



Guideline for Effective Module Reuse

by

Ittiphol Orachoonwong

Submitted in Partial Fulfillment of
the Requirements for the Degree of
Master of Science
in Information Technology
Assumption University

November, 1999

Guideline for Effective Module Reuse

by

Ittiphol Orachoonwong



**Submitted in Partial Fulfillment of
the Requirement for the Degree of
Master of Science
in Information Technology
Assumption University**

November, 1999

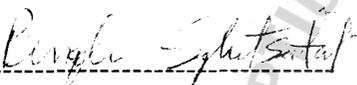
The Faculty of Science and Technology

Thesis Approval

Thesis Title Guideline for Effective Module Reuse
By Ittiphol Orachoonwong
Thesis Advisor Dr. Peraphon Sophatsathit
Academic Year 2/1997

The Department of IT, the Faculty of Science and Technology of Assumption University had approved this final report of the **twelve** credits course, **IT7000 Master Thesis**, submitted in partial fulfillment of the requirements for the degree of Master of Science in Information Technology.

Approval Committee:



(Dr. Peraphon Sophatsathit)
Advisor



(Prof. Dr. Chidchanok Lursinsap)
MUA Representative



(Dr. Vichit Avatchanakorn)
Committee



(Dr. Jirapun Daengdej)
Committee

Faculty Approval:



(Dr. Thotsapon Sortrakul)
Director



(Asst. Prof. Dr. Pratit Santiprabhob)
Dean

November / 1999
Month/Year

ACKNOWLEDGMENTS

I would very much like to express my deep gratitude to my advisor, Dr. Peraphon Sophatsathit, who provide endured and patiently answered dozens of questions, who put in the usual long hours and check every line of my paper. Without his constructive criticism and suggestion, this work would not have been possible.

I would like to thank all of the committee members, Dr. Vichit Avatchanakorn, Dr. Jirapun Daengdej, and Prof. Dr. Chidchanok Lursinsap for graciously spending their valuable time and efforts in reviewing my thesis.

My thanks go to all faculty members and secretaries for their encouragement, and also to acknowledge all the colleagues at S&T NOC, INFORES, and JAVA Center for helping me manage many things. I would like to thank all my friends (Pawut, Vatha, and Areya), my brothers (Paitoon, Pipat, Sakon, Keattisak, Sanguan and Suparwat) and my cute sisters (Pornpimon, Supannika, Phikul, Sudaporn and Sudsui), who kept in touch with me.

I, however, must apologize for being unable to name the rest of him or her individually in this limited space. I am aware that without their own initiative and support, I might never have had a source of inspiration to write my thesis like this.

ABSTRACT

Recycling of existing code, or code reuse, simplifies the task of software development considerably. Traditional code reuse encompasses the extent to which code can be used in different applications with minimal change. This paper proposes a study on code reuse to determine some guidelines for writing new code by means of reuse. The main objective is to establish a framework for reuse component so as to increase software development productivity and quality.

The results of this study confirm the benefits of code reuse over conventional reinventing the wheel approach. One distinct characteristic so obtained is that a reusable component is easier to incorporate into existing software under construction than to reinvent it with fewer errors and efforts to complete.

Keywords: Code Reuse, Modularity, and Reusable Component.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	i
ABSTRACT	ii
LIST OF FIGURES	iii
LIST OF TABLES	iv
CHAPTER 1: INTRODUCTION	1
1.1 Background	1
1.2 Motivation	1
1.3 Objectives	4
1.4 Scope of work	5
CHAPTER 2: AN APPROACH FOR CODE REUSE	7
2.1 Framework for building reusable code	7
2.2 Proposed Guidelines	11
CHAPTER 3: EXPERIMENT AND ANALYSIS	17
3.1 Experiment conditions and evaluation metrics	17
3.2 Example of module reuse	19
3.3 Experiment environment	27
3.4 Experiment results and analysis	68
CHAPTER 4: CONCLUSIONS	76
REFERENCES	78
APPENDIX A: MaCabe and Halstead Complexity Metrics	80
APPENDIX B: A sample calculation the productivity and the quality	82
APPENDIX C: Program structure chart in the experiment	85

LIST OF FIGURES

Figure 1-1: An objective of proposed guidelines	5
Figure 2-1: a) Tight coupling and b) Loosely coupled	7
Figure 2-2: Tightly coupled of no. of credits value	8
Figure 2-3: Loosely coupled of Tcredit into a function	8
Figure 2-4: Modularity model of both GPA Computing Module and Register Fee Module	9
Figure 3-1: A procedure NameFormat to be selected for a reusable component	20
Figure 3-2: A new DataFormat function built from adapting old procedure	21
Figure 3-3: Redundancy code obtained from modifying DataFormat function	22
Figure 3-4: SetName changed to constant data	23
Figure 3-5: A practical reusable component	24
Figure 3-6: Program structure chart of Electronic Phone System developed by conventional approach	25
Figure 3-7: Program structure chart of Electronic Phone System developed by reuse approach	26
Figure 3-8: Comparison of both (1) traditional approach and (2) reuse approach	69
Figure 3-9: Comparison of three style of programming languages based on productivity.	73
Figure 3-10: Graphical comparison of programming languages based on quality.	73

LIST OF TABLES

Table 1-1: Comparison guidelines for code reuse by means of feature perspectives	3
Table 3-1: Representing characteristics of new reuse component in Get_Sentence Module.	29
Table 3-2: Representing characteristics of new reuse component in RandomValue Module.	31
Table 3-3: Representing characteristics of new reuse component in Binary Search Module.	33
Table 3-4: Representing characteristics of new reuse component in Move_Elevator module.	35
Table 3-5: Representing characteristics of new reuse component in Shuffle Module.	37
Table 3-6: Representing characteristics of new reuse component in BucketSort Module.	39
Table 3-7: Representing characteristics of new reuse component in Maze Traversal Module.	41
Table 3-8: Representing characteristics of new reuse component in DataMerge Module.	43
Table 3-9: Representing characteristics of new reuse component in SpanWaveEven Module.	45
Table 3-10: Representing characteristics of new reuse component in both Open_Source and Print_Message modules.	47

Table 3-11: Representing characteristics of new reuse components in both Similarity Search and Display Module.	49
Table 3-12: Representing characteristics of new reuse component in CheckFormat Module.	51
Table 3-13: Representing characteristics of new reuse component in Distance_Coordinate module.	53
Table 3-14: Representing characteristic of new reuse component in BinaryTreeSearch module.	55
Table 3-15: Representing characteristics of new reuse component in NewBinSearch module.	57
Table 3-16: Representing characteristics of new reuse component in GenRandomDigit module.	59
Table 3-17: Representing characteristic of new reuse method in NewBinTree object.	61
Table 3-18: Representing characteristic of new reuse method in MovieTicketBooth object.	63
Table 3-19: Representing characteristic of new reuse method in SoftdrinkVendingMachine object.	65
Table 3-20: Representing characteristics of new reuse method in TicketVendingMachine object.	67
Table 3-21: Results comparing code reuse and conventional code writing	68
Table 3-22: Comparison of productivity and quality using LOC measurement	70
Table 3-23: Comparison of productivity and quality using complexity measurement	71

Table 3-24: Comparison of programming languages based on productivity	72
Table 3-25: Comparison of programming languages based on quality	72



CHAPTER 1 INTRODUCTION

1.1 BACKGROUND

Reuse is an efficient technique for saving resources and efforts. When we talk about reuse in writing software, code reuse should be the first thing that comes to mind in that reuse not only increases software productivity, but also entails better quality software, improves software system interoperability, and utilizes fewer manpower and efforts.

Implementing a reuse discipline entails more than creating and using reuse libraries. It requires formalizing the practice of reuse by including support for reuse in software development methods, tools, training, incentives, and measurements. Unless reuse is built explicit and formalized, an organization will not be able to repeatedly exploit reuse opportunities in multiple software projects. Without this repetition, the improvements to the software process that result from reuse will be limited.

1.2 MOTIVATION

Today most application domains are computerized as many organizations are searching ways to increase software development productivity, timeliness of delivery, and long term reduction in costs of software development and maintenance. As such developers must change their coding discipline to cope with development life cycle.

Before getting on to reuse fundamentals, the principle aspect of code reuse is that code incorporated into software under construction can be reused from other programs in the form of modular routines. Such an application can be carried out easier than writing similar code that spreads across a large routine [4]. As such, developers are looking for ways of writing less of code [5], or search for ways of writing code faster.

Toshiba Software Factory reported 50% reuse over its product line in 1989 which resulting in productivity increased by 57% [7]. GTE Data Services reached a 50% reuse level in the third year of their reuse program which yielded in \$12 million savings [8]. And US Naval Surface Warfare Center, Virginia Beach, VA [2] built a common architecture for a family of combat direction systems by mean of reuse. On fourteen ship system upgrades, the center achieved a level reuse of 89-99%, a 3-fold reduction in defects, and an 8-to-10 fold increase in productivity. These are some examples of reuse success stories.

Although object-oriented technologies support and encourage the reuse asset, they don't explicitly describe how to determine the reuse potential of a component such as an object class or framework [9]. In addition, it is important to understand that using object technology does not result in automatic reuse. Although object technology [1] does encourage a reuse mindset and object-oriented techniques such as encapsulation and inheritance support reuse, object technology and reuse are not one and same thing. Furthermore, object technology is neither sufficient nor essential to reuse. We can say that object technology is not a prerequisite for reuse.

In Japan, object technology is not necessarily accepted in developing mission-critical systems, because it is difficult to understand [13] because many people are eager for rapid, inexpensive, and high-quality system development by reusing existing software components.

The potential for reuse is enormous [1], since the majority of each new application system could be assembled from reusable software parts, assuming that the appropriate parts could be predetermined, built, and made readily available to system developers.

There are, however, many reasons for reuse failures precipitating from the efforts which have been blocked by many serious management and technical obstacles, such as

lack of understanding about why to practice reuse, no tools to support the practice of reuse, no methodology support for reuse and so forth. One reason of reuse failure which coincides to the objective of this work is a set of guidelines for code reuse.

A good reason of establishing these guidelines is that each programming language has its own features and rules. There is no pre-defined approach as to how reuse should be carried out in each language. Thus, it is hard for the developers to find a set of common reusable guidelines that fits the developers' needs.

A preliminary literature review was conducted in order to understand the fundamental of code reuse in software development. Three source addressing different aspects of code reuse were selected to provide some principal aspects of reuse features. They are Software Reuse Guideline (Paper I) [15], Designing for Reuse (Paper II) [16], and Rationale for the Design of Reusable Abstract Data Type Implemented in Ada (Paper III) [17]. Table 1-1 shown the feature of guideline papers.

Table 1-1: Comparison Guidelines for Code Reuse by means of feature perspectives

Analysis perspectives	Guidelines for Code Reuse			
	Paper I	Paper II	Paper III	Proposed Guideline
a) Programming Language	Independent	Delphi	Ada	Independent
b) User Levels	Casual	Sophisticated	Casual	Sophisticated
c) Domain structure	Undefined	Semantic	Specialized	Specialized
d) Characteristics of Guideline	Advice	Practical	Advice	Advice

Paper I describes some general principles of modular design at a highly conceptual level, such as representation of module design abstraction by means of a high cohesion for closely-knitted code, or loose coupling for largely independent code. Unfortunately, these characteristics are too abstract and unmeasurable. Paper II and Paper III emphasize on specific programming languages such as Delphi and Ada,

respectively. Paper II considers a set of practical reuse guidelines such as how to make a method protected instead of private and how to avoid nested procedures or references to private part in virtual methods. On the other hand, Paper III offers useful advice such as why user interface must be clean, or avoid implementing a certain package in such a way that it maintains state in private variables.

The issues established by these papers represent application step in software development. There is no coherent conceptual framework that integrates all the steps into a set of common guidelines for software code reuse. This study, therefore, proposes a set of practical guidelines that serve as an easy to understand step by step code reuse approach.

These motivations and the lack of reuse methods reinforce to establish the standard guideline that language is independent and supports functional programming languages, object-based programming languages and object-oriented programming languages. By following these proposed guidelines, developers are not constrained to implement the objected-oriented technology by reuse existing components because the proposed guidelines aim to support an independent programming language by means of reuse. Furthermore, the proposed guidelines will help or improve generic code for reuse.

1.3 OBJECTIVES

The following describe the objectives of this thesis:

1.3.1) To present a step-by-step approach for code reuse in software development based on a simple premise that if a code segment cannot be changed easily, it is not a good candidate for reuse.

1.3.2) To establish a set of proposed guidelines for code reuse in small module software which preserve the stated.

1.3.3) To present the results of an experiment which demonstrate why the notion of code reuse is superior to reinventing the wheel.

The above objectives can be summarized as shown in Figure 1-1. The figure depicts how developers can methodically apply the proposed guidelines in their software development effort by combining existing code along with outside code to create potential (module) code for reuse.

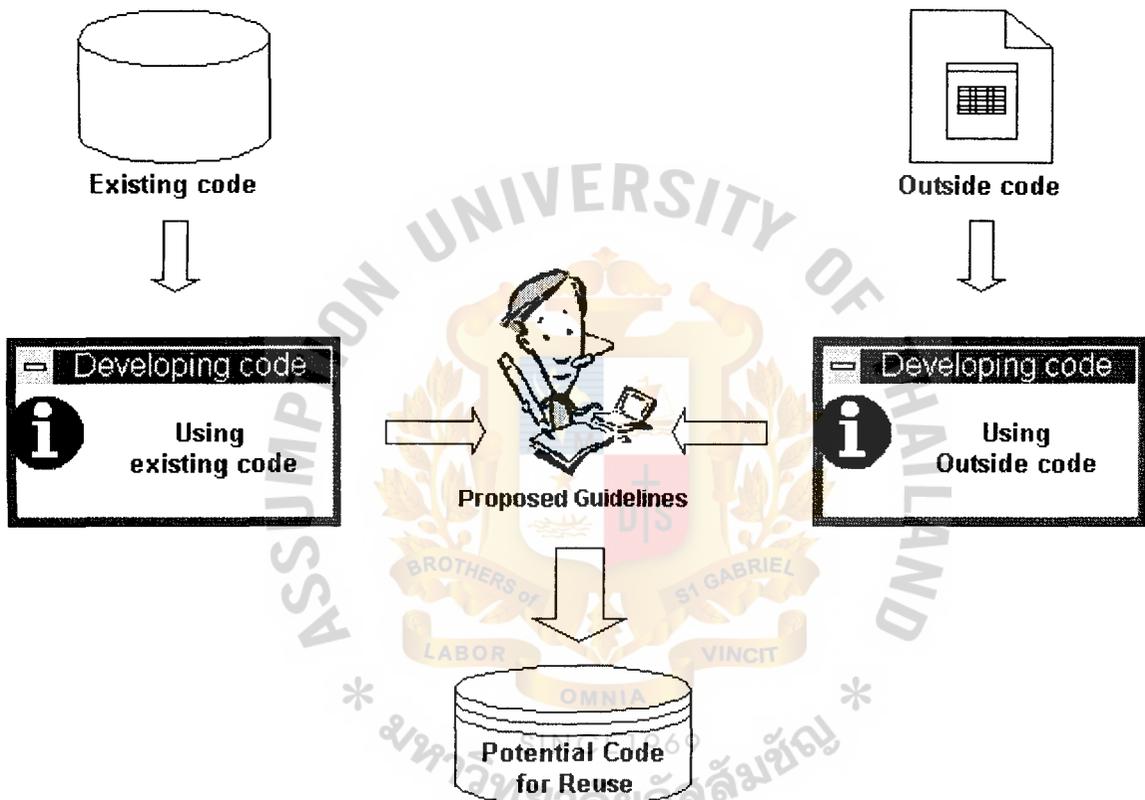


Figure 1-1: An objective of the proposed guidelines

1.4 SCOPE OF WORK

Based on the above objectives a framework for code reuse is defined so as to establish the scope of reuse. This thesis proposes a study on code reuse to determine a set of guidelines for writing new code by means of reuse for which scope of work are as follows:

- a) The proposed guidelines must build or improve new reusable components which correspond to the functional requirements or specifications.
- b) The proposed guidelines aim to support code reuse that must be independent of programming languages, i.e., procedural programming language, object-based programming language, and object-oriented programming language.
- c) The guidelines aim at supporting sophisticated end-users whose meet their complex requirements; Casual end-users are not incorporated.
- d) The small-scale experiments are set to verify the proposed guidelines which are based on small module code development less than 1500 lines of code.
- e) The step-by-step proposed guidelines is built so as to guide developer in creating potential reuse code (as illustrated in Figure 1-1).

Someone says that the best way of building instrument for saving energy is to build human resource. This notion is reflected by the proposed guidelines attempting to save the resources and programming efforts.

The remaining of this thesis is organized as follows: Chapter 2 establishes an approach for code reuse and the proposed guidelines. Some experiments and Analysis were conducted and described in Chapter 3 based on the above guidelines and the concluding remarks are given in Chapter 4.

CHAPTER 2: AN APPROACH FOR CODE REUSE

2.1 FRAMEWORK FOR BUILDING REUSEABLE CODE

The fundamental principle of building code for reuse is to construct reusable components that are easier to reuse than to reinvent. If a code segment cannot be changed easily, it is not a good candidate for reuse.

We proposed a framework for building reusable code having the following principal characteristics:

a) Coupling

We proposed reusable code to be loosely coupled over tightly coupled because any change in the data component of reusable code would not require major revisions in other code component. In other words, loosely coupled construct enables module to be detached or attached with minimal side effects as oppose to tightly coupled construct. This fact is illustrated by two coupling models in Figure 2-1.

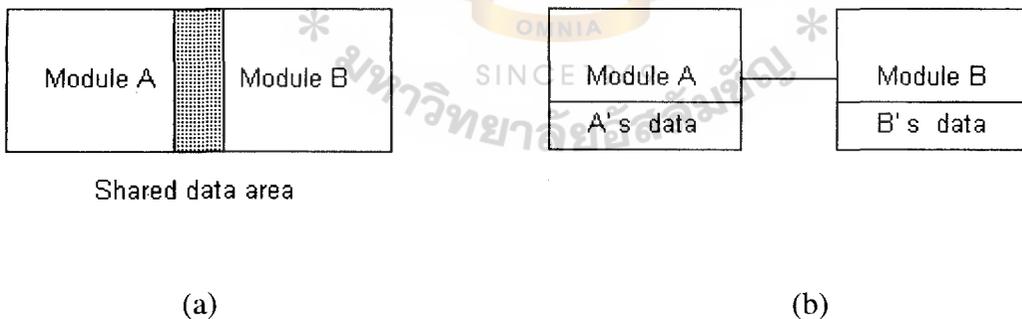


Figure 2-1: a) Tightly coupled and b) Loosely coupled

As depicted in Figure 2-2, the sample code computes GPA based on the number of subjects taken in order to determine the total credits (line 16) for use in GPA computation. The accumulated value of Tcredit, however, is also required by the Register module. As a consequence, Tcredit is a shared data area access by both GPA module and Register module.

```

1  (* This is a procedure to compute GPA of student *)
2  Procedure GPA_Computing(NoSubject : integer
3                          var Tcredit : integer);
4  Var
5    GPA, Temp : real;
6  Begin
7    while NoSubject > 0 do
8      begin
9        (* ReadData is a procedure for reading SubjectID and Grade form file *)
10       ReadData(SubjectID, Grade);
11
12       (* Credit() is a function to match the no. of credit per subject and
13          Vgrade() is a function to translate grade character to integer *)
14       Temp      := Temp + (Credit(SubjectID) * Vgrade(Grade));
15       NoSubject := NoSubject - 1;
16       Tcredit   := Tcredit + Credit(SubjectID);
17     end;
18   GPA := Temp / Tcredit;
19 End;

```

Figure2-2 : Tightly coupled of no. of credit value

On the contrary, development for reuse advocates a loosely coupled configuration because the components built are independent of the remaining code. As depicted in Figure 2-3, Tcredit is decoupled into a function for greater flexible reuse purposes.

```

1  (* Tcredit is a function to compute the total credit *)
2  Function Tcredit(NoSubject:integer): real;
3  Begin
4    while NoSubject > 0 do
5      begin
6        (* Credit() is a function to match the no. of credit per subject *)
7        Tcredit := Tcredit + Credit(SubjectID);
8        NoSubject := NoSubject - 1;
9      end;
10 End;
11
12 (* This is a procedure to compute GPA of student *)
13 Procedure GPA_Computing(NoSubject : integer
14                         var Tcredit : integer);
15 Var
16   GPA, Temp : real;
17 Begin
18   while NoSubject > 0 do
19     begin
20       (* ReadData is a procedure for reading SubjectID and Grade form file *)
21       ReadData(SubjectID, Grade);
22
23       Temp      := Temp + (Credit(SubjectID) * Vgrade(Grade));
24       NoSubject := NoSubject - 1;
25     end;
26   GPA := Temp / Tcredit(NoSubject);
27 End;
28
29 (* This is a procedure to compute the register fee of student *)
30 Procedure RegisterFee_Computing(NoSubject : integer);
31 Var
32   ReFee : real;
33 Begin
34   (* Tcredit() is a function compute total credit *)
35   ReFee := Tcredit(NoSubject) * PricePerCredit;
36   writeln(ReFee);
37 End;

```

Figure 2-3: Loosely coupled of Tcredit into a function.

b) Modularity

It is imperative that high degree of modularity [6] be desirable because it simplifies the design process and makes the overall product easier to maintain. As a consequence, developers are free to combine the components in a number of different ways to create new modular components for specific needs.

For example, if a developer wanted to compute the GPA and Registration Fee for every student, he could make total credit a reuse function to be invoked by both modules as shown in Figure 2-4.

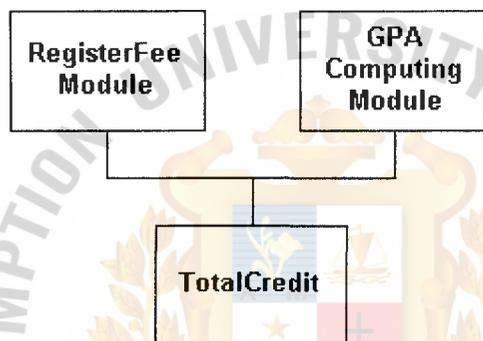


Figure2-4 : Modularity model of both GPA Computing Module and Register Fee Module.

c) Information Hiding

Information hiding [2] enables software developers to be free from delving into the implementation detail of the reuse components. Hence, the derived component's complexity is considerably reduced.

d) Abstract Data Type (ADT)

ADT serves as the format of building reusable code segment. The main purpose is to easily cement the difference between the existing component and the additional code into a unique component.

Based on the above framework, we shall address the problems of software development with code reuse step by step as follows:

□ Preliminary investigation

In this stage, the developers must search for applicable code to be incorporated into the system under investigation. Typical reusable pools include language and run-time libraries, built-in packages, etc.

□ Analysis

During system analysis stage, developers must know which component in the existing software can be replaced by reusable code. Before developers decide to choose the candidate code segment, they have to analyze the design to ensure that the candidate code segment performs the required function. Should there be any modification, it must be kept to minimal. The analysis also encompasses the reuse domain of the applicable candidate code for compatibility and correctness with the rest of the system.

□ Design

This is the heart of code reuse in that developers must have a clear view as to where the reuse component will be integrated into the overall structure of the software. Code reuse will help improve the quality of software design owing to the aforementioned framework. In addition, development time and cost will also be reduced as more reusable components are being utilized. The resulting component can be used by subsequent modular component interchange.

2.2 PROPOSED GUIDELINES

Based on the framework discussed, the guidelines for adapting reuse components that meet the reuse specification are as follows:

1. Look ahead to adapt and/or modify the reused component to take full advantage of reuse opportunities.

Building application systems from reusable components is based on the assumptions that reusable components exist somewhere, they are reasonably easy to find and understand, and they are of good quality. So if the time to search for candidate reusable components is too long in the developer's eyes, developers should decide to build the component from scratch rather than reusing existing components.

Developers must identify the code segment in order to locate and select candidate components suitable for the desired system functional requirements. The procedural guidelines for looking ahead and selecting the existing reusable components are follows as:

- 1) Determine the reuse component that corresponds to the functional requirements.
- 2) Determine any available application packages or programming languages that fit the type of the above reuse component.
- 3) While searching for existing components to be reused in whole or in part, gather not only code but also the associated documentation.
- 4) Assess the quality of candidate components, as quality is key to reusability by considering whether
 - 4.1) it is properly modularized and well-documented,
 - 4.2) it is furnished with history of use, and
 - 4.3) it has a reliable record.

5) Assess the portability of the candidate component suitable for reuse specification as follows:

- 5.1) special or additional tools required to support the use of the candidate component,
- 5.2) constraints imposed by the candidate components, e.g., for example, programming language, platform specific requirements, etc.,
- 5.3) savings from reuse components as opposed to building them from scratch. Carma McClure [9] suggests a simple rule of thumb: “The estimated saving should be at least 30 percent to make it worth reusing.”,
- 5.4) performance gain from reuse,
- 5.5) candidate component supporting loosely coupling, and
- 5.6) the extent of modification required for the reused components within the functional requirement.

Before developers decide which candidate components to choose, they need to define how to measure reuse potential of the components. The reusability of a component is measured in terms of the percent of reuse guidelines satisfied. Reuse potential is the ratio of number of guidelines satisfied by a component and the total number of guidelines that are applicable. A component [10] may be:

1. *Weakly reusable* whose potential for reuse is low, implying that the reused code satisfies fewer than 50% of the guidelines. It needs more efforts to redesign the original component for reuse.

2. *Limitedly reusable* whose potential for reuse is high, implying that the reused code satisfies between 50-70% of the relevant guidelines and needs some efforts to improve it.

3. *Strongly reusable* whose potential for reuse is high, implying that the reused code satisfies between 70-90% of the relevant guidelines and needs little modification to improve it.

4. *Immediately reusable* whose potential for reuse is very high, implying that the reused code satisfies more than 90% of the relevant guidelines and this can be reused as is without modification.

There are two alternatives of component reuse, namely,

- (1) *Black-box adapting* where the new functional requirements adapt the entire selected component structure or design either from existing code or outside code (see Figure 1-1) to fit the reuse component.

Black-box adapting requires that, developers focus on the functional requirements of the software by adjusting the interface of reuse components corresponding to the functional requirements to be a reuse component, e.g., changing data type (from float to double in C language). Nevertheless, from OOP prospective, developers may have to adapt or adjust certain private methods to protected or public methods in order for reusability.

- (2) *White-box modifying* where the selected component is modified to create a new reuse component.

White-box modifying is to examine the procedural detail of the selected components in order to ensure that the code meets requirements specification, as well as improve processing performance. Primary modifications include data structure change to comply with the requirement specification. For example, to accommodate unknown number of inputs where the desired reuse component employs array structure, it is essential that this array structure must be changed to some dynamic data structures such as linked-list, stack, etc.

2. Investigate the code for redundancy to help identify opportunities for applying additional reusable components.

When multiple software components provide the same function, or serve the same purpose, or define the same data, redundancy can occur in the designated domain or sub-domain. The developer should look for the redundant code in that domain to apply candidate reuse components.

A direct approach to program investigation employs flow graph tools [6] to locate redundant code through visual examining the resulting flow graphs. Such redundant code decreases the quality and performance of software. To avoid those problems, developers should check for redundancies by examining [9] the code that:

- 1) has the same or similar names,
- 2) uses the same or similar input and produces the same or similar output,
- 3) satisfies the same or similar requirements,
- 4) has the same or similar flow graphs,
- 5) has the same or similar complexity values based on McCabe Complexity metrics [14] or Halstead Software Science Metrics [14]. (See Appendix A.)

As redundant code is identified, developers should create a new code module or component to replace the redundant module. The resulting component code will support various newly acquired or established implementations which can be subsequently reused.

Remember that make more reusable, developer should limit size of component in mind such as function, procedure, class, or methods because it 's easy to understand and maintenance.

3. Follow the empirical frameworks while creating new components of reuse.

Developers should analyze the reuse specification flexibility and extensibility to establish code inter-dependence. This is equivalent to creating new components. To ensure this reusability framework, the code component should have the following characteristics:

- ❑ Loosely Couple between components
- ❑ High Cohesion within the component
- ❑ Modularity and encapsulated for reuse
- ❑ Simple Interface

Some mechanisms conducive to the above characteristics should involve

- minimal no. of parameters to decrease potential errors and processing time,
- minimal no. of reads and writes in the reused component, and
- proper comments to clearly describe the reused component.

4. Assess the reuse specification corresponding to the functional requirements and specification for reuse archival purpose.

In completing a domain reuse, the component must be checked against functional requirements and, if possible, archived future reuse by means of standard testing methods via black-box testing [1]. Evaluation is typically determined by testing its inputs and the corresponding outputs. All assessed components represent new generic versions of code. The new reuse component must be explicitly organized to categorize how individual component is functionally similar or different. This is equivalent to building relations to a collection of objects.

Developers can use naming scheme of parameterization and/or inheritance hierarchy to denote all the components captured as synonyms for subsequent searches and retrievals. Such activities, in effect, serve as the compilation step of archiving a code repository for future development reuse.



CHAPTER 3 EXPERIMENT AND ANALYSIS

3.1 EXPERIMENT CONDITIONS AND EVALUATION METRICS

In this chapter, a set of experiments is conducted to validate the proposed guidelines by means of reuse approach in comparison to code rewrite approach. The experiment exercised twenty small programs which involved more than fifty components. The nature of the problems based primarily on Transaction Processing Program (TPP) are solved by various programming languages such as C, Pascal, Delphi, and Java.

In each experiment, the same programmer was assigned to complete a program by means of both approaches which were controlled by the similar functional requirements and programming language.

The experiments employed traditional software metrics, namely, time spent, number of errors, and lines of code (LOC), to measure programming efforts and efficiency. As such, the result of each experiment focused on the followings:

1. *Time spent*, starting immediately after the programmer understood program requirements and began designing/modifying the program;
2. *No. of errors*, encompassing all errors after the program has passed compilation step. This includes any improper outputs resulting from invalid requirements; and
3. *Lines of Code*, comparing quantitatively the size of the program obtained from both approaches.

Results of the experiment were analyzed to yield the productivity and quality of the software product below.

1. Productivity and quality by means of Lines of Code

Size-oriented software metrics [6] are direct measures of software and the process by which it is developed. Those formulas involve:

$$\text{Productivity} = \text{KLOC/person-hour}$$

$$\text{Quality} = \text{defects/KLOC}$$

2. Productivity and quality by means of complexity

Function-oriented software metrics [6] are indirect measures of software and the process by which it is developed. Rather than counting LOC, function-oriented metrics focus on program “functionality” or “utility” based on function point measure.

To compute function point, the following relationship is used:

$$\text{FP} = \text{count-total} * [0.65 + 0.01 * \text{SUM}(F_i)]$$

where count-total is the sum of all entries obtained such as no. of user inputs, no. of user output, no. of files, and F_i are complexity adjustment values based on responses to questions noted. Resulting productivity and quality can then be computed in a manner analogous LOC as follows:

$$\text{Productivity} = \text{FP / person-hour}$$

$$\text{Quality} = \text{defects / FP}$$

3.2 EXAMPLE OF MODULE REUSE

This section demonstrates how to build a new reusable component by following the proposed guidelines in step by step using an Electronic Phone System as case study.

An Electronic Phone System (EPS) was chosen to demonstrate the viability of the proposed guidelines. The EPS is a programming system that collects personal information such as first name, surname, address, and telephone number for on-line inquires. One of the requirement specifications is to validate input data from the user and to gather all invalid inputs in an error_log file. Based on the proposed guidelines and this experiment specifications, a reusable component search plan for solving this problem was laid out as follows:

- a) Type of the reuse domain - examine functions of which process name, address, or telephone.
- b) Domain description - validate input data typed in by the user which are primarily characters.
- c) Program constraints - Functions written in Pascal or C.
- d) Associated documentation - if required.
- e) Quality of candidate components - reliability record and history of reuse.

The process proceeded as shown below.

Assuming that developer has already passed to search and select the existing reusable components in accordance with reuse specifications. Because in this section is to attempt how to modify the reuse module from candidate module. Supposedly, NameFormat procedure was selected as a candidate for reuse. The example code is shown in Figure 3-1.

```

(* This is a procedure for check format data type of name *)
Procedure NameFormat      (Min,Max : integer;
                          Name      : string);
Var
SetName : set of char;      (* To declare available set of name type *)
Long, i : integer;
ErData  : boolean;         (* To declare the status of error data *)
Begin
SetName := ['A'..'Z', 'a'..'z']; (* To initial the set of name *)
Long    := length(Name);      (* To measure the length of data *)

(* Condition if, for exam the length of data *)
if (Long > (Min-1)) and (Long < (Max+1)) then
  begin
    (* To initial data of i for start at the first position in string *)
    i := 1;

    (* Repeat loop, for exam the invalid data type in the set of name *)
    repeat
      if Name[i] in SetName then
        (* To increase i for going to the next position *)
        i := i + 1
      else
        begin
          writeln('Invalid in data type of Name');
          Erdata := true; (* To declare the status of error data *)
        end;
      until (i:=(Long+1)) or (Erdata = true);
    end
  else
    begin
      (* Condition if, for exam the minimum data input *)
      if (Long < Min) then
        writeln('Error in the minimum data input');

      (* Condition if, for exam the maximum data input *)
      if (Long > Max) then
        writeln('Error in the maximum data input');
    end;
  end;
End;

```

Figure 3-1: A procedure NameFormat to be selected for a reusable component

For first step of the guidelines, we look ahead to adapt and/or modify the reused component to take full advantage of reuse opportunities.

Adaptation and modification were required to tailor the above procedure suitable for reuse. Figure 3-2 shows the code adjustment from programmer in bold. The procedure was converted to a function on line no. 2 because new function and variable names were established to convey a more meaningful program understanding in the form of general aliases. According to the adapting interface (Black-box), the function was required to perform code change (White-box) on line no. 12, 26, and 40.

```

1 (* This is a function to check format data type of name. *)
2 Function DataFormat (Min,Max : integer;
3                       Data : string) : boolean;
4 Var
5   SetName : set of char;   (* To declare variable set of name type *)
6   Long, i : integer;
7   ErData : boolean;       (* To declare the status of error data *)
8
9 Begin
10  SetName := ['A'..'Z', 'a'..'z']; (* To initial data set of name *)
11  Long := length(Data);           (* To measure the length of data *)
12  DataFormat := true;           (* To initial status of NameFormat *)
13
14  (* Condition if, for exam the length of data *)
15  if (Long > (Min-1)) and (Long < (Max+1)) then
16    begin
17      i := 1; (* To initial i for start at the first position *)
18
19      (* Repeat loop, to examine the invalid data type of name *)
20      repeat
21        if Data[i] in SetName then
22          i := i + 1 (* To increase i in order to go to the next position *)
23        else
24          begin (* Invalid in data type of name *)
25            writeln('Invalid in data type of Name');
26            DataFormat := false; (* To declare the status of error data *)
27          end;
28        until (i:=(Long+1)) or (DataFormat = false);
29      end
30    else
31      begin
32        (* Condition if, for exam the minimum data input *)
33        if (Long < Min) then
34          writeln('Error in the minimum data input');
35
36        (* Condition if, for exam the maximum data input *)
37        if (Long > Max) then
38          writeln('Error in the maximum data input');
39
40        DataFormat := false; (* To declare the status of error data *)
41      end;
42 End;

```

Figure 3-2: A new DataFormat function built from adapting old procedure

St. Gabriel's Library

The second step is to investigate the code for redundancy to help identify opportunities for applying additional reusable components.

An approach to redundant code investigation employs flow graph tools to locate redundant code with the help of visual examining. As depicted in Figure 3-3, a new reusable component was promoted replacing redundancy code on line no. 25, 34, and 38 respectively. The new ErrorLog procedure generalized error invocation stemming from different error types, for example, ErrorLog(21) on line 34 means that the user inputs fewer data than the minimum requirement.

Additional redundant code founded by flow graph is obtained from redundancy checking on line 33 and 37 shown in Figure 3-2.

```
1 (* This is a function to check format data type of name. *)
2 Function DataFormat (Min, Max : integer;
3     Data : string) : boolean;
4 Var
5     SetName : set of char;      (* To declare variable set of name type *)
6     Long, i : integer;
7     ErData : boolean;         (* To declare the status of error data *)
8
9 Begin
10    SetName := ['A'..'Z', 'a'..'z']; (* To initial data set of name *)
11    Long := length(Data);          (* To measure the length of data *)
12    DataFormat := true;           (* To initial status of NameFormat *)
13
14    (* Condition if, for exam the length of data *)
15    if (Long > (Min-1)) and (Long < (Max+1)) then
16        begin
17            i := 1; (* To initial i for start at the first position *)
18
19            (* Repeat loop, to examine the invalid data type of name *)
20            repeat
21                if Data[i] in SetName then
22                    i := i + 1 (* To increase i in order to go to the next position *)
23                else
24                    begin
25                        ErrorLog(11);          (* To send code 11 to ErrorLog function *)
26                        DataFormat := false; (* To declare the status of error data *)
27                    end;
28            until ( i:=(Long+1)) or (DataFormat = false);
29        end
30    else
31        begin
32            (* Condition if, for exam the minimum data input *)
33            if (Long < Min) then
34                ErrorLog(21) (* To send code 21 to ErrorLog function *)
35            else
36                (* Condition if, for exam the maximum data input *)
37                if (Long > Max) then
38                    ErrorLog(31);          (* To send code 31 to ErrorLog function *)
39
40                DataFormat := false; (* To declare the status of error data *)
41            end;
42    End;
```

Figure 3-3: Redundancy code obtained from modifying DataFormat function

The third step is to follow the empirical frameworks while creating new components of reuse.

According to the code of DataFormat function, SetName (Line 10 of Figure 3-2) was tightly coupled with a set of alphabet. As such, SetName was changed to a constant variable as depicted in Figure 3-4.

```
Program ElectronicPhone (Input, Output);
Type
  SetChar = Set of char;
Var
  SetName,           (* To declare a SetName in global variable *)
  SetAddress,       (* To declare a SetAddress in global variable *)
  SetPhone : SetChar; (* To declare a SetPhone in global variable *)
.
.
(* This is a procedure to initialize the set of Name, address, and phone number *)
Procedure InitializeDataSet(var SetName, SetAddress, SetPhone : SetChar);
Begin
  SetName := ['A'..'Z', 'a'..'z'];
  SetAddress := ['A'..'Z', 'a'..'z', 0..9, '/', '.'];
  SetPhone := [0..9];
End;
```

Figure 3-4: SetName changed to constant data.



Finally, the last step of guideline advises developer to assess the domain reuse specification corresponding to the functional requirements and specification for reuse archival purpose.

The sample code was further generalized for reuse as shown in Figure 3-5.

```

1 (* This is a function for check format data type of name *)
2 Function DataFormat (Min,Max : integer;
3                     DataSet : SetChar;
4                     Data    : string) : boolean;
5 Var
6   Long, i : integer;
7   ErData  : boolean;      (* To declare the status of error data *)
8
9 Begin
10
11   Long      := length(data);      (* To measure the length of data *)
12   DataFormat := true;            (* To initial status of NameFormat *)
13
14   (* Condition if, for exam the length of data *)
15   if (Long > (Min-1)) and (Long < (Max+1)) then
16     begin
17       i := 1; (* To initial value of i for start at the first position *)
18
19       (* Repeat loop, to examine the invalid data type of name *)
20       repeat
21         if Data[i] in DataSet then
22           i := i + 1 (* To increase i for going to the next position *)
23         else
24           begin
25             (* Invalid in data type of name *)
26             ErrorLog(11);      (* To send code 11 to ErrorLog function *)
27             DataFormat := false; (* To declare the status of error data *)
28           end;
29           until ( i:=(Long+1)) or (DataFormat = false);
30         end
31       else
32         begin
33           (* Condition if, for exam the minimum data input *)
34           if (Long < Min) then
35             ErrorLog(21);      (* To send code 21 to ErrorLng function *)
36           else
37             (* Condition if, for exam the maximum data input *)
38             if (Long > Max) then
39               ErrorLog(31);    (* To send code 31 to ErrorLog function *)
40             end;
41           DataFormat := false; (* To declare the status of error data *)
42         end;
43       End;

```

Figure 3-5: A practical reusable component

The example experiment affects the effective reusability as shown in program structure chart, comparing developed with conventional approach and developed with reuse approach. Figure 3-6 and Figure 3-7 represent the same program as program structure chart conducting with conventional approach and reuse approach respectively.

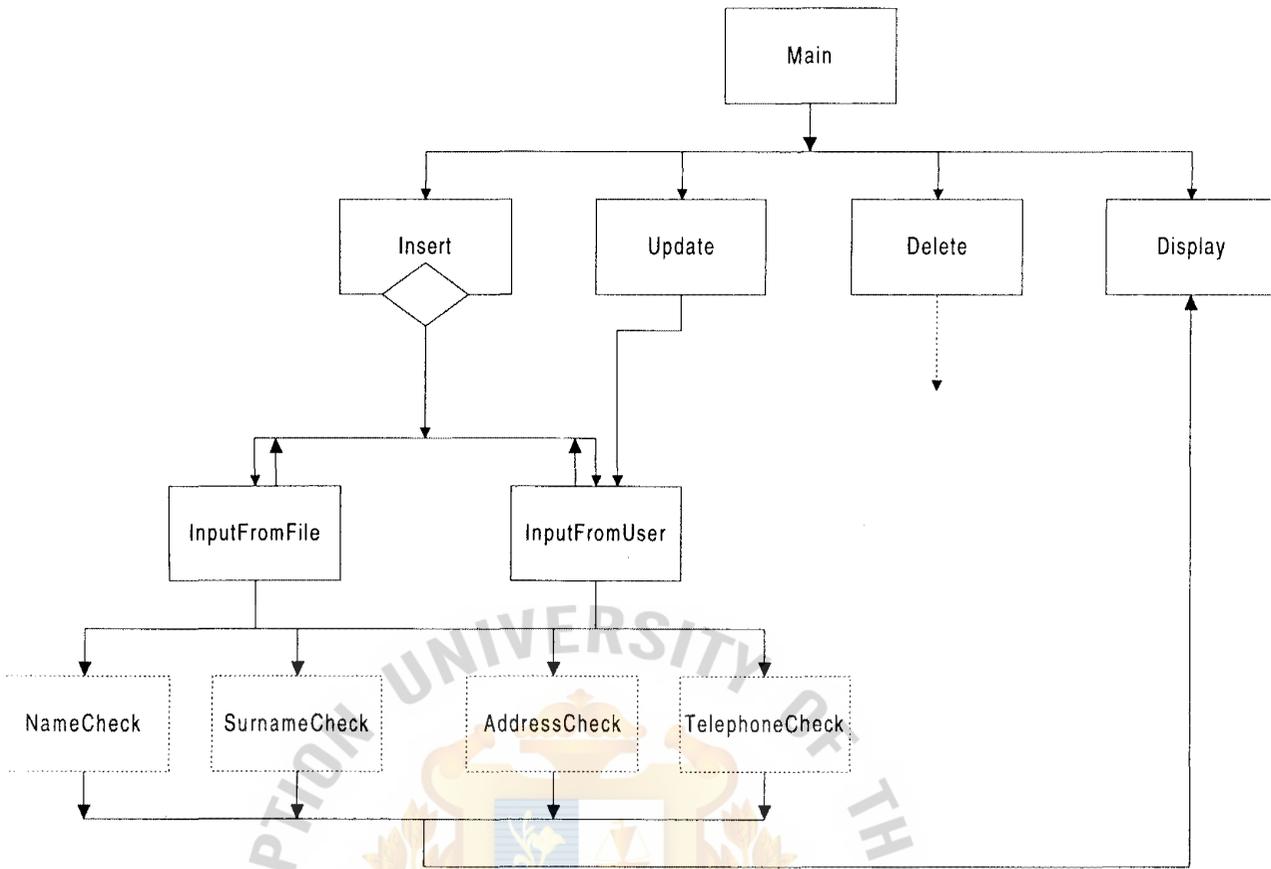


Figure 3-6: Program structure chart of Electronic Phone System developed by conventional approach

In program structure chart of Figure 3-6, Insert module requires 4 components for checking data, namely, Name, Surname, Address, and Telephone number. On the contrary, the reuse approach only requires FormatCheck module.

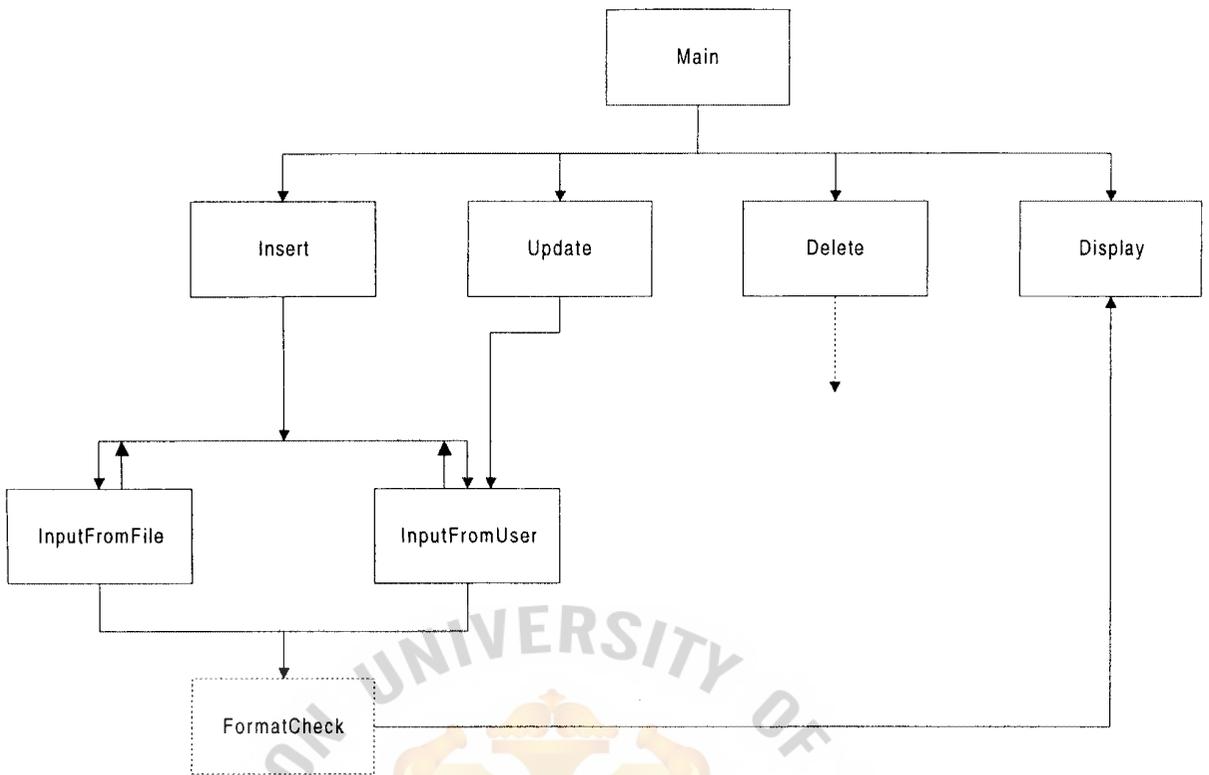


Figure 3-7: Program structure chart of Electronic Phone System developed by reuse approach

Based on the above example, we can compare the percent of reusability of both approaches by

$$\text{PercentageOf Reusability} = 100 \times \frac{\sum \text{NumberOf Re useInEach ReusedItims}}{\text{TotalNumberOfItems}}$$

Note: (1) Each reuse item of developed with conventional approach is reused twice.

(2) DataFormat item of developed with reuse approach is reused eight times.

Calculation

$$\begin{aligned}\text{Reusability(Conventional approach)} &= 100 \times \frac{[2+2+2+2]}{20} \\ &= 40 \% \\ \text{Reusability(Reuse approach)} &= 100 \times \frac{[8]}{17} \\ &= 47.05\%\end{aligned}$$

The reusability results confirm that the increasing percent of reusability of reuse approach is greater than the reusability of conventional approach.

3.3 EXPERIMENT ENVIRONMENT

Twenty experiments are employed to validate the proposed guidelines using conventional approach and reuse approach. Each experiment comprises different modules under different in environment. Thus, each experiment has different environment factors to create the desired module. The description of each sample experiment explains all relevant details to the module.

St. Gabriel's Library

Experiment #1: Get_Sentence

1. Objective: To develop a new Get_Sentence module.
2. Propose: To build a new GetSentence component inside Get_Sentence module by reusing an existing component.
3. Program environment:
 - 3.1 Language: Pascal
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: IDE (Turbo Pascal)
4. Characteristic of module:
 - 4.1 Function: to collect a first sentence of string input.
 - 4.2 No. of component inside: 1 component
5. Candidate of existing reuse component: GetWord procedure
6. Description:

A candidate component (GetWord) will read all characters in string and collects a word by founding first blank character. That is similar idea, a new component (GetSentence) will collect a sentence when a full stop founded. A program structure chart is depicted in Figure C-1 (See Appendix C).
7. Procedure:
 - 7.1 Changing title of a candidate component (GetWord) to a proper new component (GetSentence) that corresponds to the requirement specification.
 - 7.2 Modifying a little bit inside new component (GetSentence) from a blank character to a full stop character (.).

The characteristics of developed reuse component are represented in Table 3-1.

Table 3-1: Characteristics of new reuse component in Get_Sentence Module.

Component Name	Structure	Cyclomatic Complexity
GetSentence	Procedure	2

8. Conclusion:

A GetSentence component is developed by modifying the candidate component to be a new Get_Sentence module. So, a Get_Sentence module is a new effective module for reuse archive proposed.



Experiment #2: Interactive Multiplication

1. Objective: To develop a new RandomValue module
2. Propose: To build a new RandValue component inside RandomValue module by reusing existing component.
3. Program environment:
 - 3.1 Language: C
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: GCC
 - 3.4 Editor: Notepad
4. Characteristic of RandomValue module:
 - 4.1 Function: To random value via parameter dependent
 - 4.2 No. of component inside: 1 component
5. Candidate reuse component: rand function
6. Description:

Normally, a candidate component (rand function) gives a variable value such as 1-1,000,000 digits which exceed the specification. So, a new component (randValue function) is used to identify the limited boundary of random value according to the specification. A program structure chart is depicted in Figure C-2 (See Appendix C).
7. Procedure:
 - 7.1 Develop a new component (randValue function) which calls a candidate component (rand function) directly without change, allowing immediate reuse of a candidate component (rand function).
 - 7.2 The result of candidate component (rand function) is modulated with 10 (in case of one digits control), 100 (in case of two control), and 1000 (in case of three digit control) to yield the desired number of output digits.

The characteristics of developed reuse component represented in Table 3-2.

Table 3-2: Represent characteristic of new reuse component in RandomValue Module.

Component Name	Structure	Cyclomatic Complexity
RandValue	Function	3

8. Conclusion:

The candidate component can be reused without change (Immediately reusable) through RandValue component to build a new RandomValue module. So, a RandomValue module is a new effective module for reuse archive proposed.



Experiment #3: Binary Search

1. Objective: To develop a new Binary Search module
2. Propose: To build a new biSearchChar component inside Binary Search module by reusing an existing component.
3. Program environment:
 - 3.1 Language : C
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: IDE (Turbo C)
4. Characteristic of Binary Search module:
 - 4.1 Function: To search a student's name in from of binary search.
 - 4.2 No. of component inside: 1 component
5. Candidate reuse component: biSearchInt function
6. Description:

A key search of candidate component (biSearchInt function) is the desired value. The requirements specification of a new component (binSearchChar function) is to search a student's name using binary search technique. A program structure chart is depicted in Figure C-3 (See Appendix C).
7. Procedure:
 - 7.1 Changing name of a candidate component (biSearchInt function) to a proper alias component (biSearchChar function)
 - 7.2 Adapting data type of interface component (biSearchChar function) from integer (int) to character (char) that corresponds to the requirements specification.

The characteristics of developed reuse component represented in Table 3-3

Table 3-3: Representing characteristics of new reuse component in Binary Search Module.

Component Name	Structure	Cyclomatic Complexity
BiSearchChar	Function	3

8. Conclusion:

The candidate component can be reused with minor change (Strongly reusable) in BiSearchChar component to build a new Binary Search module. So, a Binary Search module is a new effective module for reuse archive proposed.



St. Gabriel's Library

Experiment #4: Elevator Simulation

- 1 Objective: To develop a new Move_Elevator module.
- 2 Propose: To build three components inside Move_Elevator module by reusing existing component.
- 3 Program environment:
 - 3.1 Language : C
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: IDE (Turbo C)
- 4 Characteristic of module:
 - 4.1 Function: To compute the effective distance of elevator movement
 - 4.2 No. of component inside: 3 components
- 5 Candidate of existing reuse component: isMoving function
- 6 Description:

Two components (TimeOfArrival and TimeOfBreaking function) require writing code from scratch, except isMoving function. A program structure chart is depicted in Figure C-4 (See Appendix C).
7. Procedure:
 - 7.1 Take the existing reuse function to be a component in the new created module (Move_Elevator).
 - 7.2 Add two functions named as TimeOfArrival and TimeOfBreaking to the new created module.
 - 7.3 Add a candidate component (isMoving) to Move_Elevator module.

The characteristics of developed reuse component represented in Table 3-4.

Table 3-4: Representing characteristics of new reuse component in Move_Elevator module.

Component Name	Structure	Cyclomatic Complexity
IsMoving	Function	1

8. Conclusion:

The candidate component can be reused without change (Immediately reusable) to build a new Move_Elevator module. So, a Move_Elevator module is a new effective module for reuse archive proposed.



Experiment #5: Card Game

1. Objective: To develop a new shuffle module
2. Propose: To build a new randomPosition component inside shuffle module by reusing an existing component.
- 3 Program environment:
 - 3.1 Language : C
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: GCC
 - 3.4 Editor: pico
- 4 Characteristic of module:
 - 4.1 Function: To random value in order to assign a new card
 - 4.2 No. of component inside: 2 components
- 5 Candidate of existing reuse component: randValue component from Exp. 2
- 6 Description:

Although the candidate component (randValue function) can support 2 digits, the outcome generated exceeds some possible values obtain from regular deck. A new designed component (randomPosition function) requires 2 digits whose falls between 1-51 (52 faces possibility in Card Game) in random Card Game. A program structure chart is depicted in Figure C-5 (See Appendix C).
7. Solution:
 - 7.1 Changing name of existing component (randValue function) to a proper new alias component (randomPosition function) that corresponds to a requirement specification.
 - 7.2 Modifying a little bit inside new component (randomPosition function) by allowing random only 2 digits and return value mustn't over 51.

The characteristics of developed reuse component are represented in Table 3-5.

Table 3-5: Representing characteristics of new reuse component in Shuffle Module.

Component Name	Structure	Cyclomatic Complexity
RandomPosition	Function	0

8. Conclusion:

The candidate component can be reused with minor change (Strongly reusable) in RandomPosition component to build a new Shuffle module. So, a Shuffle module is a new effective module for reuse archive proposed.



Experiment #6: Bucket Sort

1. Objective: To develop a new BucketSort module
2. Propose: To reuse an existing component (numberOfDigit) inside a new BucketSort module.
3. Program environment:
 - 3.1 Language : C
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: GCC
 - 3.4 Editor: Notepad
4. Characteristic of module:
 - 4.1 Function: To sort a title of book in form of bucket sort.
 - 4.2 No. of component inside: 4 components
5. Candidate of existing reuse component: numberOfDigits function
6. Description:

There are four components inside a bucket sort module such as numberOfDigits, distributeElement, collectElement, and zeroBucket. Three components must be written from scratch, except numberOfDigit component. An existing component (numberOfDigits function) supports only small numbers, unfortunately. The requirements specification of new component support large numbers. A program structure chart is depicted in Figure C-6 (See Appendix C).
7. Solution:
 - 7.1 Modify an integer variable to be an array of integers for increasing the number of digits in large number.
 - 7.2 Insert a parameter to support array type.

The characteristics of developed reuse component represented in Table 3-6

Table 3-6: Representing characteristics of new reuse component in BucketSort Module.

Component Name	Structure	Cyclomatic Complexity
NumberOfDigit	Function	2

8. Conclusion:

The candidate component can be reused with minor change (Strongly reusable) in numberOfDigit component to build a new BuckSort module. So, a Bucket module is a new effective module for reuse archive proposed.



Experiment #7: MazeTraversal

1. Objective: To develop a new MazeTraversal module
2. Propose: To build a new printMaze component inside MazeTraversal module by reusing existing component.
3. Program environment:
 - 3.1 Language : C
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: GCC
 - 3.4 Editor: pico
4. Characteristic of module:
 - 4.1 Function: To find route in maze in 2 dimensions array.
 - 4.2 No. of component inside: 3 components
5. Candidate of existing reuse component: printMatrix function
6. Description:

A printMaze, a cooraAreEdge, and a validMaze are three key functions in Maze Traversal module. Our design can only reuse the printMaze function. The others components must be written from scratch. A program structure chart is depicted in Figure C-7 (See Appendix C).
7. Procedure:
 - 7.1 Adapt name of a candidate component (printMatrix) to a new component printMaze and change data type in parameter from integer to character.
 - 7.2 Modify a little bit inside component by changing a variable declaration from integer (%d) to character (%c).

The characteristics of developed reuse component represented in Table 3-7.

Table 3-7: Representing characteristics of new reuse component in Maze Traversal Module.

Component Name	Structure	Cyclomatic Complexity
PrintMaze	Function	3

8. Conclusion:

The candidate component can be reused with minor change (Strongly reusable) in printMaze component to build a new Maze Traversal module. So, a Maze Traversal module is a new effective module for reuse archive proposed.



Experiment #8: Merge Score

1. Objective: To develop a new DataMerge module
2. Propose: To build a new MergeSort component inside DataMerge module by reusing an existing component.
3. Program environment:
 - 3.1 Language : Pascal
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: IDE (Turbo Pascal)
4. Characteristic of module:
 - 4.1 Function: To merge the score of students.
 - 4.2 No. of component inside: 1 component
5. Candidate of existing reuse component: MergeSort procedure
6. Description:

A candidate component (MergeSort) supports only string type that merges two-array inputs. A new designed module (DataMerge) requires a MergeSort component to merge two-array input of student's score. A program structure chart is depicted in Figure C-8 (See Appendix C).
7. Procedure
 - 7.1 Changing the data type of MergeSort from string to integer type that corresponds to a specific requirement.
 - 7.2 Adding some code to call MergeSort directly.

St. Gabriel's Library

The characteristics of developed reuse component represented in Table 3-8.

Table 3-8: Representing characteristics of new reuse component in DataMerge Module.

Component Name	Structure	Cyclomatic Complexity
MergeSort	Procedure	4

8. Conclusion:

The candidate component can be reused with minor change (Strongly reusable) in MergeSort component to build a new DataMerge module. So, a DataMerge module is a new effective module for reuse archive proposed.



Experiment #9: 3DMazeRunning

1. Objective: To develop a new SpanWaveEven module.
2. Propose: To build new OutOfDepthRange component inside SpanWaveEven component by reuse an existing component.
3. Program environment:
 - 3.1 Language : C
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: IDE (Turbo C)
4. Characteristic of module:
 - 4.1 Function: To find a way in maze of 3Dimension
 - 4.2 No. of component inside: 3 components
5. Candidate of existing reuse component: outOfRowRange (component inside Maze2Dimension module)
6. Description:

To develop SpanWaveEven module which requires three components, namely outOfRowRange, outOfColumnRange, and outOfDepthRange. Two candidate components (outOfRowRange and outOfColumnRange function inside Maze2Dimension module) can be immediately reused in SpanWaveEven module except outOfDepthRange. The last component must be written from scratch. A program structure chart is depicted in Figure C-9 (See Appendix C).
7. Procedure:
 - 7.1 Adapt the name of candidate component from outOfRowRange to outOfDepthRange that corresponds to the requirement specification.
 - 7.2 Add some code to call outOfDepthRang directly.

The characteristics of developed reuse component represented in Table 3-9.

Table 3-9: Representing characteristics of new reuse component in SpanWaveEven Module.

Component Name	Structure	Cyclomatic Complexity
OutOfRowRange	Function	0
OutOfRowRange	Function	0
OutOfDepthRange	Function	0

8. Conclusion:

Two candidate components can be reused without change (Immediately reusable) and one candidate component can be reused with minor change in outOfDepthRange (Strongly reusable) component to build a new SpanWaveEven module. So, a SpanWaveEven module is a new effective module for reuse archive proposed.

Experiment #10: SML_Compiler

1. Objective: To develop two modules (Open_Source and Print_Message)
2. Propose: To build two components (OpenSource and PrintMessage) inside two modules by reusing existing components
3. Program environment:
 - 3.1 Language : C
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: GCC
 - 3.4 Editor: pico
4. Characteristic of two applied modules:
 - 4.1 Function: To open file for read only (Open_Source module) and display log_file to show on screen (Print_Message module)
 - 4.2 No. of component inside each module: 1 component
5. Candidate of existing reuse component: a openFile function for Open_Source module and a displayScreen function for Print_Message module
6. Description:

The candidate reuse component (openFile function) opens an output file for writing. This violates the solid specification. A new designed component requires opening the output as read-only. As for output, the displayScreen is an immediately reusable for Print_Message module. A program structure chart is depicted in Figure C-10 (See Appendix C).

7. Procedure:

- 7.1 Adapt the name of two candidate components from openFile and displayScreen to openSource and printMessage, respectively.
- 7.2 Modify mode of opening file inside openSource component for read-only mode that corresponds to the functional requirement.

The characteristics of developed reuse component represented in Table 3-10

Table 3-10: Representing characteristics of new reuse component in both Open_Source and Print_Message modules.

Component Name	Structure	Cyclomatic Complexity
PrintMessage	Function	1
OpenSource	Function	3

8. Conclusion:

Two candidate component can be reused with minor change (Strongly reusable) in printMessage and openSoure component to build a new Open_Source module. So, a Open_Source module is a new effective module for reuse archive proposed.



Experiment #11: Similarity Search

1. Objective: To develop of both Similarity Search module and Display module
2. Propose: To build two components of both similarity module and displayData module by reusing existing reuse components
3. Program environment:
 - 3.1 Language : C
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: IDE (Turbo C)
4. Characteristic of two modules:
 - 4.1 Function: To search the term of similar work (Similarity Search module) and display data on screen (Display module).
 - 4.2 No. of component inside each module: 1 component
5. Candidate of existing reuse component: 1) sequentialSearch component for Similarity module and 2) showOnScreen component for Display module
6. Description:

The similarity module requires sequential search while the showOnScreen supports one dimension array to display sorted data on screen. A new designed Display module combines all functional specifications into 2-dimension array. A program structure chart is depicted in Figure C-11 (See Appendix C).
7. Solution:
 - 7.1 Adapting the name of candidate component (showOnScreen function) to a new reuse component (displayData) corresponding to a specific requirement.
 - 7.2 Modifying inside component (displayData) to support 2-dimension array.
 - 7.3 Adding some code in Similarity module to call sequentialSearch directly.

The characteristics of developed reuse component represented in Table 3-11.

Table 3-11: Representing characteristics of new reuse components in both Similarity Search and Display Module.

Component Name	Structure	Cyclomatic Complexity
Similarity	Function	3
DisplayData	Function	2

8. Conclusion:

First, a candidate component (sequentialSearch) can be reused without change (Immediately reusable) in similarity component to build a new Similarity Search module. Second, a candidate component (showOnscreen) can be reused with minor change (Strongly reusable) in displayData component to build a new Display Module. So, both a Similarity Search module and a Display module are two new effective modules for reuse archive proposed.

Experiment #12: Electronic Phone System

1. Objective: To develop CheckFormat module
2. Propose: To build a new dataFormat component inside CheckFormat by reusing existing component.
3. Program environment:
 - 3.1 Language : Pascal
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: IDE (Turbo Pascal)
4. Characteristic of module:
 - 4.1 Function: To check any input error from user insert and update information
 - 4.2 No. of component inside: 1 component
5. Candidate of existing reuse component: nameFormat procedure
6. Description:

The candidate component (nameFormat procedure) checks the valid type of name and shows the error type of user input. A new reuse component (dataFormat function) will check valid type of any information and keep the error type from user input such as name, surname, telephone number, and Address. A program structure chart is depicted in Figure C-11 (See Appendix C).
7. Procedure:
 - 7.1 Adapting the name of candidate component (nameFormat) to a proper name requirement (dataFormat).
 - 7.2 Modifying inside code from procedure to function that corresponds to a specification requirement.
 - 7.3 Promoting redundancy code to be ErrorLog procedure (new component).

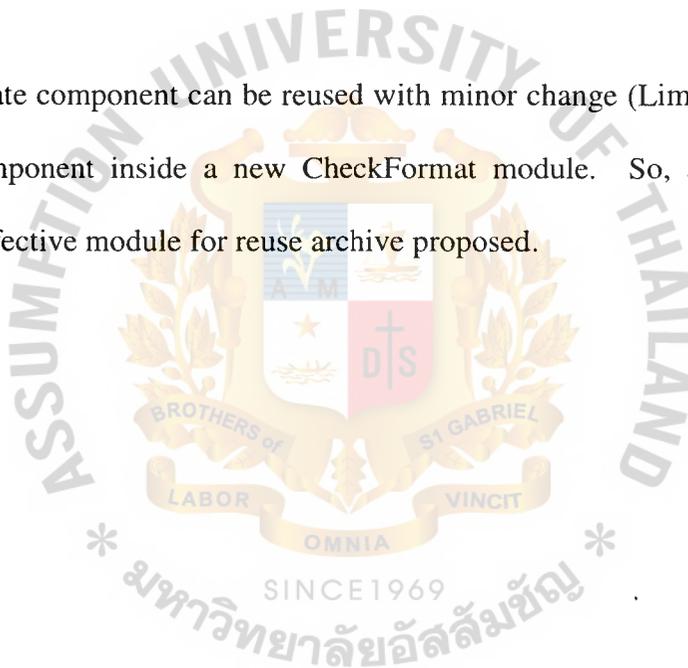
The characteristics of developed reuse component represented in Table 3-12.

Table 3-12: Representing characteristics of new reuse component in CheckFormat Module.

Component Name	Structure	Cyclomatic Complexity
DataFormat	Function	4
ErrorLog	Procedure	1

8. Conclusion:

The candidate component can be reused with minor change (Limitedly reusable) as dataFormat component inside a new CheckFormat module. So, a CheckFormat module is a new effective module for reuse archive proposed.



St. Gabriel's Library

Experiment #13: GGS

1. Objective: To develop a new Distance_Coordinate object
2. Propose: To build distanceToCoordinate method inside Distance_Coordinate module by reusing outside object.
3. Program environment:
 - 3.1 Language : Delphi
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: IDE (Delphi 4.0)
4. Characteristic of module:
 - 4.1 Function: To compute latitude and longitude to mapping on screen
 - 4.2 No. of component inside: 1 component
5. Candidate of existing reuse component: LaLongitude function
6. Description:

A distanceToCoordinate component requires computing the position area by means of latitude and longitude that correspond directly to a candidate component. A program structure chart is depicted in Figure C-13 (See Appendix C).
7. Procedure:
 - 7.1 Changing LaLongitude function to be a virtual function (as Protected), thus permitting distanceToCoordinate to call LaLongitude function directly.
 - 7.2 Adding some code inside distanceToCoordinate component to call LaLongitude component directly.

The characteristics of developed reuse function represented in Table 3-13.

Table 3-13: Representing characteristics of new reuse component in Distance_Coordinate module.

Component Name	Structure	Cyclomatic Complexity
DistanceToCoordinate	Procedure	0

8. Conclusion:

A candidate component can be reused without change (Immediately reusable) in distanceToCoordinate component to build a new Distance_Coordinate module. So, a Distance_Coordinate module is a new effective module for reuse archive proposed.



Experiment #14: Simple Inventory Program

1. Objective: To develop a new BinaryTreeSearch module
2. Propose: To reuse outside module for building a new BinaryTreeSearch module.
3. Program environment:
 - 3.1 Language : Delphi
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: IDE (Delphi 4.0)
4. Characteristic of module:
 - 4.1 Function: To search inventory information
 - 4.2 No. of component inside: 3 components
5. Candidate of existing reuse component: insertTree component and deleteTree component in BinaryTree module.
6. Description:

BinaryTree module provides insertTree component and deleteTree component that corresponds to the specific requirements of BinaryTreeSearch module. The remaining components (binTreeSearch procedure) must be written from scratch. A program structure chart is depicted in Figure C-14 (See Appendix C).
7. Procedure:
 - 7.1 Replacing BinaryTree module to a new BinaryTreeSearch module
 - 7.2 Writing a binTreeSearch component following the proposed guidelines such as high cohesion and loosely coupled principles.

The characteristics of developed reuse method represented in Table 3-14.

Table 3-14: Representing characteristic of new reuse component in BinaryTreeSearch module.

Component Name	Structure	Cyclomatic Complexity
InsertTree	Procedure	Unknown
DeleteTree	Procedure	Unknown
BinTreeSearch	Procedure	3

8. Conclusion:

Two candidate component can be reused without change (Immediately reusable) to build a new BinaryTreeSearch module. So, a BinaryTreeSearch module is a new effective module for reuse archive proposed.



Experiment #15: Fire Alarm Sale System

1. Objective: To develop a new NewBinSearch module.
2. Propose: To build a new component inside NewBinSearch module by reusing existing module
3. Program environment:
 - 3.1 Language : Delphi
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: IDE (Delphi 4.0)
4. Characteristic of module:
 - 4.1 Function: To add getSum (method) for counting the number of user while insert information on file.
 - 4.2 No. of component inside: 2 components
5. Candidate of existing reuse module: BinarySearch module
6. Description:

NewBinSearch is created for adding a getSum component in order to count the number of user while adding information. In so doing, the getSum component invokes an insertData component which adds new information to BinarySearch module. A program structure chart is depicted in Figure C-15 (See Appendix C).
7. Procedure:
 - 7.1 Replace insertData component from BinarySearch module and add virtual (allowing for reuse) to NewBinSearch module.
 - 7.2 Write getSum from scratch.

The characteristics of developed reuse component represented in Table 3-15.

Table 3-15: Representing characteristics of new reuse component in NewBinSearch module.

Method Name	Structure	Cyclomatic Complexity
InsertData	Procedure	1
GetSum	Procedure	0

8. Conclusion:

A candidate component (insertData) can be reused without change (Immediately reusable) to build a NewBinSearch module. So, a NewBinSearch module is a new effective module for reuse archive proposed.



Experiment #16: SafetyNet

1. Objective: To develop a new GenRandomDigit module
2. Propose: To build three components inside GenRandomDigit module by reuse outside module.
3. Program environment:
 - 3.1 Language : Delphi
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: IDE (Delphi 4.0)
4. Characteristic of module:
 - 4.1 Function: To produce the functions of random by returning digit value only
 - 4.2 No. of component inside: 3 components
5. Candidate of existing reuse component: genRandom component in GenPassword module
6. Description:

A GenPassword module provides two components namely, genRandom and checkPassword to complete reading user's password. The genRandom component subsequently invokes GenRandomDigit module to generate random value. A program structure chart is depicted in Figure C-16 (See Appendix C).
7. Procedure:
 - 7.1 Replacing a GenPassword module to a GenRandomDigit module.
 - 7.2 Adapting a genRandom function by adding virtual (Allowing for reuse) according to the proposed guidelines such as high cohesion, loosely coupled principles.

The characteristics of developed reuse method represented in Table 3-16.

Table 3-16: Representing characteristics of new reuse component in GenRandomDigit module.

Component Name	Structure	Cyclomatic Complexity
GenRandom	Procedure	1

8. Conclusion:

A candidate component can be reused with minor change (Strongly reusable) to build a new GenRandomDigit module. So, a GenRandomDigit module is a new effective module for reuse archive proposed.



Experiment #17: Binary Tree

1. Objective: To develop a new NewBinTree object
2. Propose: To build three methods inside NewBinTree object by reuse outside object.

3. Program environment:

3.1 Language : Java

3.2 Programmer level: Sophisticated

3.3 Compiler: JDK 1.2.1

3.4 Editor: Kawa

4. Characteristic of Object:

4.1 Function: To insert, delete and search in form of binary tree

4.2 No. of method inside: 3 methods

5. Candidate of existing reuse method: insertTree and deleteTree methods in BinaryTree object.

6. Description:

A NewBinTree object requires three methods, namely, insertTree method, deleteTree method, and searchTree method. Two methods can be reused from BinaryTree object. The searchTree method requires NewBinTree module to build a binary tree. A program structure chart is depicted in Figure C-17 (See Appendix C).

7. Procedure:

7.1 Inherit BinaryTree object from NewBinTree object.

7.2 Build searchTree method is according to the proposal guidelines such as high cohesion, loosely coupled principles.

7.3 Set all methods to be protected mode

The characteristics of developed reuse method represented in Table 3-17.

Table 3-17: Representing characteristic of new reuse method in NewBinTree object.

Component Name	Structure	Cyclomatic Complexity
insertTree	Method	Unknown
deleteTree	Method	Unknown
searchTree	Method	3

8. Conclusion:

Two candidate methods can be reused without change (Immediately reusable) to build a new NewBinTree object. So, a NewBinTree object is a new effective object for reuse archive proposed.



Experiment #18: ReserveTicket

1. Objective: To develop a new MovieTicketBooth object
2. Propose: To build five methods inside MovieTicketBooth object by reusing outside object.
3. Program environment:
 - 3.1 Language : Java
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: JDK 1.2.1
 - 3.4 Editor: Kawa
4. Characteristic of Object:
 - 4.1 Function: To reserve and cancel the ticket
 - 4.2 No. of method inside: 3 methods
5. Candidate of existing reuse method: positionAvailable methods in TicketBooth object.
6. Description:

A new MovieTicketBooth object requires three methods, namely positionAvaliable, reserve, and cancel. So one specific requirement of new object requires a positionAvailable method embedded within embed TicketBooth object to locate the desired position. A program structure chart is depicted in Figure C-18 (See Appendix C).
7. Solution:
 - 7.1 Inherit Movie object to Ticket object for reusing a positionAvaliable method.
 - 7.2 Build both reserveTicket method and cancelTicket method by writing from scratch according to the proposed guidelines.
 - 7.3 Set all methods of new object to protected mode.

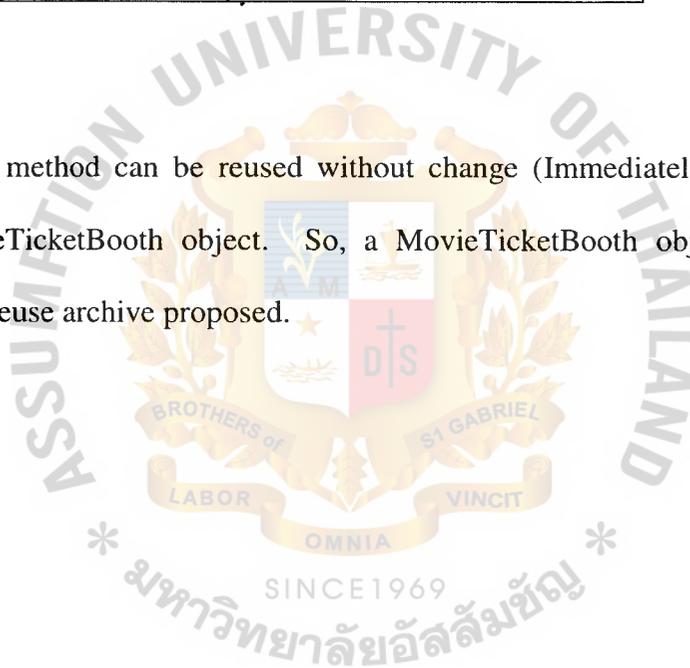
The characteristics of developed reuse method represented in Table 3-18.

Table 3-18: Representing characteristic of new reuse methods in MovieTicketBooth object.

Component Name	Structure	Cyclomatic Complexity
positionAvailable	Method	Unknown
reserveTicket	Method	1
cancelTicket	Method	1

8. Conclusion:

A candidate method can be reused without change (Immediately reusable) to build a new MovieTicketBooth object. So, a MovieTicketBooth object is a new effective object for reuse archive proposed.



Experiment #19: Softdrink_Vending_Machine

1. Objective: To develop a new SoftdrinkVendingMachine object
2. Propose: To build three methods inside SoftdrinkVendingMachine object by reuse outside object.
3. Program environment:
 - 3.1 Language : Java
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: JDK 1.2.1
 - 3.4 Editor: Kawa
4. Characteristic of Object:
 - 4.1 Function: To produce the functions of soft drink vending machine
 - 4.2 No. of method inside: 5 methods
5. Candidate of existing reuse method: Three methods in VendingMachine object
6. Description:

A VendingMachine object provides three methods, namely, productAvailable, priceOfProduct, and exchangeCalculate. These methods in turn are required in new SoftdrinkVendingMachine object. A program structure chart is depicted in Figure C-19 (See Appendix C).
7. Solution:
 - 7.1 Inherit VendingMachine object to SoftdrinkVendingMachine object.
 - 7.2 Build getTemperature method and controlTemperature method from scratch following the proposed guidelines
 - 7.3 Add protected mode to all new methods.

The characteristics of developed reuse method represented in Table 3-19.

Table 3-19: Representing characteristic of new reuse methods in SoftdrinkVendingMachine object.

Component Name	Structure	Cyclomatic Complexity
productAvailable	Method	Unknown
priceOfProduct	Method	Unknown
exchangeCalculate	Method	Unknown
getTemperature	Method	0
controlTemperature	Method	1

8. Conclusion:

Three candidate methods can be reused without change (Immediately reusable) to build a new SoftdrinkVendingMachine object. So, a SoftdrinkVendingMachine object is a new effective object for reuse archive proposed.

Experiment #20: Ticket_Vending_Machine

1. Objective: To develop a new TicketVendingMachine object
2. Propose: To build three methods inside TicketVendingMachine object by reuse outside object.
3. Program environment:
 - 3.1 Language : Java
 - 3.2 Programmer level: Sophisticated
 - 3.3 Compiler: JDK 1.2.1
 - 3.4 Editor: Kawa
4. Characteristic of Object:
 - 4.1 Function: To produce the functions of ticket vending machine
 - 4.2 No. of method inside: 5 methods
5. Candidate of existing reuse method: Three methods in VendingMachine object from Exp. 19.
6. Description:

A VendingMachine object provides three methods, namely, productAvailable, priceOfProduct, and exchangeCalculate. These methods in turn are required in new TicketVendingMachine object. Moreover, TicketVendingMachine object also requires ticketClassification method and ticketTimeStamp method by writing from scratch. A program structure chart is depicted in Figure C-20 (See Appendix C).
7. Solution:
 - 7.1 Inherit VendingMachine object to TicketVendingMachine object and build two new methods from scratch following the proposed guidelines
 - 7.2 Add protected mode to all new methods.

The characteristics of developed reuse method represented in Table 3-20.

Table 3-20: Representing characteristics of new reuse methods in TicketVendingMachine object.

Component Name	Structure	Cyclomatic Complexity
PriceOfProduct	Method	Unknown
ProductAvailable	Method	Unknown
ExchangeCalculate	Method	Unknown
TicketClassification	Method	9
TicketTimeStamp	Method	0

8. Conclusion:

Three candidate methods can be reused without change (Immediately reusable) to build a new TicketVendingMachine object. So, a TicketVendingMachine object is a new effective object for reuse archive proposed.

3.4 EXPERIMENT RESULTS AND ANALYSIS

Traditional software metrics were applied to all experiments, namely, time spent, number of errors, and lines of code (LOC), to measure productivity and quality. The results are shown in Table 3-21.

Table 3-21: Results comparing code reuse and conventional code writing

No. of Experiment	Developing with reuse approach			Developing with conventional approach		
	LOC	No. of error	Time to finish (person-hour)	LOC	No. of error	Time to finish (person-hour)
Functional Programming Language						
Exp. I	48	0	½	40	0	1
Exp. II	88	0	½	80	1	¾
Exp. III	95	0	2	88	2	5
Exp. IV	105	2	1 ½	105	2	1
Exp. V	145	1	½	135	1	1 ½
Exp. VI	157	2	2	163	5	2
Exp. VII	175	2	1 ½	190	4	2
Exp. VIII	230	1	4 ½	247	3	12
Exp. IX	363	2	5	421	6	8
Exp. X	610	2	10	690	7	15
Exp. XI	660	3	19	726	8	25
Exp. XII	752	2	7 ½	848	11	14
Object-Base Programming Language						
Exp. XIII	308	2	11	318	5	14
Exp. XIV	460	2	5 ½	480	7	9
Exp. XV	688	3	15	804	8	20
Exp. XVI	873	4	17	893	7	21
Object-Oriented Programming Language						
Exp. XVII	322	1	5	316	2	7
Exp. XVIII	486	1	4	488	2	6
Exp. XIX	396	3	12	402	5	15
Exp. XX	483	1	15	498	4	17

As depicted in Table 3-21, results comparison of code reuse and conventional code writing are as follows:

- 1) LOC of the complete programs developed with reuse approach is lower than developed with conventional approach except Exp. I, II, III, and V.
- 2) The number of errors of the complete program developed with reuse approach is less when compared with conventional approach.
- 3) Time of the complete program developed with reuse approach is lower than developed with conventional approach except Exp. IV and VI.

Results of the experiment raise two interesting:

- 1) Why is LOC of developed with reuse approach in Exp. I, II, III, and V higher than developed with conventional approach?
- 2) Why is time spent on complete program in developed with reuse approach in both Exp. IV and VI is not less than that of developed with conventional approach?

A simple and straightforward explanation to the first question is that the developer has to concentrate on how to build program for reuse. Thus, functions/module in a small program may be unnecessary because of the sheer size of the entire program. Moreover, conventional approach does not take into account promoting future code reuse.

For example, a loop to write string may take only 2 or 3 lines of code in GetSentence procedure of Read_Sentence program. Such a specific implementation does not support modularity. In order to make this code segment reusable, it is necessary to generalize its functionality, which eventually may require 8 lines of code as shown in Figure 3-8.

```
Procedure GetSentence (var St : SText;  
                      var Last : integer);  
Begin
```

```
(1) write('Output : ');  
    for i := 1 to Last do  
        write(St[i]);
```

(1) Looping string output, it's required 3 lines of code.

```
(2) Output(St, Last);
```

```
End;
```

```
Procedure Output (var St : SText;  
                 Var Last : integer);
```

```
Begin  
    write('Output : ');  
    for i := 1 to Last do  
        write(St[i]);  
    readln;  
End;
```

(2) To promote output procedure for support reusable component, it's required 8 lines of code.

Figure 3-8: Comparison of both (1) traditional approach and (2) reuse approach

Although, the similar programmer code in each experiment but LOC in comment for reuse approach is almost equal or greater than LOC in comment of conventional approach because reuse approach requires a good comment for other reader. So, it must be clean.

The main issue concerning the second question is that both programs contain complexity algorithm. As such, it calls for considerable effort to understand the candidate component. In this case, developer should decide to build the reusable component from scratch or reuse from existing component.

Further analysis on LOC and complexity measures [6] revealed similar trend for software productivity and quality as the module size increased. This is shown in Table 3-22 and Table 3-23.

Table 3-22: Comparison of productivity and quality using LOC measurement

No. of Experiment	Productivity			Quality		
	Reuse	Non-reuse	Difference	Reuse	Non-reuse	Difference
Functional Programming Language (C or Pascal)						
Exp. I	0.096	0.040	0.056	0.000	0.000	0.000
Exp. II	0.176	0.107	0.069	0.000	12.500	12.500
Exp. III	0.048	0.018	0.030	0.000	22.727	22.727
Exp. IV	0.070	0.105	-0.035	19.048	19.048	0.000
Exp. V	0.290	0.090	0.200	6.897	7.407	0.511
Exp. VI	0.079	0.082	-0.003	12.739	30.675	17.936
Exp. VII	0.117	0.095	0.022	11.429	21.053	9.624
Exp. VIII	0.051	0.021	0.031	4.348	12.146	7.798
Exp. IX	0.073	0.053	0.020	5.510	14.252	8.742
Exp. X	0.061	0.046	0.015	3.279	10.145	6.866
Exp. XI	0.073	0.029	0.044	4.545	11.019	6.474
Exp. XII	0.100	0.061	0.040	2.660	12.972	10.312
Object-Based Programming Language (Delphi)						
Exp. XIII	0.028	0.023	0.005	6.494	15.723	9.230
Exp. XIV	0.084	0.053	0.030	4.348	14.583	10.236
Exp. XV	0.046	0.040	0.006	4.360	9.950	5.590
Exp. XVI	0.051	0.043	0.009	4.582	7.839	3.257
Object-Oriented Programming Language (Java)						
Exp. XVII	0.064	0.045	0.019	3.106	6.329	3.224
Exp. XVIII	0.122	0.081	0.040	2.058	4.098	2.041
Exp. XIX	0.033	0.027	0.006	7.576	12.438	4.862
Exp. XX	0.032	0.029	0.003	2.070	8.032	5.962

St. Gabriel's Library

Note that larger numbers in productivity column imply higher programmer productivity whereas larger figures in quality column (state equivalently more errors) imply lower output quality. This is independent of metrics used, i.e., LOC or complexity.

Table 3-23: Comparison of productivity and quality using complexity measurement

No. of Experiment	Productivity			Quality		
	Reuse	Non-reuse	Difference	Reuse	Non-reuse	Difference
Functional Programming Language (C or Pascal)						
Exp. I	10.360	4.830	5.530	0.000	0.000	0.000
Exp. II	10.780	6.720	4.060	0.000	0.198	0.198
Exp. III	19.110	7.154	11.956	0.000	0.056	0.056
Exp. IV	7.187	10.080	-2.893	0.186	0.198	0.013
Exp. V	22.680	7.093	15.587	0.088	0.094	0.006
Exp. VI	14.400	13.500	0.900	0.069	0.185	0.116
Exp. VII	6.320	4.440	1.880	0.211	0.450	0.239
Exp. VIII	9.778	3.438	6.340	0.023	0.073	0.050
Exp. IX	5.712	3.358	2.355	0.070	0.223	0.153
Exp. X	4.240	2.650	1.590	0.047	0.176	0.129
Exp. XI	4.703	1.591	3.112	0.071	0.201	0.130
Exp. XII	11.371	5.720	5.651	0.023	0.137	0.114
Object-Based Programming Language (Delphi)						
Exp. XIII	14.991	11.086	3.905	0.012	0.032	0.020
Exp. XIV	16.298	9.360	6.938	0.022	0.083	0.061
Exp. XV	20.172	14.207	5.966	0.010	0.028	0.018
Exp. XVI	4.384	3.330	1.054	0.054	0.100	0.046
Object-Oriented Programming Language (Java)						
Exp. XVII	22.410	15.043	7.367	0.009	0.019	0.010
Exp. XVIII	4.438	2.750	1.688	0.056	0.121	0.065
Exp. XIX	4.703	3.536	1.167	0.053	0.094	0.041
Exp. XX	5.755	4.772	0.983	0.012	0.049	0.038

Sample calculations of productivity and quality are given in Appendix B. The percentage improvement of productivity and quality, using LOC measurement and complexity measurement are 55.90% and 58.47%, and 63.22% and 59.67% respectively. Therefore, it is apparent from the experiment that software reuse contributes greatly to productivity and quality as the size of the software becomes larger. The framework established earlier plays an important role in the development improvement. The experiment results confirm:

- 1) LOC reduction implying less effort,
- 2) error reduction implying increase in software reliability, and

3) time spent reduction implying earlier finishing time of the product.

Analysis of results is divided into *theoretical aspect* and *implementing aspect*.

The theoretical aspect encompasses:

- 1) Complexity reduction helps reduce time to understand the reuse component;
- 2) Loose coupling allows reuse component to be inserted or deleted similar to hardware counterpart's plug and play capability;
- 3) High cohesion helps predict some properties of product implementation such as ease of debugging, ease of maintenance, and ease of modification [12].

The Implementing aspect involves:

- 1) Higher efficiency of project management enables the developer to control the project's completion within the allotted time and costs.
- 2) Better software quality can be attained from the above built-in theoretical aspect of the reuse component.

Further analysis on LOC and complexity measures [6] revealed similar trend for software productivity and quality by means of various programming languages those are shown in Table 3-24 and Table 3-25.

Table 3-24: Comparison of programming languages based on productivity

Programming Language	Average productivity using LOC			Average productivity using complexity		
	Reuse	Non-reuse	Improvement	Reuse	Non-reuse	Improvement
Functional	0.103	0.062	65.59%	10.553	5.881	79.44%
Object-based	0.052	0.040	31.55%	13.961	9.495	47.03%
Object-oriented	0.063	0.046	37.54%	9.326	6.525	42.93%

Table 3-25: Comparison of programming languages based on quality

Programming Language	Average quality using LOC			Average quality using complexity		
	Reuse	Non-reuse	Improvement	Reuse	Non-reuse	Improvement
Functional	5.871	14.495	59.50%	0.066	0.166	60.44%
Object-based	4.946	12.024	58.87%	0.025	0.061	59.75%
Object-oriented	3.702	7.724	52.07%	0.033	0.071	54.19%

Comparative results of Table 3-24 and Table 3-25 are depicted in Figure 3-9 and

Figure 3-10 respectively.

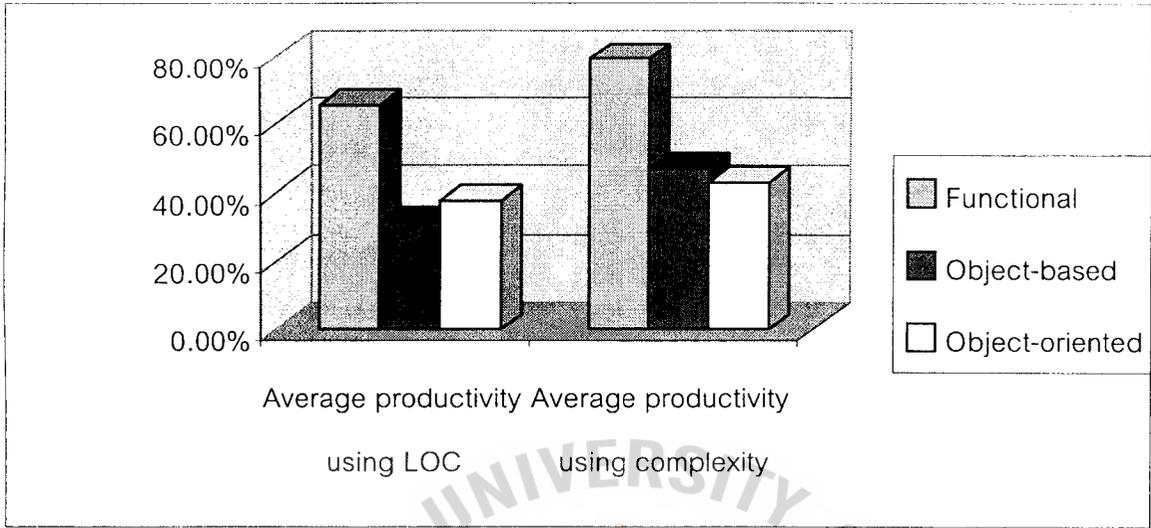


Figure 3-9: Comparison of three style of programming languages based on productivity.

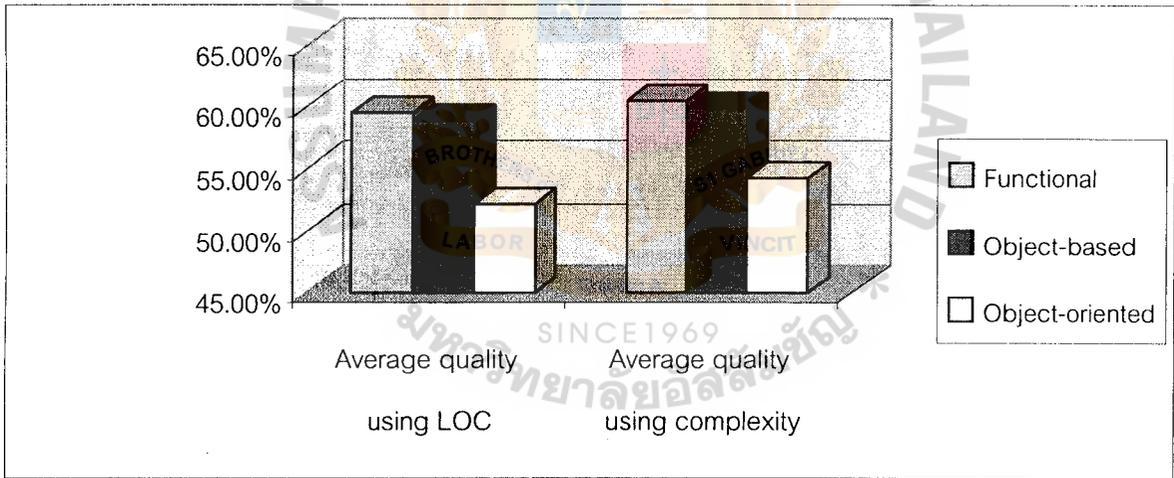


Figure 3-10: Graphical comparison of programming languages based on quality.

As depicted in Figure 3-9, the percentage of average productivity increment between reuse approach and non-reuse approach of functional programming language (C or Pascal) is higher rather than Object-based programming language (Delphi) and Object-oriented programming language (Java). This implies that different program

structures affect code development with reuse because writing code for reuse is not inherent for C or Pascal, whereas both Delphi and Java contain properties reusable code that support modern programming language.

It can be seen from Figure 3-9, the productivity of C or Pascal is much higher than Delphi and Java. Figure 3-10 shows slight percentage increase of quality of procedural language over Object-based language and Object-oriented language. Remember the quality of software depends on the number of defects. Variation in programming languages does not effect software quality. Thus the small-scale experiments confirm higher software productivity as the proposed guidelines were followed.

The feature of developing with reuse approach is to build the application software being avoidable to start at zero. The proposed guidelines will benefits software development with reuse in many regards as follows:

1. *Simplicity in code design plan*

Normally it 's hard to design code in a large application encompassing many functions. All large applications require high expertise. But with code reuse, the designers could arrive at simpler design plan for the desired.

2. *Creative your code reuse (or Developing code reuse in nearly your work)*

Instead of taking only available code from existing libraries or external sources, repository of the domain corresponding to functional requirements will be added up in the archived inventory that are closer to user's need.

3. *Improving portability*

To write code reuse in standard format (such as ADT style), it can help developers to be easily understandable. Such almost users are familiar with Microsoft WINDOWS interface.

4. *Effective use can be made of programmers.*

A visionary software development team can avoid reinventing the wheel on different projects by means of reuse. Hence, programmers will have more time to develop new code segment for future reuse.

5. *System reliability is increased.*

Reuse code segments which have been exercised in working systems are more reliable than newly developed code segments.

6. *Overall process risk is reduced.*

Uncertainty is one of the importance factors that all project managers try to minimize as much as they can. Therefore programmers develop applications using reuse code can reduce the uncertainty considerably.

7. *Software development time can be reduced.*

Shorter development time means increase speed of delivery. Bringing the system to market as early as possible is perhaps more important than focusing on overall development costs.

CHAPTER 4 CONCLUSIONS

This study proposes an approach of code reuse guidelines to aid in software development. The main objective is to prevent developers from reinventing the wheel. The underlying principles of code reuse rely on simplicity of reusable code selection and new code design strategies. A small scale experiment was performed to determine the effective use of code reuse. The results confirmed an overall increase in productivity and quality. As a consequent, software developers should first think reuse as a means for minimizing resources and development efforts.

Some limited factors make guidelines unsuccessful developments are as follows:

- 1) High complexity algorithm of candidate component, developing program based on proposed guidelines is required the candidate component. If it is not easy to understand or hard to modify, it will be required mush efforts rather than writing from scratch.
- 2) Limited Transaction processing program support, this guideline doesn't build to advocate the Artificial Intelligence program (AI).
- 3) GUI-oriented and simplicity code, adapting and/or modifying the procedural detail of component corresponds to functional requirement is required, so some language (such as Power Builder) are GUI-oriented and required a simple code is not suitable for this proposed guidelines.

According to one objective of proposed guidelines is to build or improve the performance and quality of reuse component, so sophisticated end users whose meet their complex requirements are target group. It is important to remember that firstly creating a reusable component not only increase the development cost but also due to the extra effort needed to introduce the characteristic of reusable component.

The first version of code for reuse took much efforts but less in percent of reuse because almost candidate components weren't created with reuse. It is hoped that by following the proposed guidelines, the resulting code will be efficient for reuse and simplistic in subsequent development domain.

This thesis, however, has demonstrated only one aspect of reuse in the area of program development or coding. A full-fledged software development endeavor should incorporate as many reuse aspects as possible to take full advantage of the reusable software technology.

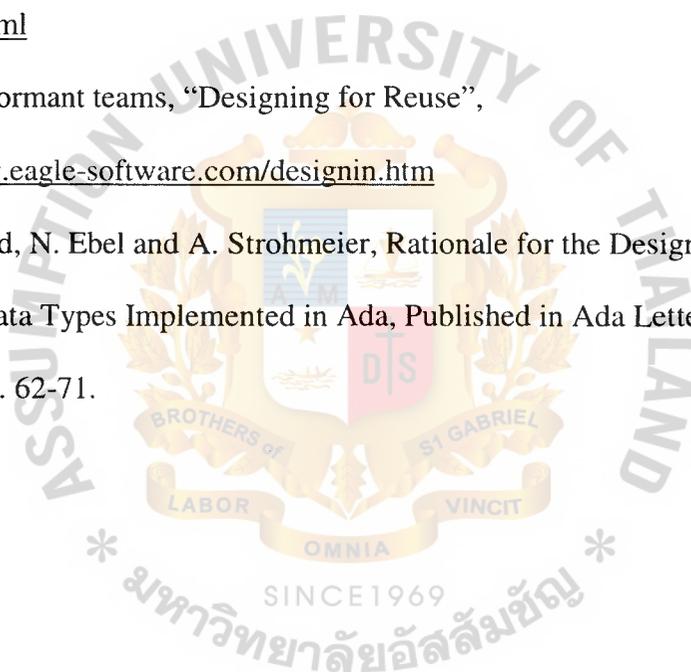
Future work based on reuse modification and adaptation techniques of the proposed guidelines, the criteria on which technique of reuse component should be employed under certain circumstances still remain unexplored. Software development trend will continue to strive for reusable components throughout the development lifecycle. This means that there will be different reusable component repositories developed for high potential reuse inventory. Such archival systems call for an efficient storage and retrieval system to support open access for software community.

St. Gabriel's Library

REFERENCES

- [1] Ian Sommerville. Software Engineering, Fifth Edition, pp. 219, 1995.
- [2] Ted Davis, "Adopting a Policy of Reuse", *IEEE Spectrum* (June 1994) pp. 44-48.
- [3] R. Capilla, "Análisis del dominio: ¿ hacia un modelo de reutilización sistemático?," April 1996.
- [4] Steve McConnell "Why You Should Use Routines...Routinely," *IEEE Software* July/August 1998.
- [5] B. Boehm, "Improving software productivity," *IEEE Software*, pp. 43-57, Sept. 1987
- [6] Roger S. Pressman. Software Engineering, Third edition. McGraw-Hill Companies, Inc., 1992.
- [7] M. Cusumano, "The Software Factory: A Historical Interpretation," *IEEE Software* (March 1989) pp. 23-30.
- [8] R. Prieto-Diaz, "A Domain Analysis Process Model," SPC-92032, *Software Productivity Consortium*, Herndon, VA.
- [9] Carma McClure, "Reuse-based Software Development Methodologies Explained" copy right (C) 1997 by Extended Intelligence, Inc.
- [10] M. Ramchandran and I. Sommerville, "Software Reuse Guidelines"
<http://www.comp.lancs.ac.uk/computing/research/cseg/projects/APPRAISAL/paper/paper.html>
- [11] Rafael Capilla, "Application of Domain Analysis to Knowledge Reuse"
<http://www.umcs.maine.edu/~ftp/wisr/wisr8/papers/capilla/capilla.html>

- [12] E. Yourdon and L. Constantine, Structured Design. Englewood Cliffs, N.J.: Prentice Hall, 1979.
- [13] The International User Group Council, “Reuse in Object-Oriented Development”, <http://www.guide.org/jgs/jgsoo1.htm>
- [14] Carma McClure, “Software Reuse Techniques”, Published by Prentice Hall PTR Prentice-Hall, Inc.
- [15] M. Ramchandran and I. Sommerville, “Software Reuse Guidelines”
<http://www.complan.cs.ac.uk/computing/research/cseg/projects/APPRAISAL/paper/paper.html>
- [16] Delphi informant teams, “Designing for Reuse”,
<http://www.eagle-software.com/designin.htm>
- [17] C. Genillard, N. Ebel and A. Strohmeier, Rationale for the Design of Reusable Abstract Data Types Implemented in Ada, Published in Ada Letters (1989), Vol. 9 No. 2, pp. 62-71.



APPENDIX A

McCabe Complexity Metrics and Halstead Complexity Metrics can be used to check for redundancy.

A1: McCabe Metrics [14]

McCabe Cyclomatic, Essential, and Design Complexity Metrics can be used to detect software redundancies. They be calculated by hand or by automatic complexity metrics tools.

Cyclomatic Complexity is a graph theory complexity measure that is an application of flow graphs to software program logic.

Cyclomatic Complexity = the number of logic paths in a program function

Essential Complexity is a measure of the “structuredness” of a program function that is calculated by counting the number of GO Tos (excluding GO TO EXITS). It is a count of the number of times a control path branches outside of the function and does not return. In a perfectly structured function, its value is 1.

Essential Complexity = Number of returning branches / total number of branches

Design Complexity is the complexity of a “design reduced” program function in which the function’s flow graph is reduced by treating all logic decisions and loops that do not contain calls to immediate subordinate functions as if they were straight lines (that is, one path). Thus, the function’s design complexity is less than or equal to its cyclomatic complexity, and typically is much less.

A2: Halstead Metrics [14]

Halstead's Software Science Complexity Metrics are also used to measure the complexity characteristics of software programs. While McCabe's metrics focus on logic path in a program, Halstead's Metrics are based on counting the number of unique operators and operands in a program.

Operator are reserved programming language words such as ADD, GREATER THAN, MOVE, READ, IF, CALL; arithmetic operators such as +, -, *, /; and logical operators such as GREATER THAN or EQUAL TO.

Operand are data variables and constants in the program.

Some of the basic Halstead Metrics for calculating Length and Volume of program are

$$\text{Length} = N + O$$

$$\text{Volume} = L * \log_2 (n+o)$$

Where n = number of unique operators

o = number of unique operands

O = total number of operators

N = total number of operands

Some programs may be more sensitive to one metric than another, so it is a good idea to calculate both metrics for each program or program function. Also, both metrics can be applied to source code or to design specifications, and both are supported by several automated tools that work with many different programming languages.

APPENDIX B

B1: Calculation the productivity and the quality by measuring Line of Code

Size-oriented software metrics [11] is direct measures of software and the process by which it is developed. Those formulas involve:

$$\text{Productivity} = \text{KLOC/person-hour}$$

$$\text{Quality} = \text{defects/KLOC}$$

A Sample calculation of productivity and quality

Experiment #.XII : Electronic Phone System (EPS)

Program Language : Pascal

Implementation by : Reusable code

Experiment Results : LOC is equal to 752, Defect is equal to 2
and person-hour is equal to 7 ½

$$\text{Productivity} = \text{KLOC/person-hour}$$

$$= (752 / 1000) / 7.5$$

$$= 0.100$$

$$\text{Quality} = \text{defect/KLOC}$$

$$= 2 / (752 / 1000)$$

$$= 2.660$$

B2: Calculation the productivity and the quality by measuring complexity

Function-oriented software metrics [11] are indirect measures of software and the process by which it is developed. Rather than counting LOC, function-oriented metrics focus on program “functionality” or “utility”.

To compute function points, the following relationship is used:

$$FP = \text{count-total} * [0.65 + 0.01 * \text{SUM}(F_i)]$$

Once function points have been calculated, they are used in a manner analogous to LOC as a measure of software productivity and quality:

$$\text{Productivity} = FP / \text{person-hour}$$

$$\text{Quality} = \text{defects} / FP$$

A Sample calculation of productivity and quality

Experiment #.XII : Electronic Phone System (EPS)

Program Language : Pascal

Implementation by : Reusable code

Experiment Results : Defect is equal to 2 and person-hour is equal to 7 ½

Table B-1: Computing function-point metrics

Measurement parameter	Count	Weight factor			Total
		Simple	Average	Complex	
Number of user inputs	12	3	4	*6	72
Number of user outputs	2	4	*5	7	10
Number of user inquires	2	3	*4	6	8
Number of file	1	*7	10	15	7
Number of external interfaces	1	5	*7	10	7
Count-Total					104

Table B-2: Computing the complexity adjustment values

Complexity Adjustment values	Factor						Total
	No-influence (0)	Incidental (1)	Moderate (2)	Average (3)	Significant (4)	Essential (5)	
Is the code designed to be reusable?						Yes	5
Are there distributed processing functions?				Yes			3
Is the internal processing complex?				Yes			3
Are the inputs, outputs, files complex?		Yes					2
Is the application designed to facilitate change and ease of use by the user?					Yes		4
Count-Total							17

Calculation:

Count-total is equal to 104 and $\text{SUM}(F_i)$ is equal to 17. So,

$$\begin{aligned}
 \text{FP} &= \text{count-total} * [0.65 + 0.01 * \text{SUM}(F_i)] \\
 &= 104 * [0.65 + 0.01 * 17] \\
 &= 85.28
 \end{aligned}$$

hence person-hour is equal to $7 \frac{1}{2}$

$$\begin{aligned}
 \text{Productivity} &= \text{FP} / \text{person-hour} \\
 &= 85.28 / 7.5 \\
 &= 11.371
 \end{aligned}$$

$$\begin{aligned}
 \text{Quality} &= \text{defects} / \text{FP} \\
 &= 2 / 85.28 \\
 &= 0.023
 \end{aligned}$$

APPENDIX C

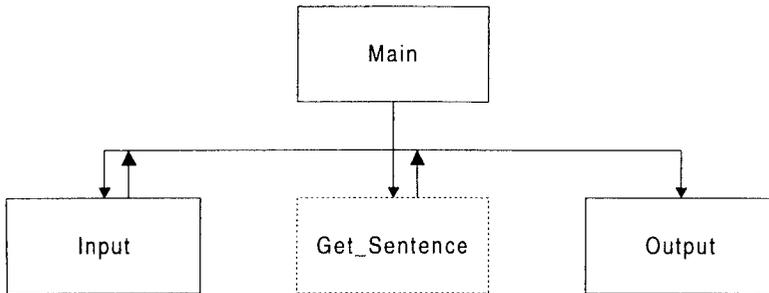


Figure C-1: A program structure of Get_Sentence module in experiment#1

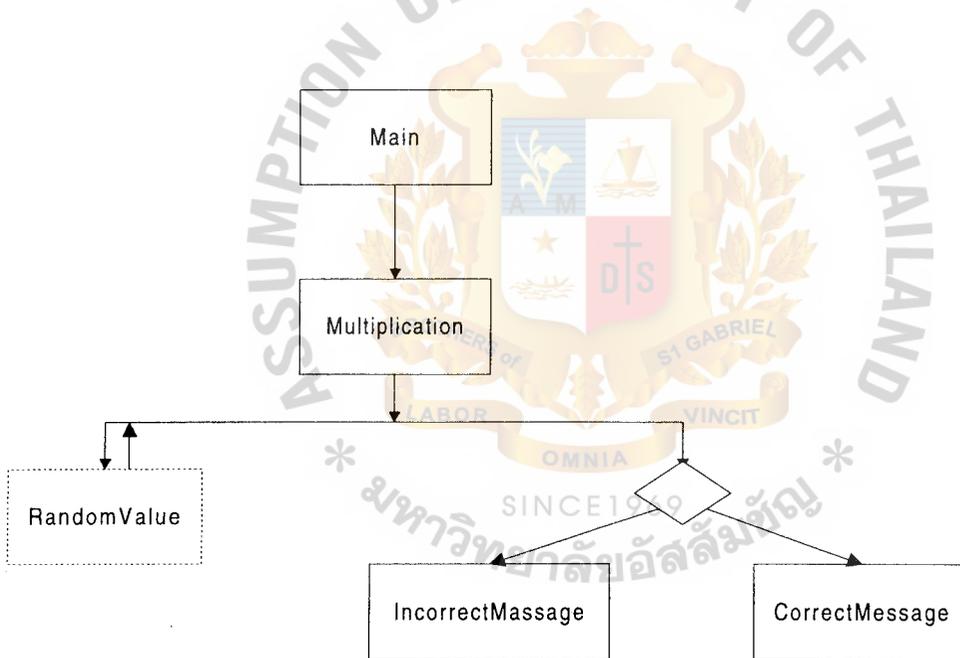


Figure C-2: A program structure of RandomValue module in experiment#2

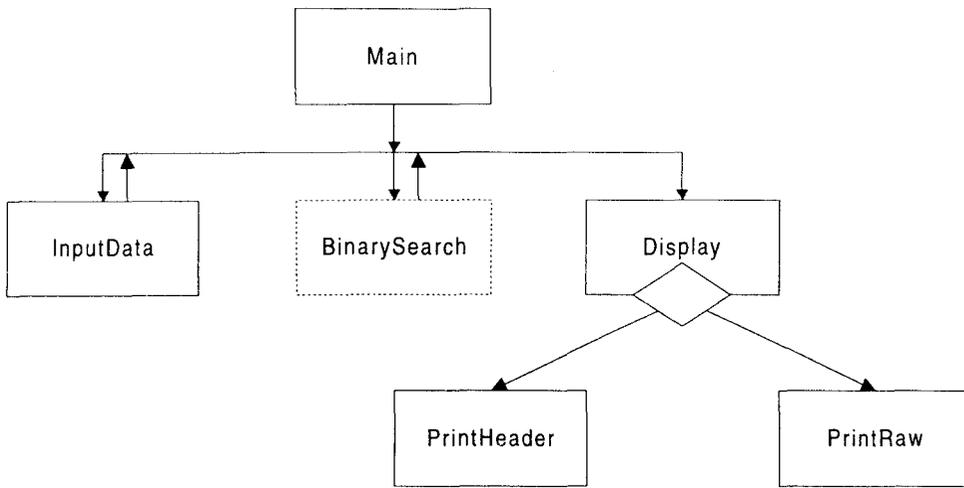


Figure C-3: A program structure of BinarySearch module in experiment#3

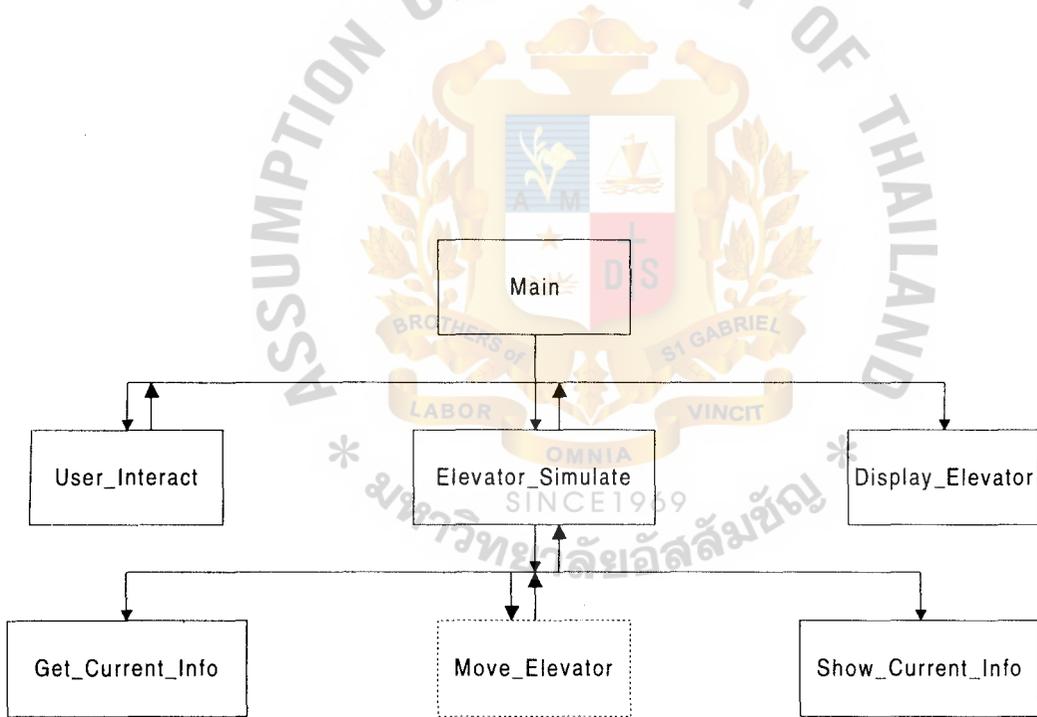


Figure C-4: A program structure of Move_Elevator module in experiment#4

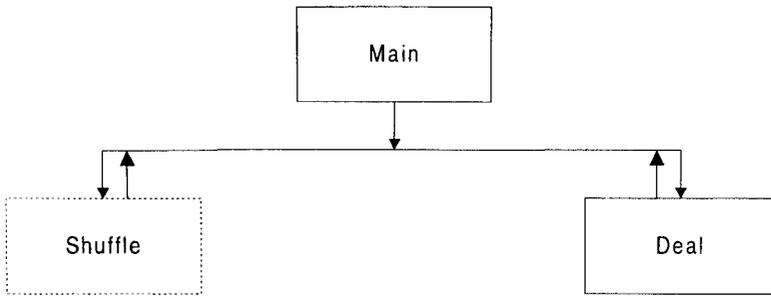


Figure C-5: A program structure of Shuffle module in experiment#5

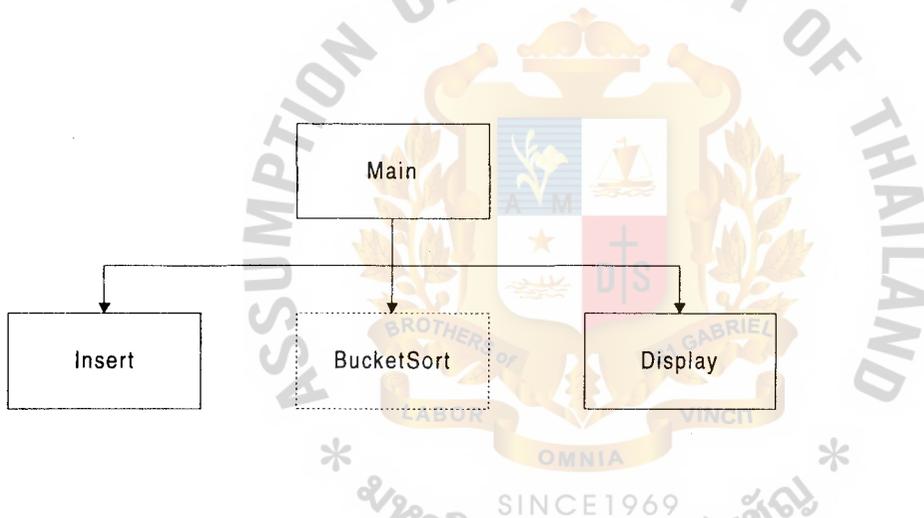


Figure C-6: A program structure of BucketSort module in experiment#6

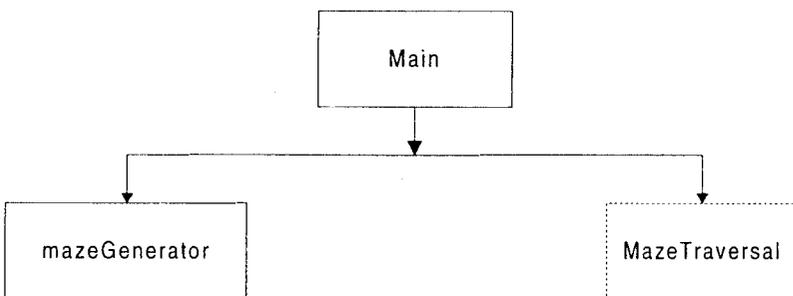


Figure C-7: A program structure of MazeTraval module in experiment#7

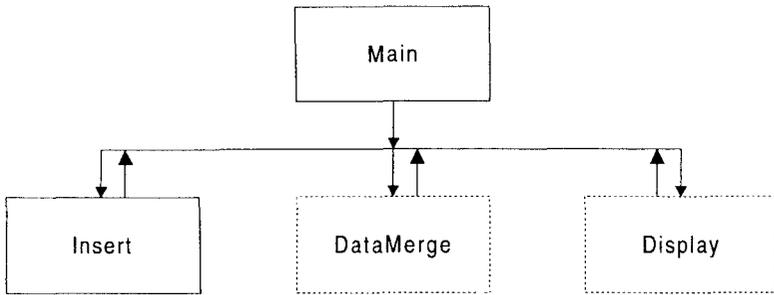


Figure C-8: A program structure of DataMerge module in experiment#8

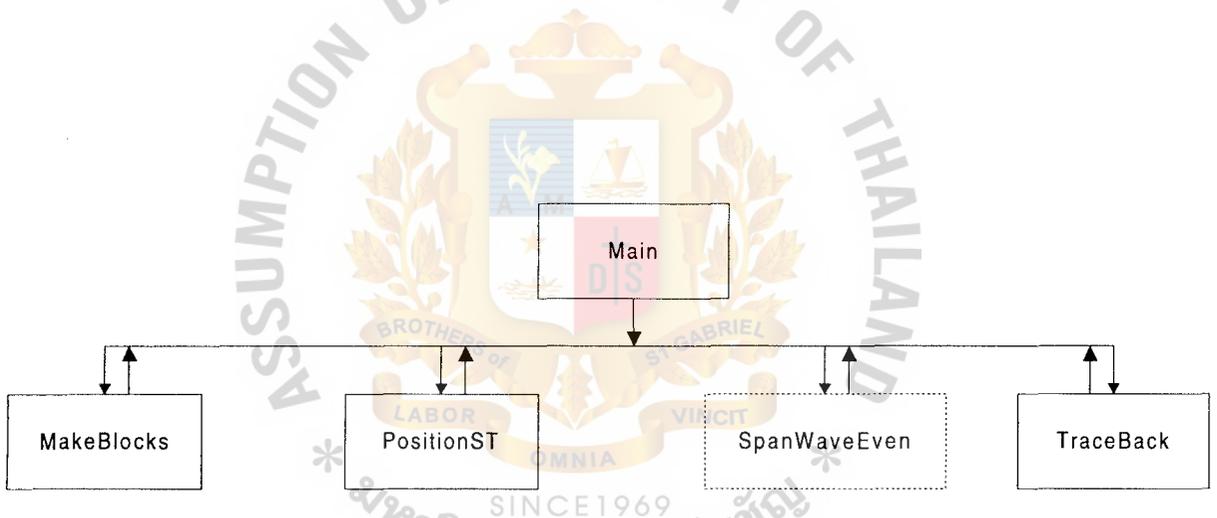


Figure C-9: A program structure of SpanWaveEven module in experiment#9

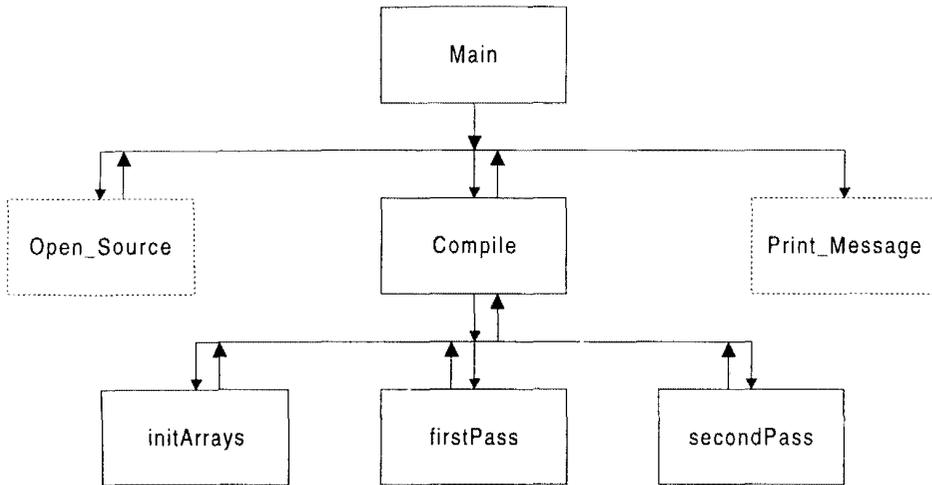


Figure C-10: A program structure of Open_Source module and Print_Message module in experiment#10

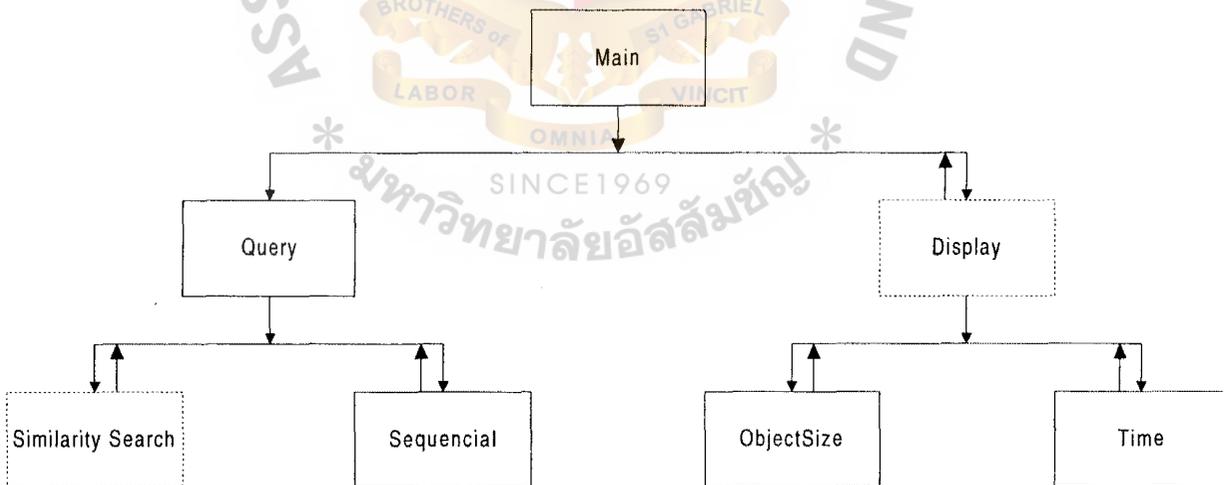


Figure C-11: A program structure of Similarity Search module in experiment#11

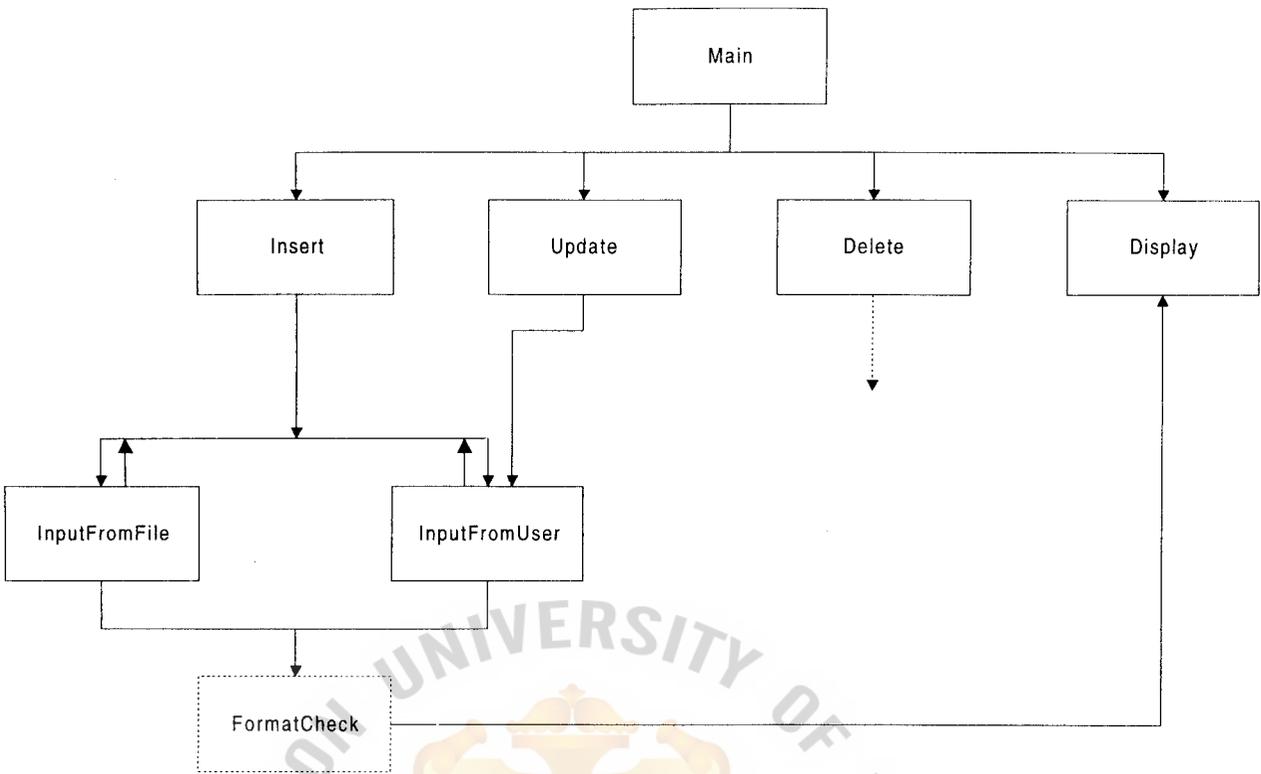


Figure C-12: A program structure of CheckFormat module in experiment#12

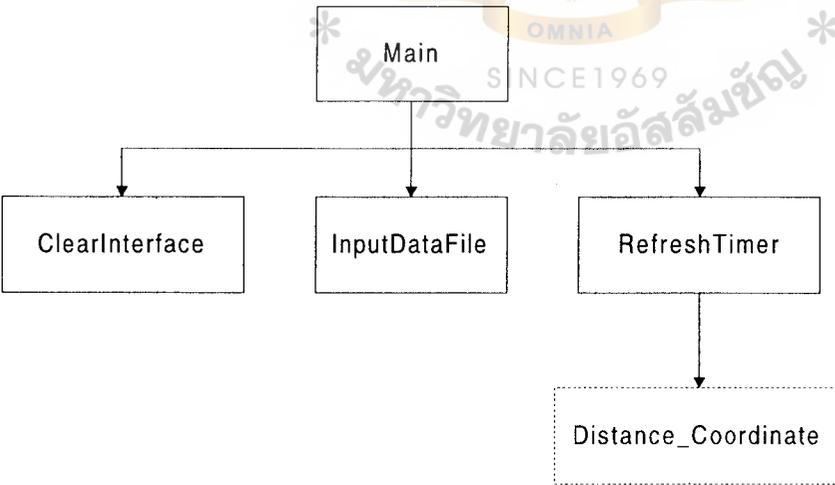


Figure C-13: A program structure of Distance_Coordinate module in experiment#13

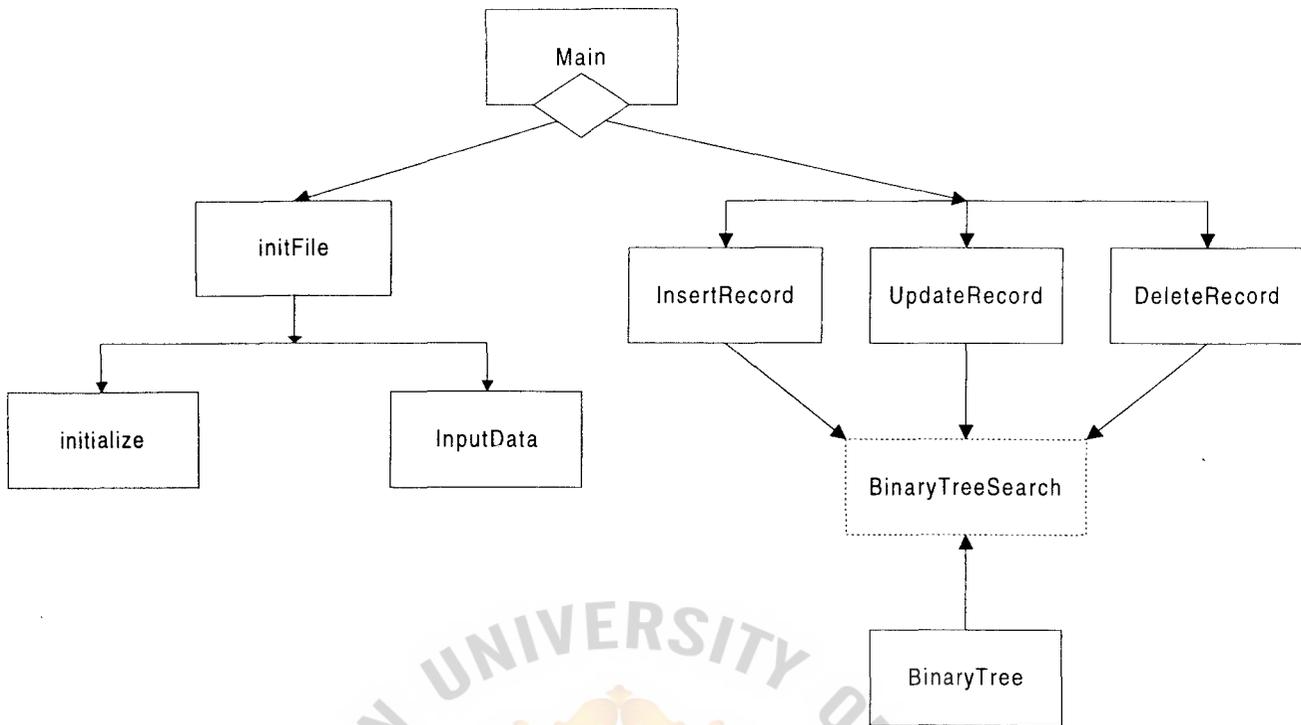


Figure C-14: A program structure of BinaryTreeSeach module in experiment#14

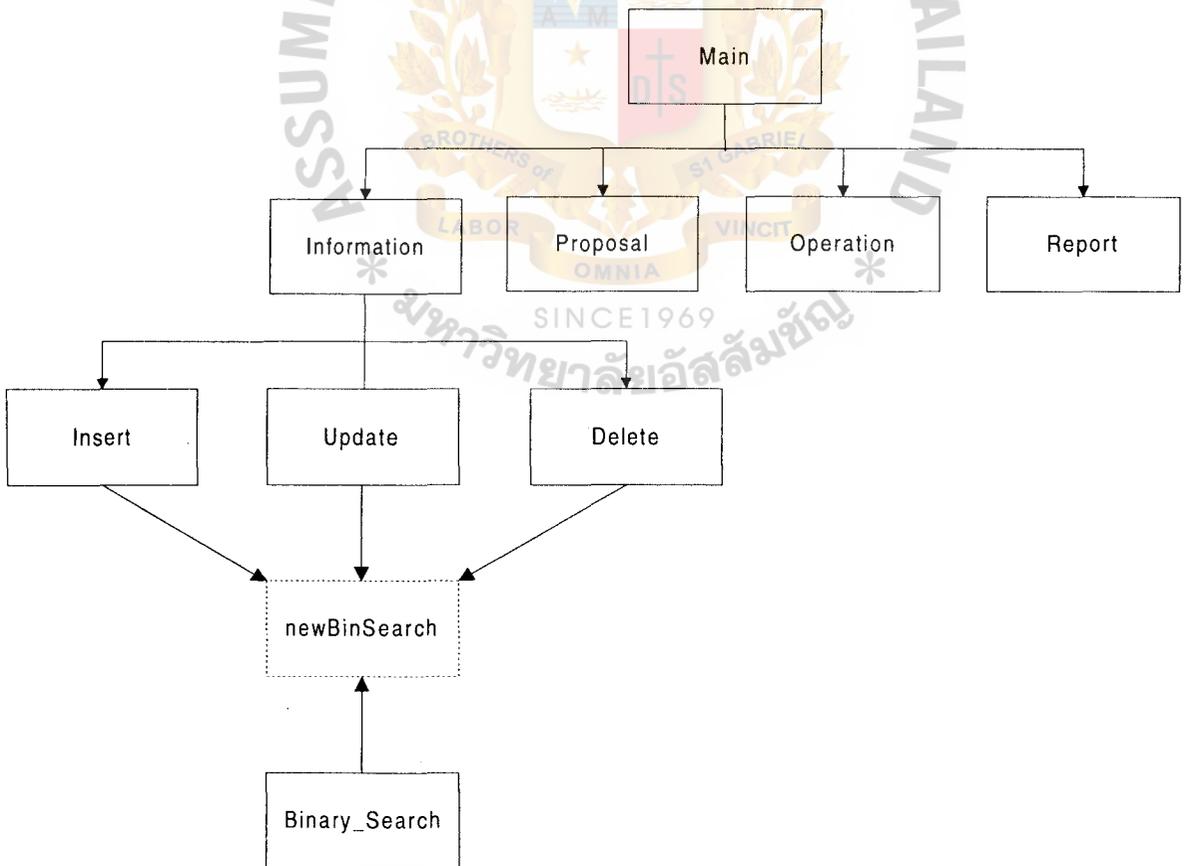


Figure C-15: A program structure of NewBinSearch module in experiment#15

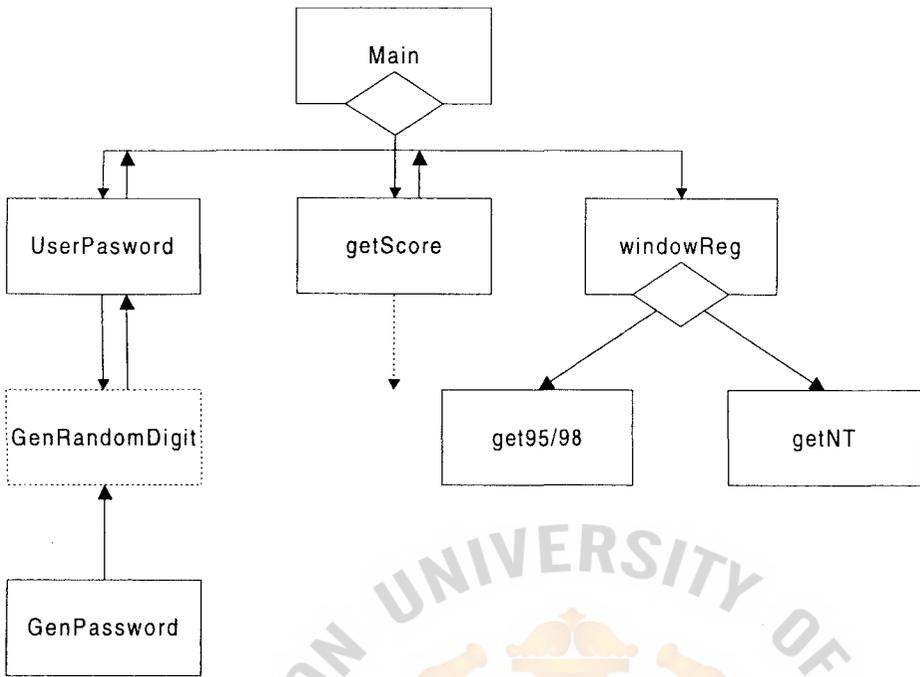


Figure C-16: A program structure of GenRamdonDigit module in experiment#16

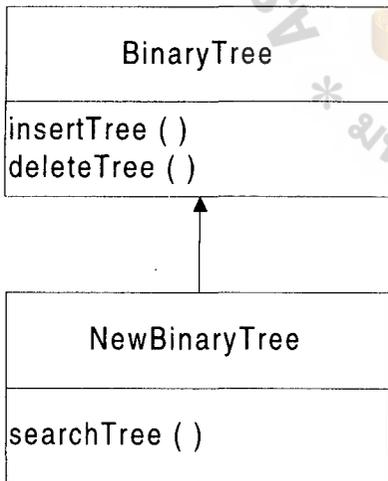


Figure C-17: A program structure of NewBinaryTree object in experiment#17

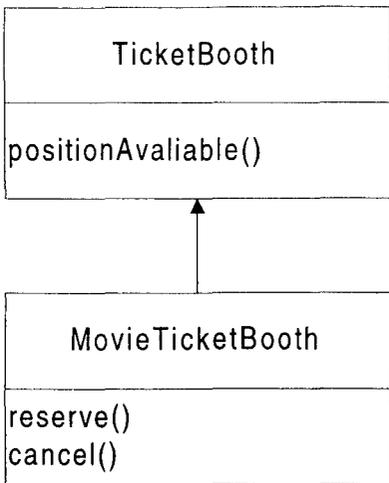


Figure C-18: A program structure of MovieTicketBooth object in experiment#18

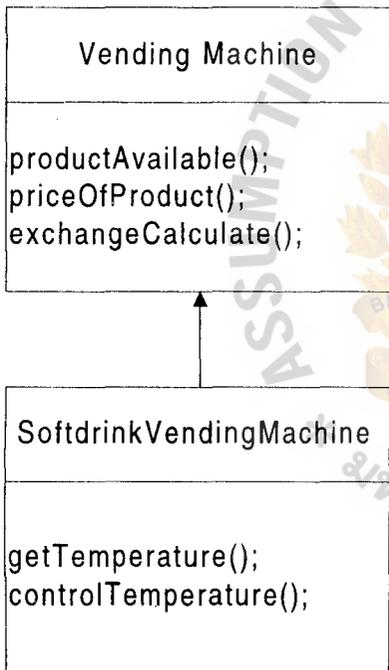


Figure C-19: A program structure of SoftdrinkVendingMachine object in experiment#19

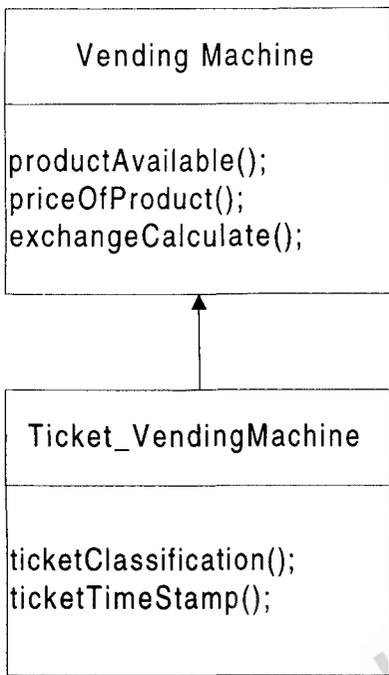


Figure C-20: A program structure of Ticket_VendingMachine object in experiment#20



St. Gabriel's Library

