications. Thus, to be a suitable software component, Tracz [14] recommended that the detailed design documents should be associated with code modules. In this paper, we regard the software module as the reusable component that includes module design specification, source code, and related documents of unit testing.

In this paper, the software component is lay emphasis on the reusable program module in the same application domain, but not library routines of compiler or software components of COTS (Commercial Off-The Shelf). Therefore, the software module is regarded as the suitable software component in this paper. In Section 2, two widespread software reuse approaches that base on the documents of design and implementation phase will be discussed. In order to increase component adaptability and maintainability, the RSC (Reusable Software Component) should cover some necessary documents that include source code, detailed design documents, and related information of unit testing. The documents will be discussed in Section 3. A software component with high quality and high reuse potential is a necessary condition to be a RSC. In Section 4, an approach for evaluate the quality characteristics of software component will be described. Finally, a summary and our future work are given in Section 5.

2. Two widespread approaches for software reuse

Since 1980, many study results of software reuse are proposed continuous. However, building block style [6, 9] and generative style [10, 12] are two widespread approaches that are applied to the detailed design and implementation phase of software development.

2.1 Building block approach for software reuse

In software development procedure, the detailed design and implementation are two suitable phases to apply building block approach. It is because the documents of these two phases are definite, simple, easy to understand, and easy to modify. Based on the functional decomposition method, a software system can be modularized into many simple and independent module/program units after preliminary design. Each module/program unit has a specific function to be done. Using the specific function of module/program unit, user can try to retrieve the suitable candidate components from the RSC library (shown in Figure 1). To modify the candidate component is almost necessary for adapting to the new application software. However, reusing the suitable software component can always reduce the development time and cost. Reusing the suitable software component also can avoid redesigning and rewriting all of module/program unit in the application software development.

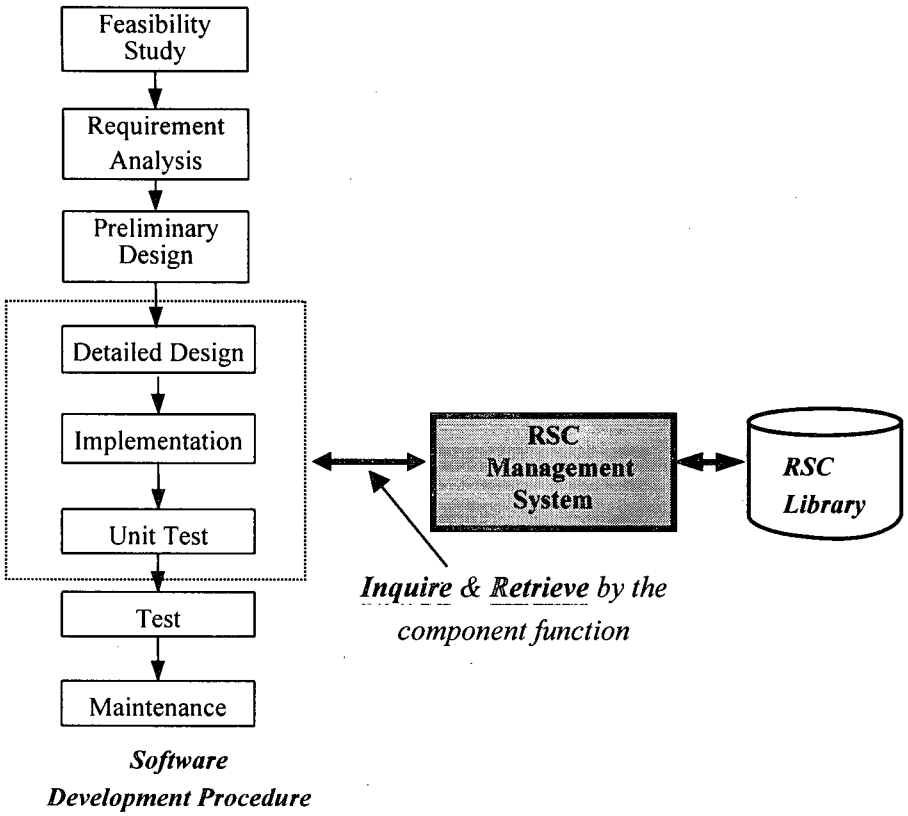

Figure 1. A procedure of building block approach for software reuse

2.2 Generative approach for software reuse

A feature of generative approach is to reduce the eight phases of software development procedure to five phases as follows: feasibility study, requirement analysis, partial preliminary design, test and maintenance. It provides great improvement in software development time and cost. Applying generative approach to develop new application software must have two major functions as follows:

(1) Prepare and manage all of general purpose RSC that base on the requirement of specific domain in advance.

(2) Provide a simple, definite, and easy to understand grammar and processor of requirement specification language for specific domain applications.

According to the grammar of requirement specification language, system analyst can specify the requirement specification by the language after he collect enough information of system requirement analysis. Then the language processor can analyze the language, retrieve the suitable software components, and generate the application software automatically (see Figure 2).

3. The necessary documents of a RSC

Software with effective modularity, i.e., independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design/code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize functional independence is a key to good design, and design is the key to software quality. For this, the documents that are produced by the detailed design and implementation phase have high reuse value.

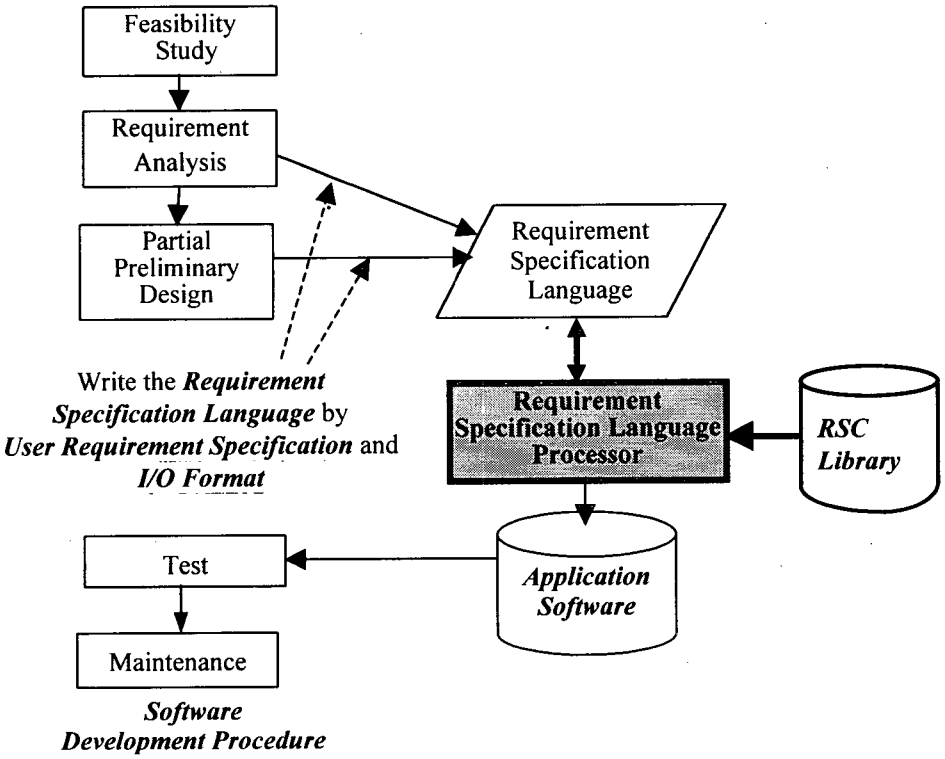

Figure 2. A procedure of generative approach for software reuse

Detailed design phase is a bridge of the preliminary design phase and the implementation phase. The major tasks are to decompose functional module into program unit, to define internal variable, to design the program logic, and to plan unit test specification. Based on the tasks of detailed design phase, several documents have to be completed in this phase. Module detailed design documents and unit program test specifications are two major documents of detailed design phase. The documents not only have high relationship with module/program unit but also have high reuse value. Implementation phase is a communication channel from detailed design phase to test phase. The major tasks of implementation phase include programming, unit program testing, and unit test report writing. According to the tasks, some important documents with high reuse value shall be produced. The documents include source code of module/program unit, test cases of unit testing, and test report of unit testing.

Summarizing the documents of detailed design and implementation phase, the necessary documents of a RSC can be identified as follows: module detailed design documents, source code

of module/program unit, and the related documents of unit testing. In order to be a RSC, each document has to be reviewed cautiously to verify the document quality. In addition, the documents and source code must also pass to the consistency confirmation. It is because verified, complete, and consistent documents are a basic condition to achieve the objective of software reuse.

4. A quality evaluation approach for the RSC

The RSC with high reuse potential and high quality is a necessary condition to improve software quality and productivity. In this Section, an approach to evaluate the quality characteristics of RSC will be proposed.

4.1 Quality Characteristics of the RSC

A software component with high reuse potential not sure has high quality. Reusing a software component with high reuse potential and high quality is a necessary condition to improve software quality and productivity. In [12], McClure lists several properties of a RSC as follows: **Additivity, Formal mathematical basis, Self-contained, Easily describable, Programming language-independent, Verifiable, Simple interface, Easily changed, Reusable.** Caldiera and Basili also defined some important qualities for component reuse [2]. They are **correctness, readability, testability, easy of modification,** and **performance.** In [13], Tracz recommended several quality characteristics of the RSC as:

- It is **well designed** - low complexity.
- It is **well documented** - according to an established standard.
- It is **well tested** - certified for reuse.
- Its function is **well understood** and likely to be used appropriately.

Although, some characteristics are impossible to measure or predict directly, there still exist many software characteristics that can be used for measuring the RSC. To survey and summarize these characteristics, we identify a set of corresponding software metrics for the RSC (see Table 1).

Table 1. A contrastive table for the quality characteristics of RSC and corresponding metrics

Quality Characteristics / *S/W Metrics*	*Proposed by McClure*	*Proposed by Caldiera & Basili*	*Proposed by Tracz*
Low Complexity	- additivity - easily changed - easily describable	- easy to modification - readability	- well designed - well documented - well understood
High testability	- verifiable	- testability	- well tested
High modularity	- simple interface - seft-contained - reusable		- well designed - used appropriately

- **Complexity:** The complexity of the program is a measure of the effort required to understand the program and is usually based on the control or data flow of the program. Thus, program complexity measures can be used to determine the complexity of a program before and

after modification, and used to identify candidate components for further refinement and reuse.

- **Modularity:** Software with effective modularity, i.e., independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified. Independent modules are easier to maintain and test because secondary effects caused by design/code modification are limited, error propagation is reduced, and reusable modules are possible.
- **Testability:** Testability is the ease of testing the program, to ensure that it is error-free and meets its specification. The ease and effectiveness of testing will have an impact upon the adaptability of a software component.

4.2 Some software metrics for the quality characteristics measurement of RSC

In order to measure the quality characteristics of the RSC, we shall consider some software metrics. According to the meta-metrics defined by the Conte [3], we are interested in metrics that are simple, robust, useful for the design and implementation phases, and that can be analyzed properly. The metrics can be separated into three sets as follows:

- **Code Complexity Measures**. In the mid-1970s, there was significant research activity for developing measures of code complexity. The code metrics was easy to obtain since they could be calculated from the product code by automated tools. Early examples of these techniques include McCabes's Cyclomatic Complexity measure [11] and Halstead's Software Science [5]. Program nesting level can ensure that components are readable and not functionally overloaded. To measure component complexity, we also include it.

(1) *Logic structure metric (McCabe's Cyclomatic complexity):* In [11], McCabe observed that the difficulty of understanding a program is largely determined by the complexity of the control flow graph for that program. A program with a larger decision count is believed to be generally more complex than another program with a smaller decision count.

(2) *Data structure metric:* In [5], Halstead developed a number of metrics that are computed from easily obtained properties of source code. Halstead's effort metric can be interpreted as the effort required to read and understand a program. It is also well correlated with the observed effort required to debug and modify small programs.

(3) *Nesting level of program construct:* As a general guideline, nesting of program constructs to depths greater than three or four levels should be avoided.

- **Testability measure:** Three commonly used measures of test coverage in unit program testing are statement coverage, branch coverage, and logical path coverage. To measure the logical path coverage is more difficult and needs more effort than other test coverage. Typically, test completion criteria do not consider the measure of logical path coverage. So, we use statement coverage and branch coverage to measure the testability of software component. In [4], Fenton considers that normally a developer would insist on 100% statement coverage and a high branch coverage of around 85%.
- **Modularity measures:** Coupling and cohesion are the major metrics for measuring the module modularity. Coupling is a measure of the degree of interdependence between modules. Cohesion is an attribute of individual modules, describing their functional strength. The internal cohesion of a module is measured in term of the strength of binding elements within the module.

Three high-level software metrics can be measured by seven primitive metrics as follows:

(1) Complexity characteristic is measured by logic structure metric, data structure metric and nesting level of program construct.

(2) Testability characteristic is measured by statement coverage and branch coverage.

(3) Modularity characteristic is measured by cohesion metric and coupling metric.

Each primitive metric can be collected and computed by the automatic or semi-automatic software tools. According to the data collection, programmer experience, and expert knowledge, the reasonable threshold of RSC quality characteristics are reommended. Applying the experience and knowledge of domain expert to select the software component with high reuse potential. Referring the software metrics and recommended thresholds to evaluate the quality characteristics of software component. The RSC extraction procedure is proposed and shown in Figure 3.

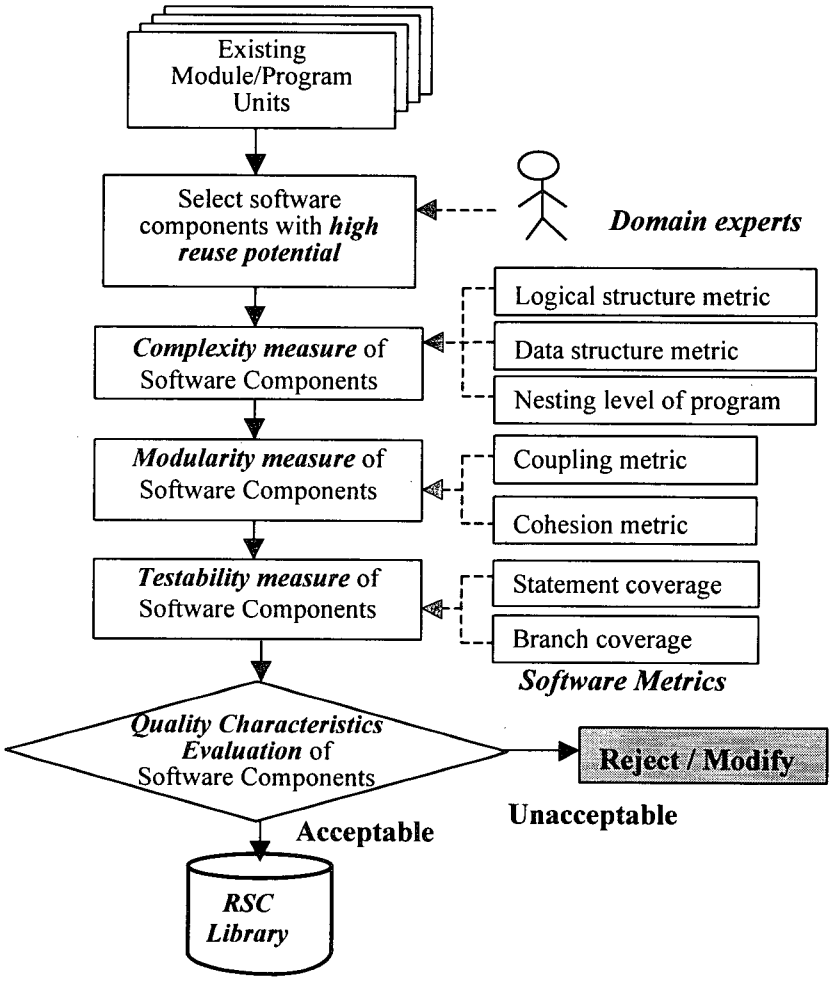

Figure 3. Extracting procedure for the RSC

5. Conclusions

In software development procedure, each phase has the specific documents to be completed. The completed documents of detailed design and implementation phase have the features of clear,

simple, easy to understand, and easy to modify. The features make the documents of detailed design and implementation phase have high reuse value and potential. Two widespread software reuse approaches also based on the documents of design and implementation phase. Therefore, in this paper, the documents of detailed design and implementation phase are regarded as the necessary documents of the RSC.

The major objective of software reuse is to improve software quality and productivity. Extracting the RSC that have high quality and high reuse potential is one of the major tasks to accomplish the objective. The domain experts extract the high reuse potential components by their knowledge and experience. However, they always omit the quality characteristics of software components. In this paper, several software metrics are specified for measuring the quality characteristics of the RSC. The individual software metric cannot evaluate the overall quality characteristics of the RSC. Therefore, the software metrics should be combined and conflict situations in metric combination must be reduced [7]. Our future works is to plan and study a hybrid metric combination model that based on the linear and nonlinear combination for resolving the metric combination and applies the rule-based system for reducing the conflict situations in metric combination.

References

[1] Boehm, B. W., M. H. Penedo, "A Software Development Environment for Improving Productivity," *IEEE Computer*, Vol. 17, No. 6, pp. 30-44, 1984.

[2] Caldiera, G. and V. R. Basili, "Identifying and Qualifying Reusable Software Components," *IEEE Computer*, Vol 24, No 2, pp. 61-70, 1991.

[3] Conte S.D., Dunsmore, H.E., and Shen, V.Y., *Software Engineering Metrics and Models*, Benjamin/Cummings, Menlo Park, 1986.

[4] Fenton N.E., *Software Metrics - A Rigorous Approach*, Chapman & Hall, London, 1991.

[5] Halstead, M. H., *Elements of Software Science,* North-Holland, New York, 1977.

[6] Kaiser, G. E., and D. Garlan, "Melding Software Systems from Reusable Building Blocks," *IEEE Software*, Vol. 4, No. 4, pp. 17-24, 1987.

[7] Lai, S. T. and C. C. Yang, "A Reusable Software Component Extraction Approach for Increasing Software Quality," *Proc. of Taipei International Quality Conference*, pp.177-182, 1994.

[8] Lanergan, R. G. and C. A. Grasso, "Software Engineering with Reusable Designs and Code," *IEEE Trans. Software Eng.*, Vol. 10, No. 5, pp.498-501, 1984.

[9] Lenz, M., H. A. Schmid, and P. W. Wolf, "Software Reuse through Building Blocks," *IEEE Software*, Vol. 4, No. 4, pp. 34-42, 1987.

[10] Martin, J. and C. McClure, "Diagramming Techniques for Analysts and Programmers," Prentice Hall, 1984.

[11] McCabe, T. J., "A Complexity Measure," *IEEE Trans. Software Eng.*, Vol 2, No 4, pp.308-320, 1976.

[12] McClure, C., *The Three Rs of Software Automation: Re-engineering,, Repository, Reusability,* Prentice Hall, 1992.

[13] Tracz, W., "Software Reuse Myths," *ACM SIGSOFT Software Emgineering Notes,* Vol.3, No.1, pp.17-21, 1988.

[14] Tracz, W., "Software Reuse: Motivators and Inhibitors," *Proc. COMPCON'87,* pp. 358-363, 1987.