# *Ada code reuse guidelines for design-for-reuse*

Kim, Hyoseob

# Ada Code Reuse Guidelines for Design-for-Reuse

Hyoseob Kim

M.Sc. Thesis

Centre for Software Maintenance
Department of Computer Science
University of Durham

October 1995

# Abstract

The phenomenal growth in the costs of producing software over the last three decades has forced the computing industry to look for alternative strategies to that implied by the waterfall model of computer system development. One frequently observed solution is that of reusing the code from previously designed systems in the construction of new ones; this technique is known as *software reuse.*

Ada language was developed as a tool to address the above problems and is believed to have many useful language features such as *package* and *generics* to produce reusable software. But programming in Ada does not guarantee the production of highly reusable software. Therefore guidelines for users are needed to maximise the benefits from using Ada. In this thesis, Ada code reuse guidelines are proposed, and as an attempt to prove the usefulness of them, reuse metrics are studied.

The thesis concludes by stressing the novelty of the approach, the difficulties encountered, and enhancements to the proposed methods to overcome these shortcomings.

# Acknowledgements

A technical work of this size can obviously not be produced without a great deal of help, advice and encouragement from others. A number of people have aided and abetted in its production.

I owe a great debt of gratitude to my supervisor Cornelia Boldyreff, who was always on hand to listen to my ideas and always willing to contribute her own. Also, she took the time to read the drafts of this thesis and provided invaluable feedback.

On a personal level, my deepest thanks are due to my parents who have always supported their son emotionally and financially.

## Copyright

## Declaration

No part of the material offered has previously been submitted by the author for a degree in the University of Durham or in any other University. All of the work presented here is the sole work of the author and no-one else.

# Contents

**1 Introduction** **1**

1.1 Introduction . . . . . 1

1.2 The Criteria for Success . . . . . 3

1.3 Outline of Thesis . . . . . 3

**2 Software Reuse** **5**

2.1 Introduction . . . . . 5

2.2 Benefits of Software Reuse . . . . . 6

2.3 Existing Software Reuse Methods . . . . . 9

2.4 Factors militating against Software Reuse . . . . . 11

2.4.1 Technical Factors . . . . . 11

2.4.2 Cultural Factors . . . . . 12

2.4.3 Managerial Factors . . . . . 13

2.4.4 Legal Factors . . . . . 13

2.5 Summary . . . . . 15

**3 Ada Reuse Guidelines** **16**

3.1 Ada Language . . . . . 16

3.1.1 History of Ada . . . . . 16

3.1.2 The Major Features of Ada . . . . . 17

3.2 Existing Guidelines . . . . . 19

3.3 Characteristics of Reusable Components . . . . . . . . . . . . . . . . . . . . . . 20
3.4 Standards Possessed by Good Guidelines . . . . . . . . . . . . . . . . . . . . . . . 21
3.5 Guidelines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.5.1 Principle of Adaptability . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.5.2 Principle of Comprehensibility . . . . . . . . . . . . . . . . . . . . . . . 30
3.5.3 Principle of Independence . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.5.4 Principle of Robustness . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

**4 Reuse Metrics: Metrics regarding Software Reuse 49**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.2 Software Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.2.1 Objectives of Reuse Metrics . . . . . . . . . . . . . . . . . . . . . . . 50
4.2.2 Measurement Theory in Software Engineering . . . . . . . . . . . . . 51
4.2.3 Characteristics possessed by Ideal Metrics . . . . . . . . . . . . . . . 55
4.2.4 Reuse Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

**5 Case Study 61**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.2 Experimental Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
5.2.1 Experimental Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
5.2.2 Experimental Materials . . . . . . . . . . . . . . . . . . . . . . . . . 62
5.2.3 Experimental Framework . . . . . . . . . . . . . . . . . . . . . . . . 62
5.3 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
5.3.1 Adaptability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
5.3.2 Comprehensibility . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.3.3 Independence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

5.3.4 Robustness . . . . . . . . . . . . . . . . . . . . . . . . . . 73

5.3.5 Reusability . . . . . . . . . . . . . . . . . . . . . . . . . . 74

5.4 Experimental Analysis . . . . . . . . . . . . . . . . . . . . . . . . 75

5.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

**6 Evaluation** **80**

6.1 Evaluation Against the Criteria for Success . . . . . . . . . . . . . . . . . . 80

6.2 Strengths and Weaknesses . . . . . . . . . . . . . . . . . . . . . . 81

6.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

**7 Conclusions** **83**

7.1 The Main Achievements of the Research . . . . . . . . . . . . . . . . 83

7.2 General Conclusions of the Research . . . . . . . . . . . . . . . . . 83

7.3 The Limitations of the Approach . . . . . . . . . . . . . . . . . . . 84

7.4 Suggestions for Future Research . . . . . . . . . . . . . . . . . . . 84

7.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

**A Collected Ada Code Reuse Guidelines** **85**

A.1 Principle of Adaptability . . . . . . . . . . . . . . . . . . . . . . . 85

A.2 Principle of Comprehensibility . . . . . . . . . . . . . . . . . . . . 86

A.3 Principle of Independence . . . . . . . . . . . . . . . . . . . . . . 87

A.4 Principle of Robustness . . . . . . . . . . . . . . . . . . . . . . . 88

**B Materials used in the Case Study** **89**

B.1 Introduction to 3 repositories used . . . . . . . . . . . . . . . . . . 89

B.1.1 AdaBasis . . . . . . . . . . . . . . . . . . . . . . . . . . 89

B.1.2 ELSA . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

B.1.3 LGL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

B.2 Introduction to 3 CASE Tools used . . . . . . . . . . . . . . . . . . 90

B.2.1 Ada System Dependency Analyzer 2.1 . . . . . . . . . . . . . . . . . 90

B.2.2 Certifier_1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91

B.2.3 Met_Pars . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

B.3 Source Codes used . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

**Bibliography** **104**

# List of Tables

5.1 Materials used in this Case Study . . . . . . . . . . . . . . . . . . . . . . . . . 63

# List of Figures

1.1 Software Development Process . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.2 A Software System as a Collection of Components . . . . . . . . . . . . . . . 3

1.3 Research Coverage . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1 An approach to modelling reusability . . . . . . . . . . . . . . . . . . . . 7

2.2 Adaptive Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.3 Parameterised Process . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.4 Engineered Process . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.1 Three steps relevant to the use of a generic unit . . . . . . . . . . . . . . . . 24

3.2 Kinds of Abstract Data Objects and Types available in Ada . . . . . . . . . 27

3.3 Three Kinds of Independence . . . . . . . . . . . . . . . . . . . . . . . 40

3.4 A method to improve independence of components . . . . . . . . . . . . . . 42

4.1 One-to-one mapping and one-to-many mapping . . . . . . . . . . . . . . . . 52

4.2 Measurement and the intelligent barrier . . . . . . . . . . . . . . . . . . . 54

4.3 Relationships among classes of scales . . . . . . . . . . . . . . . . . . . . 55

5.1 The GQM Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

5.2 Graph showing No. of Units in 7 Stacks with their Normal Curve . . . . . . 65

5.3 Adaptability of 7 Stacks . . . . . . . . . . . . . . . . . . . . . . . . . . 66

5.4 Comprehensibility of 7 Stacks . . . . . . . . . . . . . . . . . . . . . . . 70

5.5 Independence of 7 Stacks . . . . . . . . . . . . . . . . . . . . . . . . . 73

5.6 Robustness of 7 Stacks . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

5.7 Reusability of 7 Stacks . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

5.8 Relationship between disparity of properties and reusability of components . 76

5.9 Guideline Usage Statistics on Adaptability . . . . . . . . . . . . . . . . . . . 77

5.10 Guideline Usage Statistics on Comprehensibility . . . . . . . . . . . . . . . . 78

5.11 Guideline Usage Statistics on Independence . . . . . . . . . . . . . . . . . . 78

5.12 Guideline Usage Statistics on Robustness . . . . . . . . . . . . . . . . . . . . 79

5.13 Guideline Usage Statistics on Reusability . . . . . . . . . . . . . . . . . . . . 79

6.1 Reusability as a combination of overlapped properties . . . . . . . . . . . . . 81

# Chapter 1

# Introduction

## 1.1 Introduction

The *software crisis* has been with us for quite some time [1], and is not diminishing. As many researchers express, the crisis is represented by two major phenomena. According to a Software Engineering Institute (SEI) report [2], we lack the capacity to produce software. As hardware prices dramatically decrease, these days more people can own their own hardware systems[1]. So the demand for software by which hardware systems operate is exploding, while programmers' productivity is limited. According to statistical data [3], demand for software increases at a rate of 12%/year, while productivity and the number of personnel involved in software projects increase only at a rate of 4%/year each. Further evidence of this difference is the fact that many software projects finish over budget. This difference between demand and supply for software resulted in an enormous gap between hardware and software development during the past few decades. The SEI report states that *post-deployment software support* (*PDSS* or *maintenance*) is the most rapidly growing workload of the software process. Maintenence has long been known to be a large consumer of software budgets, with estimates from 40 to 70 percent [1]. Increases in demand by the maintenance phase lead to further reduction in new software development capacity because personnel are siphoned off. As one of the many attempts to address this problem of maintaining software, *the Centre for Software Maintenance* was established at University of Durham, in England, in 1987 [4].

Another aspect of the software crisis is the lack of quality. Although quality can be a subjective characteristic, overall system quality usually can be assessed in terms of providing the functionality expected by the customer, meeting customer performance requirements, and freedom from defects. In addition to them, the quality factors of a software system also contain working as advertised, having acceptable usage of time and space resources (*efficiency*), being composable with other components (*composibility*), being understandable

[1]In this thesis, hardware systems will mean not only computers, but also other peripheral devices such as printers, scanners, etc. which also need software to operate them.

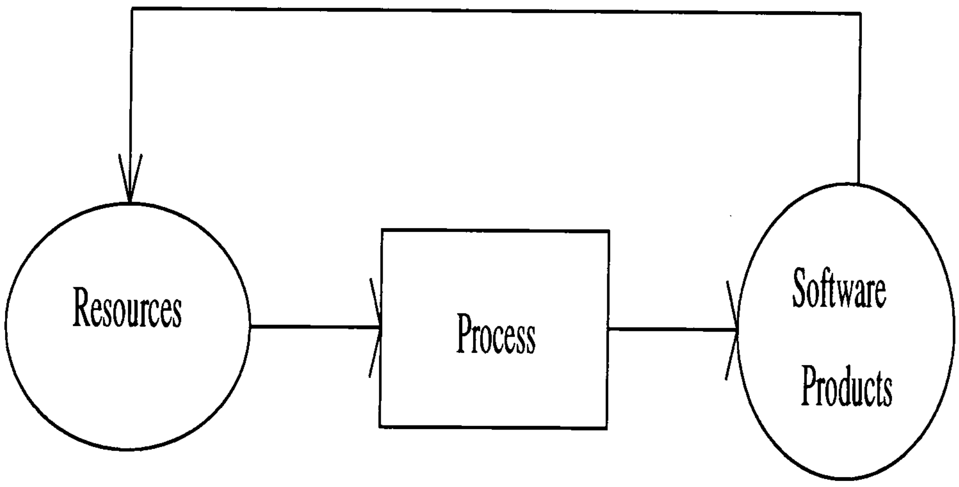


Figure 1.1: Software Development Process

by clients and maintainers (*readability*), and being usable in a possibly different context (*portability* or *rehostability*) [5].

Two different approaches to overcoming the software crisis, i.e. improving programmers' productivity and software quality have been tried. One is *process-oriented approach* and the other is called *component-oriented approach.*

In the former approach, many software engineers have focussed on improving the software development processes. As described in Figure 1.1, software products are produced through a series of process with resources.

This approach usually includes the use of computer-aided software engineering (CASE) tools. The hope is that improvements in how an organisation goes about managing software development will lead to better productivity and to higher quality systems [6].

Another approach to the quality and productivity problem, similar to but crucially different from the software-process approach, is improving the design and implementation of individual components of a software system [5]. Compared to the above approach, this is bottom-up thinking. Rather than working on a grand scale, this approach attempts to apply software engineering principles to component design in order to achieve improvements one component at a time. The idea is that small gains in quality and productivity at the component level will accrue to substantial gains over the entire software system. Figure 1.2 represents a software systems as a compound comprising many components and their relationships. Because we adopt already tested and validated, existing components, the mother system will have smaller faults than one developed from scratch. It all results in cheaper maintenance cost.

Although the former approach will be briefly explained, chapter 2, the introductory chapter on software reuse, mainly is targeted at the latter, with respect to "Ada".

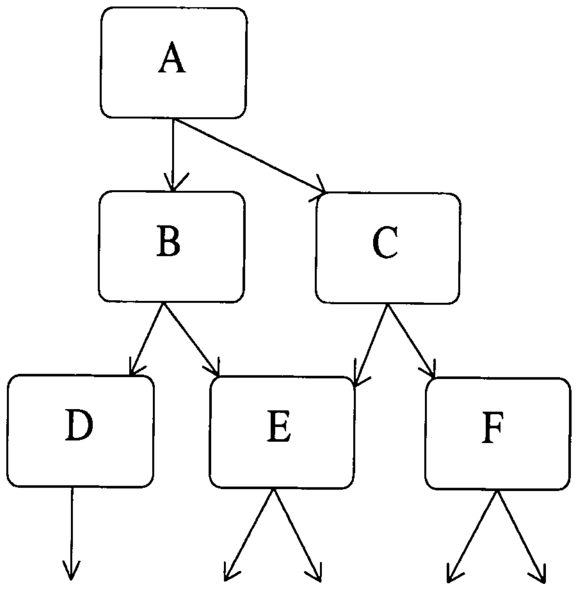


Figure 1.2: A Software System as a Collection of Components

## 1.2 The Criteria for Success

The main objective of this research is to propose a method[2] for producing Ada components of high reusability. The criteria for the success of the method are the following:

1. identifying the differences between Ada code reuse and high level reuse or code reuse in other programming languages;
2. establishment of the exact meaning of the term *reusability*;
3. suggesting guidelines for an approach to code reuse;
4. validating the usefulness and usability of the guidelines by software measurement.

Those criteria will be judged in chapter 6, Evaluation.

## 1.3 Outline of Thesis

The thesis is organised as follows. Chapter 2 introduces general principles and concepts relating to software reuse. The chapter handles *design with reuse* and *design reuse* as well as *design for reuse*, although the main topic of this thesis is the last one. Chapter 3 shows how to maximise code reuse in Ada, which is believed broadly to be one of the best languages in which to implement a reuse scheme efficiently and easily. In chapter 4, *reuse metrics* are

[2]In this context by method, the author means, a systematic approach with defined procedures.

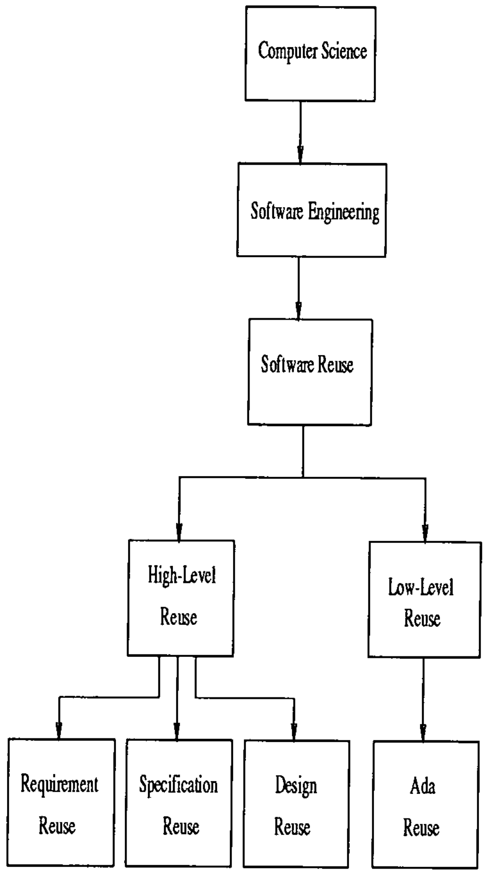


Figure 1.3: Research Coverage

studied to evaluate and validate the impact on applying the reuse guidelines of chapter 3 to Ada code. After that, experiments with the guidelines of chapter 3 and reuse metrics of chapter 4 upon some Ada code are done in chapter 5. Validity of the Ada code reuse guidelines and reuse measurement proposed is evaluated in chapter 6 on the basis of the results from chapter 5. Finally chapter 7 discusses possible future work and conclusions drawn from the work so far.

Figure 1.3 describes the whole structure of the thesis. The work presented in this thesis has links with other research topics in software engineering such as "software cost estimation", "software safety" or "reengineering". For reuse to flourish, correct software cost estimation is needed to investigate the impacts in organisations where reuse is being implemented. And, in code reuse, especially in the case of Ada, it is important for safe and reliable software to be built, for Ada is mainly used in large projects. Code reuse is considered an approach to building such safe software, along with formal methods. Reengineering techniques are needed to make legacy systems more reusable.

# Chapter 2

# Software Reuse

This chapter attempts to provide a framework for the study of software reuse, and *Ada code reuse* in particular. First of all, definitions of software reuse and history of research carried out on it are studied. Then the potential benefits of the introduction of widespread reuse are described to identify the motives of reusing software. After that, existing software reuse methods are introduced and the reasons why code reuse is still a viable proposition are argued, although much bigger benefits are expected from so-called *high-level reuse.* Finally the barriers that have to be overcome for successful adoption of reuse to happen are investigated.

## 2.1 Introduction

Software reuse is not a new idea. It has existed since the early days of computing in specialised domains in the form of shared programmer knowledge and subroutine libraries. As early as 1953, Wilkes and others had already recognised the importance of subprogram libraries of reusable program [7]. However, it had not been broadly advocated as a means for program construction until McIlroy [8] first proposed a *component manufacturing facility* based on code at the NATO Software Engineering Conference held at Garmisch in 1968. At the time of his speech, his idea of a component manufacturing facility was dismissed since technologies then were not mature enough to implement it. Large reductions in hardware costs in the last decade, however, have increased the significance of software development and maintenance expenditure. Namely, the main focus has shifted from hardware to software as software costs have overtaken hardware costs. This shift of emphasis has led to software reuse becoming an element of active software engineering research.

Among many reasons which caused this huge gap between hardware cost and software cost, the representative one can be found in whether we reuse existing products or processes or not. As a matter of fact, in the area of the electronics industry, reusing components is a common practice. Hardware systems are now developed by the selection and combination of standard integrated circuits, which encapsulate massive amounts of functionality. This packaging concept has enabled hardware components to be created which perform a particular 'service' without the designer needing to know details of internal operation [9]. The above methods used by the hardware industry are also strong points which the Ada language supplies, and will be discussed in chapter 3 in detail.

Regarding software reuse, there exist many definitions. Although a quite narrow definition that "software reuse is re-application of source code" is possible, a much broader definition is needed to get more benefits from reusing software artifacts since only 13% of the whole investment during the software life-cycle is spent at the phase of coding. In terms of this, Biggerstaff's following definition is more suitable [10]:

> Software reuse is the re-application of various type of knowledge about a certain system with the aim of reducing the burden of development and maintenance. The reusable elements consist of domain knowledge, development experiences, project choices, architectural structures, specifications, code, documentation and so on.

According to the above definition, anything produced during a software project becomes an object of reuse. As another important term, we need to know the correct meaning of "reusability". It is defined as follows [11]:

> The ability to reuse a software component or to use it repeatedly in applications other than the one for which it was originally built.

In addition to the above definition, since we have little knowledge about the characteristics of reusability, attempts to decompose it into better known characteristics have been carried out. One model of them, where reusability is broken into five factors, is shown in figure 2.1. It was drawn by Fenton [12] due to McCall and Boehm et al.

In the following sections, benefits of software reuse and existing software reuse methods and, finally, factors to overcome to accomplish successful software reuse are studied.

## 2.2 Benefits of Software Reuse

The incentive for software reuse comes from the amount of replication that is performed in software creation. In a California study of banking and insurance applications [13], 75% of functions were found to be common to more than one program. Furthermore, only 30% of software developed was concerned with the actual application, the other 70% being application independent (define/equate data items, format reports, perform validation). Lanergan and Grasso [14] found 40–60% of actual program code in a missile factory repeated in more than one application. Jones [13] tentatively concluded that up to 85% of all the code written in 1983 may have been of a common, generic nature.

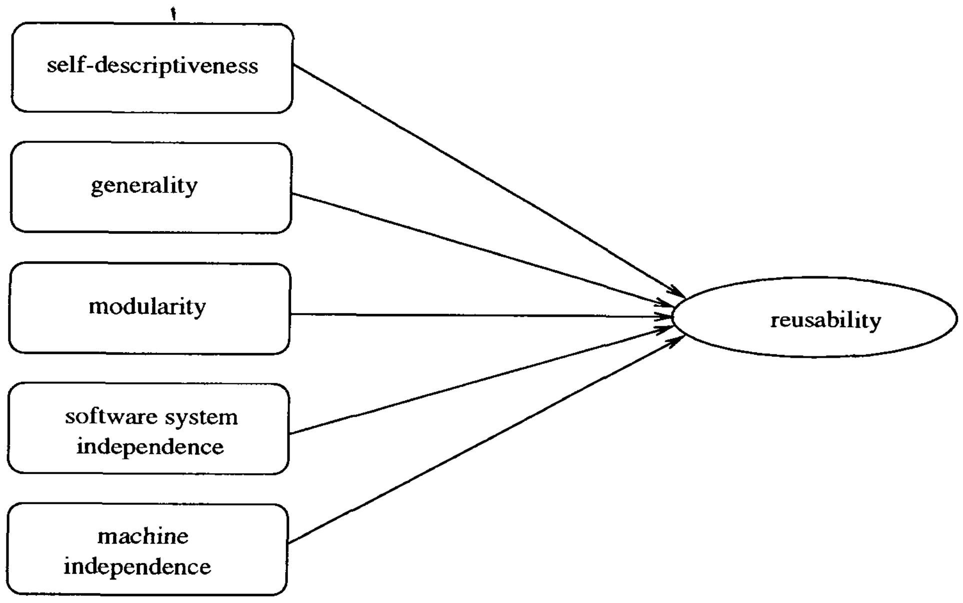


Figure 2.1: An approach to modelling reusability

Another important motive for software reuse is the economic aspect. Software costs continue to escalate rapidly and this has lead to alarm within the computing profession. Indeed this 'software crisis', first identified over 25 years ago [8], continues to plague us. The costs of producing software have been dramatically increasing, and this has only been slightly offset by computer hardware productivity advances. According to Boehm's research, software costs are increasing at a rate of approximately 12% per year [15].

The huge software cost has been mainly caused from the following three reasons [16, 17].

1. User requirements for software are getting more and more complex.

   The construction of larger and more ambitious software projects and the stringent environments they operate in impose severe performance requirements on the production of new software. Typical examples of these are found in *real-time, embedded systems.*

2. Demand for qualified personnel for building and maintaining software is ever increasing.

   At present, there is a critical shortfall in the number of qualified personnel entering the employment market. This fact, allied to a rapid escalation in salaries, makes the cost of any software development expensive and the idea of reusing existing software sensible.

3. Software development technologies developed during the last few decades failed to catch up to needed and expected growth in software productivity.

   Rates of software productivity have been creeping forward, as opposed to the leaps and bounds achieved in hardware. With new tools and methodologies, only 4% per year of the rate of the productivity growth has been achieved, while demand for new software has been increasing 12% per year [18, pages 6–12].

In addition to economic benefit, the application of software reuse leads to the construction of more reliable systems. The best test for reliability of code is its actual use within a system. During its functioning within the system, errors within it should have been noticed and fixed; thus it will have already conformed to an error-checking process. Theoretically, the more times a component is reused, the more confidence can be placed in it. This aspect of software reuse is especially important to persuade people to reuse other people's components. The use of computer software in life-critical applications is ever increasing. From civil air transports to nuclear power plants, computer software is finding its way into more life-critical applications every year. There are two sources of error with which an ultra-reliable system deal [19]

1. system failure due to physical component failure
2. system failure due to design errors.

At the moment, as software engineers, neither can we handle the former one, nor are concerned with it. Thus along with formal specification and verification, using reusable components makes sense in terms of software reliability.

Reusing such time-tested software also decreases maintenance cost substantially. It is estimated that 60% to 70% of the total life-cycle costs are spent on maintenance. In order to make changes, it is necessary first to understand the software and this could involve around 47% to 60% of the maintenance effort. This means that some 30–35% of the total life-cycle costs are consumed in understanding software after it has been delivered in order to make changes [20]. Another investigation conducted by Horowitz and Munson shows almost same result [16]. They estimate that maintenance of a software system exceeds the development cost of the original system by a factor of three. Through reusing pre-validated software, it is anticipated that smaller number of defects would arise from built software. Another advantage is that less time would be required to maintain software system since reusable components are easier to adapt than ones which are not so.

Software reuse also aids in the production of standards within an organisation. A typical source of inconsistency within a system is that fundamentally similar operations are carried out in totally different ways. A reuse technology will promote the coding of frequently used routines as components as done in electronics industry, and the reuse of these will guarantee consistency within a suite of programs. Furthermore, this standardisation will aid the process of writing more understandable and consistent code.

Another argument for the introduction of software reuse is the utilisation of specialised personnels' knowledge and expertise beyond time and space. Personnel having been involved in a certain project often are not available when needed. Therefore reusing software components allows reusers and maintainers to utilise experts without involving the persons who had actually developed them.

Finally, software reuse can help developers and maintainers estimate cost more correctly and quickly. Over budget and late delivery mean not only developing complex software is difficult, but also the cost estimation itself is wrong. Through reusing previous software products, more correct cost prediction can be performed.

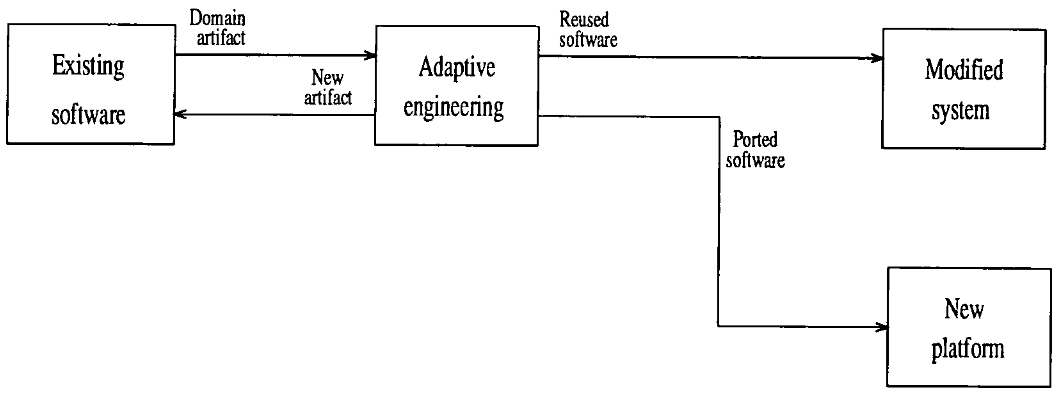


Figure 2.2: Adaptive Process

## 2.3 Existing Software Reuse Methods

The adoption of reuse into software development will include the definition of new products and processes. On the product side, we must identify the form of deliverables that can support reuse; on the process side, the approach needed to develop and apply those products. In order to move from the current, *ad hoc* approach to reuse, to a systematic reuse process, we must be able to both abstract a problem domain and create reusable solutions [21].

Three kinds of reuse which are the most representative are as follows:

1. Design-for-Reuse
2. Design-with-Reuse
3. Design Reuse.

It has been said that design-for-reuse should precede design-with-reuse. In other words, components which were not developed for future reuse in mind would need modification resulting in higher cost before reusing them than other reusable ones.

In the meantime, design reuse indicates value of attempts to reuse the earlier products during the software life cycle since they are believed to be less machine or language-dependent than source code.

Cohen of SEI classified the current reuse methods into three different processes [21]. The first process, one widely used today, is to adapt an existing system to meet a new set of requirements. The second, a relatively new practice, identifies families of programs, providing support for parameterisation of commonality, and customisation for unique requirements. The third process is an abstract-based engineering approach to discovering and exploiting commonality in software systems as the basis for software development. Figure 2.2, 2.3 and 2.4 compare these three approaches.

Each process offers its own set of benefits and risks. The *adaptive approach* (figure 2.2) requires little new investment by an organisation, and can support new developments, provided

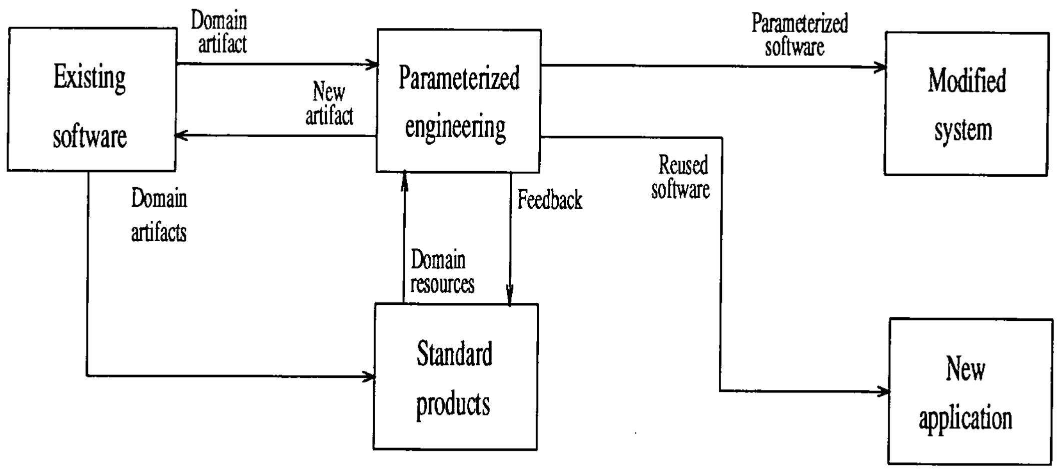


Figure 2.3: Parameterised Process

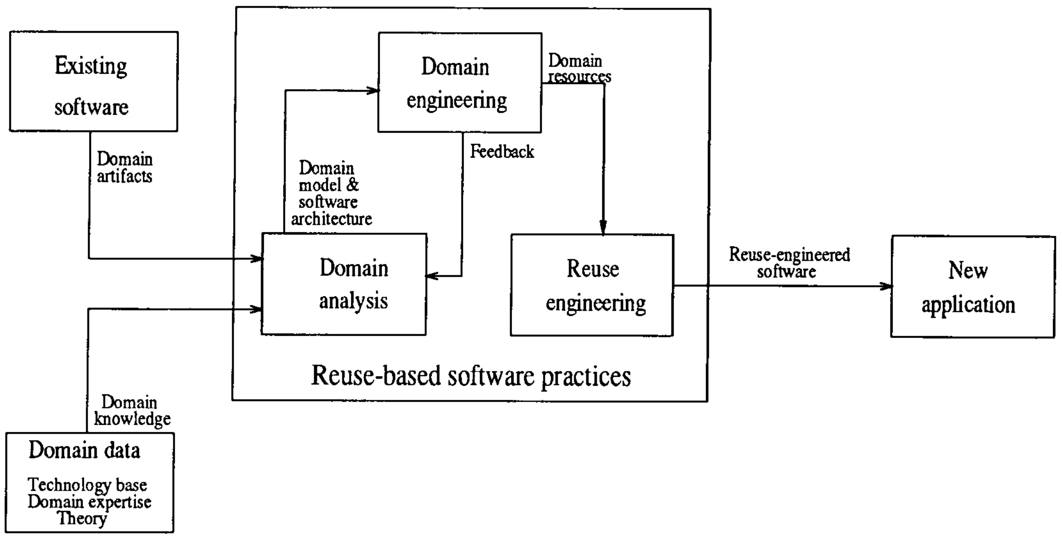


Figure 2.4: Engineered Process

they require only incremental changes from previous applications. However, applications that require major modifications and upgrades typical of most aerospace applications, will only achieve marginal benefits from adapting old software. The *parameterised approach* (figure 2.3) establishes a framework for all new implementations, a major investment for an organisation. There is a significant pay-off, provided the framework is stable. However, in areas with rapidly evolving technology, there is no stable framework. The investment in establishing standard products may be at risk if new requirements do not fit previously established standards. Like the parameterised approach, *engineered reuse approach* (figure 2.4) requires a large investment. The domain resources must meet the requirements of a wide range of applications or this investment will also be at risk. If the resources are properly developed this approach offers a greater degree of flexibility than the parametrised method, and can adapt to changing requirements.

Justification for Code Reuse

Even if bigger benefits can be obtained from reusing higher-level software artifacts, code reuse is still a worthwhile goal. The reasons are as follows:

First, code is a tangible entity. Among software products, few things are directly touchable and perceivable. Source code and Z specification are two of them. This fact is shown by that most of the software metrics developed until now are concerned with source code.

Secondly, code is more easily breakable into components than more abstract representations. With respect to developing reusable components and emerging into the so-called "software component industry", this kind of property is especially important. Simply we might not be able to develop components if we cannot divide software into components.

Thirdly, code reuse provides a higher possibility of successful implementation and diffusion in the short term. This is especially important to minimise managerial problems which will be explained in the following section.

## 2.4 Factors militating against Software Reuse

There are four kinds of barriers that have to be tackled before widespread reuse can be realised. They are technical factors, cultural factors, managerial factors and legal factors. And it has been shown that non-technical aspects are as important as technical ones.

### 2.4.1 Technical Factors

Sommerville identified six technical problems to be solved to success of software reuse [22].

Firstly, desirable attributes for reuse are to be investigated. Once we get to know about the characteristic, *reusability*, we would be able to develop high reusable, new components or re-engineer existing components in a cost-effective way to increase their reusability.

Secondly, methodology problems arise since most existing software design methods are intended to support software development without reuse. Therefore new development methodology is needed to open the so-called "software component industry".

Thirdly, new documentation standards for reusable components are to established. The documentation of a reusable component must specify both its functional and non-functional characteristics. Usually, more documentation is required than for components which are simply part of a larger system. Ideally, reusable components would be formally specified so that there is no ambiguity about their behaviour. However, this is unlikely to happen in the foreseeable future. Thus, more rigid documentation standards should be used to help users reuse the components more easily.

The fourth problem is about how components can be certified as reusable. In order to convince managers of the value of reuse, they must have confidence in the components which will be reused. This implies that we need some kind of *component certification scheme* which will certify the quality or usefulness of the component. But setting up such a scheme has been known as difficult and expensive. In chapter 5, "Case Study", a component certification tool is used.

Fifthly, probably the most important and frequently mentioned problems in the reuse research community are about component retrieval. In a large company (such as an aerospace company) there might be potentially hundreds of, if not thousands of, reusable components available. They are collected from many different computers and projects. Therefore finding what components exist and retrieving these components could be a major problem. Some cataloguing scheme using existing database systems must be established.

Finally, we can think about configuration management (CM) in reuse environment. The normal model of configuration management is currently project-based. The software developed as part of a project is maintained in a project archive. On the contrary, reuse requires software to be shared and, perhaps, components to be modified and stored in a software library or a software repository. The following questions can be asked, associated with configuration management. What relationships should be maintained between the reuse library and the original base components in the CM system? How should changes be propagated? How can traceability back to the original components be implemented? The answers to these questions are still being studied.

### 2.4.2 Cultural Factors

One of the fundamental questions that has to be answered is whether the structure of a society has an effect on the acceptance of reuse. It has been claimed [23] that there is a paradox between the application of a software reuse technology and the approach to life in a Western society. In the West, society tends to be very individualistic, with competitiveness rife in almost all fields of life. This results in an innovative approach to product development. It is argued that this conflicts with a reuse technology which relies on cooperation and trust for its successful application. It is noticeable that one of the best examples of success in applying reuse has occurred in "Japanese Software Factories" [24], in a society where a cooperative and paternalistic ethos is supported. The adoption of the SIGMA project by major industrial and academic bodies in Japan [25] is a venture that one would never expect to be undertaken in the West.

There is a very widespread phenomenon called "Not-Invented-Here (NIH)" syndrome within software community. This arises from the fact that software engineering is perceived as a skilled profession, and reuse implies a form of de-skilling, thus there is a lack of motivation to cultivate a reuse technology [9]. This can only be removed by supplying cheap components of high quality and encouraging sufficient management motivation.

### 2.4.3 Managerial Factors

A major factor in the successful implementation of reuse is its acceptance and encouragement by management [26]. Unless such backing is forthcoming, reuse stands little chance of success. There are many obstacles which have to be reconciled with the potential benefits outlined in section 2.2.

The first fact to be taken into consideration is the greater cost of producing reusable code compared to "solution-specific" production [9]. It is not easy to produce general or "generic" components that are suitable for reuse. This results in much more time and effort on the part of a software team, and greater cost for the project as a whole. Since project managers are rewarded for producing systems to deadline and within budgetary constraints, there is little incentive for them to encourage the production of generic components.

There is little quantitative evidence of the successful application of reuse in many fields. In incorporation of a reuse technology, management must be prepared to sacrifice short-term returns to gain unquantifiable benefits in the long-term. This is something many organisations are unprepared to risk. The only way this problem is likely to be alleviated is by wider scale availability of component libraries.

Management obstacles to reuse may be the most intractable of all to surmount. The adoption of risk-taking policies is necessary to promote the application of reuse, and demonstrate the immense benefits that can accrue from it. It is very much a "chicken-or-the-egg" situation, requiring go-ahead firms who are prepared to sacrifice returns in the short-term for the undeniable but unquantifiable benefits in the long-term.

### 2.4.4 Legal Factors

There exist two kind of legal issues, i.e. *intellectual property right* and *liability*. The former forces responsibility to keep copyright, patent, and trade secret laws, whereas the latter is about handling with any damage caused by a certain piece of software. Many decisions about the development, distribution, maintenance, enhancement and, especially, reuse of software are likely to be affected by constraints imposed by intellectual property laws and liability laws.

The primary purpose of the intellectual property laws is to encourage the development and dissemination of innovative works for use by the public. The creation or invention of useful items and artistic works generally requires the investment of considerable time, energy, and resources by skilled, talented people. To encourage such activities, the intellectual property laws provide, as an incentive, the opportunity to obtain exclusive rights to commercial exploitation of the innovative or artistic work for a specified period of time. Generally it is said that developing reusable components needs a big initial investment. So the developers' rights must be protected. Otherwise reuse would not happen. [27]

Intellectual property systems may be thought of as consisting of six elements [28]:

1. A definition of the subject matter to which the intellectual property law applies (e.g., machines are within the subject matter of patent, but not copyright).
2. A set of requisites for protection, which includes:
   - What qualities the subject matter must possess to be protectable (e.g., how much creativity must be shown to be entitled to intellectual property rights).
   - Who is entitled to assert the intellectual property right.
   - What procedural steps must be taken to acquire or retain the intellectual property rights.
3. A set of rights ("exclusive rights") to exclude other people from certain activities.
4. A public policy limitation on the extent of the owner's intellectual property rights.
5. A procedure for determining whether "infringement" has occurred. (An infringement is a violation of one of the exclusive rights.)
6. A specification of what remedies are available.

Although there are some intellectual property systems that do not apply to software, there are many that do. Many articles, books, and legal decisions discuss or hypothesise about the appropriate forms of intellectual property protection for computer programs. Unfortunately, there is as yet little certainty in this area of the law. Lawyers and legal scholars debate not only the present state of the law, but also the directions in which the law should be moving. For software is both a "writing" (traditionally copyright-protected) and a "machine" (traditionally patent-protected).

Copyright issues arise not only in external reuse environment, but also in internal reuse. For instance, if a component is developed by an employee, who will own its copyright between him and his employer? As another problem, nowadays many components are reverse-engineered. In this case, it must be made sure whether reverse-engineering old legacy codes is legal or not.

Another thing that we should consider when we reuse software is software product liability. A typical story of the topic is found in Armour and Humphrey's technical report [29] and is quoted below.

Voyne Ray Cox settled into the radiation machine for the eighth routine treatment of his largely cured cancer. The operator went to the control room and pushed some buttons. Soon, the machine went into action and the treatment began. A soft whir and then an intense pain made him yell for help and jump from the machine. The doctors assured him there was nothing to worry about. What they didn't know was that the operator had inadvertently pushed an unusual sequence of controls that activated a defective part of the software controlling the machine. He didn't die for six months but he had received a lethal dose of radiation. This software defect actually killed two patients and severely injured several others.

It has been believed that software defects are rarely lethal and the number of injuries and deaths is now very small. Software, however, is now the principal controlling element in many industrial and consumer products. In particular, the Ada language has been used in *safety-critical applications* such as nuclear power stations or aerospace industries. Thus, users are starting to realise that software, particularly poor quality software, can cause products to do strange and even terrifying things. Software bugs are erroneous instructions and, when computers encounter them, they do precisely what the defects instruct. As a worst case, an error could cause a 0 to be read as a 1, an up control to be shut down, or, as with the radiation machine quoted above, a shield to be removed instead of inserted. A software error could mean life or death.

The best way to overcome this problem is to develop software of high quality. Software reuse and SEI (Software Engineering Institute)'s CMM (Capability Maturity Model) are such attempts to achieve that goal. But until it becomes common practice, software products liability laws are needed.

Although it would be comforting to users to provide unequivocal answers to all important questions on intellectual property and software product liability, the fact is that the intellectual property laws and liability laws are in the process of evolving to provide adequate and appropriate protection for software.There are many questions for which there are as yet no clear answers.

## 2.5 Summary

In this chapter, background topics relating to software reuse were discussed. The software crisis was the main motivation for which the idea of reusing software was born. The benefits which can arise from reusing software were also discussed. After that, three of the most representative reuse methods were introduced and briefly studied. They were *adaptive process*, *parameterised process*, and *engineered process*. Finally, the reasons why software reuse is still long way to success were identified in terms of technical, cultural, managerial and legal factors.

# Chapter 3

# Ada Reuse Guidelines

This chapter proposes Ada reuse guidelines at the level of source code. To achieve that goal, firstly, section 3.1 discusses Ada's strong points as well as weak points with respect to reusing its source code. Then existing Ada code reuse guidelines are reviewed in section 3.2. Having done this, the standards possessed by good guidelines and characteristics of reusable components are discussed. Finally, guidelines are suggested on the basis of the above things to maximise the Ada's strong points while complementing and minimising its weak ones.

## 3.1 Ada Language

### 3.1.1 History of Ada

Related to Ada, several things are interesting. First of all, Ada was the second woman and a wife of Lamech after Eve who appeared in the Holy Bible with their names [30, Genesis 4:19]. However, the high level programming language Ada was named in honour of Augusta Ada Byron, the Countess of Lovelace and the daughter of English poet Lord Byron. She was the assistant, associate and supporter of Charles Babbage, the mathematician and inventor of a calculating machine called the Analytical Engine. Because she wrote some programs at that time, she is believed as the world's first computer programmer [31].

Ada, the language itself, was designed at the initiative and under the auspices of the United States Department of Defence (DoD). DoD studies in the early and in the middle 1970s indicated that enormous savings in software costs (about $24 billion between 1983 and 1999) might be achieved if the DoD used one common language for all its applications instead of the over 450 programming languages and incompatible dialects used by its programmers [31].

Then, starting with *Strawman* (1975), the language's requirements were refined through *Woodenman* (1975), *Tinman* (1976), *Ironman* (December 1978) and finally *Steelman* (1978) [32]. After that, an international competition was held to design a language based on the above requirements. Seventeen companies submitted proposals out of which four were selected as semi-finalists. The competition was won by a language designed by a team of computer scientists lead by Jean Ichbiah of CII Honeywell Bull. After some modifications, this language was named Ada by a member of Whitaker's group, navy commander John Cooper [33]. In February 1983, Ada became an ANSI standard [34]. Since then, DoD Directives 3405.1 and DoD Instruction 5000.2, as well as the FY91 DoD Appropriations Act, mandate use of Ada, where cost effective, for all applications [35].

### 3.1.2 The Major Features of Ada

Before explaining the unique features of Ada, it would be better to think of the motives for developing Ada. In addition to economic reason described in section 3.1.1, it was developed to address important and recognised problems in software development such as language simplicity, completeness, program reliability, correctness, maintainability, portability, the development of large programs, real-time programming and error handling [31].

As a result, Ada contains the following major features [36]:

- Structured Constructs — Ada is a modern block-structured language with a complete and regular set of program constructs.
- Strong Typing — Ada can detect many errors at compile time, as well as at run time. Languages such as Ada, for which it is possible to enforce type compatibility strictly, are said to be strongly typed. Ada is one of only a few languages that are truly strongly typed. Pascal, for example, often is said to be strongly typed, but it actually has an obscure loophole in its typing system that permits incompatible types to be mixed. Other languages, such as C, tout their lack of type checking as a feature.

  Sometimes users want a language that doesn't have type checking. In writing software such as compilers or operating systems, it sometimes is convenient to be able to ignore the data type of a value. Thus some programmers may prefer languages that allow them easily to forego compatibility checking. Of course, doing so requires that they be especially careful when writing programs that use different types. It is a bit like performing a high-wire act without the benefit of a safety net! [33].
- Modularity — Ada is written in modules with well-defined interfaces. Interfaces and internal implementations of modules are kept separate. This allows large and complex systems to be successfully developed in Ada. One of the main language features to implement this characteristic is package concept. Using it makes the text of a package body hide from its users. Through it, we can expect two benefits. One is *confidentiality*. A software producer supplying the services of a given package may want to protect his investment by not showing the package implementation. Another reason is known as *information hiding*. Letting a user read the implementation would create a danger that the user derive some additional implicit assumptions based on an analysis of the current implementation. Thus, in an Ada library, we can only access specifications, but cannot access bodies.

- Tasking — Concurrent programs can be directly realised in Ada.
- Exception Handling — Ada provides special constructs to handle both expected and unexpected errors.
- Generics — Ada provides a powerful means of developing reusable, tailorable components.
- Readability — Ada strongly supports the writing of clear, nearly self-documenting programs. It has been argued that Ada has a high degree of *readability*, whereas making writing programs a little more difficult than other languages. In other words, Ada language has a high *readability* but a low *writeability*.
- Data Abstraction — Data can be structured and described in meaningful terms, hiding unnecessary detail at each level through using private types and limited private types.
- Precision Specification — Programmers can specify the precision needed for different types of numeric data, ensuring portability of mathematical software.
- External Interfacing — Ada provides a complete set of low-level facilities, for example to handle interrupts and to control the exact layout of data and programs in memory.

Having the above features, Ada is believed to be suitable for developing software which has the following attributes [36]:

1. An expected life-time of comparatively long years, with changes and upgrades expected during this life-time.
2. A size and complexity that is too much for a single person to handle.
3. A requirement to deal with several simultaneous inputs, or to perform several concurrent tasks.
4. A requirement for portability.
5. A requirement for reusability: The Ada language was designed so that creation of reusable software would be relatively easy and straightforward [34].
6. A strong quality requirement (i.e. defects really matter, in financial and/or human terms).
7. A strong performance requirement (i.e. run-time and/or reaction time is important).

Therefore questions like "Is Ada better than C++ or Pascal?" are neither suitable nor useful. These considerations only make sense in terms of that environments or situations in which the needed software will be used. Instead, the question "What kind of software do we want to develop?" is more suitable to decide the programming languages to be used in software projects.

## 3.2 Existing Guidelines

Software component reuse is the key to significant gains in productivity. However, to achieve its full potential, our attention should be focussed on *development for reuse* or, in other words, *design for reuse*, which is a process of producing potentially reusable components [37]. It can be easily predicted that we would experience difficulties when trying to reuse a component that is not designed for reuse. In order to avoid such ad-hoc reuse style which is common nowadays, first of all, it must be made clear what the term "reusability" means. Having defined the characteristics of potentially reusable components, reuse guidelines are to be developed to represent such characteristics clearly.

These days almost every journal relevant to software engineering contain some articles on software reuse. Also in many conferences and workshops software reuse is a topic for discussion whether it is a main issue or a minor issue. It is also true that a substantial portion of conferences and workshops is associated with Ada. Thus, many Ada code reuse guidelines have been suggested. Among them, the following ones are notable:

- Nissen and Wallis's guidelines in 1984 [38]

  Theses guidelines were originally on "portability" issues, but also supply valuable guidelines on "reusability".

- St. Dennis's guidelines in 1986 [39]

  In his paper, he defines a set of characteristics of reusable software as well as guidelines for implementing them in the Ada language.

- the Ada-Europe Software Reuse Working Group's guidelines in 1990 [40]
- The Software Technology for Adaptable, Reliable Systems (STARS) Reusability Guidelines in 1990 [41]
- Ramachandran and Sommerville's guidelines in 1992 [42]
- Software Productivity Consortium (SPC) Services Corporation's guidelines in 1995 [43]

Although it is observed that each set of guidelines have been enhanced, compared to the previous ones, those guidelines are sometimes unrealisable and contradictory with respect to other guidelines. Therefore more complete and well-organised guidelines are needed.

At large those guidelines can be divided into two different groups in terms of each guidelines' structure. The first group of guidelines suggests guidelines enumerated in terms of language features. The Ada-Europe Software Reuse Working Group's, STARS', and Ramachandran and Sommerville's guideline belong to this group. On the other hand, the second group contains guidelines classified with characteristics contributing to reusability. The typical examples of this kind of guidelines are St. Dennis' ones and SPC's. In this thesis the second classification scheme is used to propose guidelines since the first one could become non-readable like a "Reference Manual for the Ada Programming Language" and thus avoided by users. Another reason is that the property of reusability can be addressed more easily and efficiently with the second group of guidelines.

In the following sections of this chapter, characteristics of reusable components and standards of useful guidelines are discussed. And then, based on the characteristics, guidelines with rationale and specific examples are proposed.

## 3.3 Characteristics of Reusable Components

Regardless of development method, experience indicates that reusable code has certain characteristics. St. Dennis posits the following 15 language-independent characteristics of reusable software [39]:

1. Interface is both syntactically and semantically clear.
2. Interface is written at appropriate (abstract) level.
3. Component does not interfere with its environment.
4. Component is designed as object-oriented; that is, packaged as typed data with procedures and functions which act on that data.
5. Actions based on function results are made at the next level up.
6. Component incorporates scaffolding for use during "building phase".
7. Separate the information needed to use software, its specification, from the details of its implementation, its body.
8. Component exhibits high cohesion/low coupling.
9. Component and interface are written to be readable by persons other than the author.
10. Component is written with the right balance between generality and specificity.
11. Component is accompanied by sufficient documentation to make it findable.
12. Component can be used without change or with only minor modification.
13. Insulate a component from host/target dependencies and assumptions about its environment; Isolate a component from format and content of information passed through it which it does not use.
14. Component is standardised in the areas of invoking controlling, terminating its function, error-handling, communication and structure.
15. Components should be written to exploit domain of applicability; components should constitute the right abstraction and modularity for the application.

In the mean time, SPC argues reusable software possesses the following four characteristics:

1. Reusable parts must be adaptable. To maximise its reuse potential, a part must be able to adapt to the needs of a wide variety of users.
2. Reusable parts must be understandable. A reusable part should be a model of clarity. The requirements for commenting reusable parts are even more stringent than those for parts specific to a particular application.
3. Reusable parts should be independent. It should be possible to reuse a single part without also adopting many other parts that are apparently unrelated. Also, they are ideally required not to contain environment or machine-dependent facts.
4. Reusable parts must be of the highest possible quality. They must be correct, reliable, and robust. An error or weakness in a reusable part may have far-reaching consequences, and it is important that other programmers can have a high degree of confidence in any parts offered for reuse. This is especially important to overcome the managerial barrier against successful software reuse.

After thoroughly examining the above two groups, it can be said that the former group can be incorporated into the latter one. Thus, the latter one is used to classify guidelines in the thesis.

## 3.4 Standards Possessed by Good Guidelines

In order to produce highly reusable components explained in the previous section, we need highly usable guidelines for users. Somerville et al. [44] argue that good guidelines must adhere to the following standards:

- They must be understandable by software engineers with a reasonable level of Ada programming expertise.

  There is no point in producing complex guidelines which rely on subtle knowledge of programming language semantics. Very few people understand such guidelines.

- They must be applicable without a great deal of additional effort.

  Guidelines will not be applied during development if it means taking longer to develop a component. In essence, they should help engineers make a design choice which has to be made anyway as part of the development process.

- They must be unambiguous.

  If guidelines are unambiguous, it is possible to decide whether or not they have been applied without detailed knowledge of the software component. This is important for *reuse certification.* An assessor of component reusability cannot be expected to have detailed knowledge of all components.

- They must recognise that embedded systems, which were the main goal of the development of Ada, usually have performance and memory utilisation requirements.

  Wherever possible, the suggestions made should not degrade the efficiency of a component. If changes are proposed, which affect the component's efficiency, this should be explicit so that the efficiency implications may be analysed.

In addition to the above things, good guidelines must supply rationale, and be validated either through empirical experiments or by formal argument. Simple and clear examples are needed to help users to understand and apply the guidelines.

Finally, it has been observed that some guidelines are contradictory to the proposers' other guidelines. All guidelines are to be consistent not to make users confused.

## 3.5 Guidelines

On the basis of the above characteristics of reusable components, guidelines follow below to address them. Reusability of a component is investigated as the four constituents, i.e. adaptability, comprehensibility, independence and, finally, robustness. Most of them were from existing guidelines found in the existing literature, although some were made by the author. Guidelines are regrouped into 4 groups to which each guidelines contribute.

### 3.5.1 Principle of Adaptability

Reusable parts often need to be changed before they can be used in a specific application. They should be structured so that change is easy and as localised as possible [43]. Here, two factors are mainly related to adaptability of code. They are "completeness" and "generality". Detailed explanations about them follow below.

Completeness

"Completeness" means that components should have all functions and operations for current and future needs. Ideally, each component should contain all of the functionality that can be associated with such a component. Completeness, however, can cause development effort to be spent on features not needed for the current project, but probably needed on future projects. It should be tempered by development cost, benefits provided by the component, and likelihood of use.

There exist two kinds of completeness, i.e. intra-completeness and inter-completeness. They means that not only should components themselves be complete, but also should the relationship between them be so. Guidelines A01–A03 are about the former, whereas A04 indicates the latter.

A01: Make components as complete as possible [41].

A02: Provide complete functionality in a reusable part or set of parts [43].

As explained above, it is impossible to implement component in a perfectly complete manner, but completeness is still a useful goal. To enhance reusability, components should be made as complete as practical.

Related to "abstract data types (ADTs)", the following three operator classes are needed: constructors, observers, and iterators [33, chapter 4]. "Constructors" are operations that alter the state of an abstract data type. "Observers" are operations that allow us to observe the state of an abstract data type without changing it. Finally, "iterators" indicate operations that allow us to process all the components in an abstract data type.

While this kind of strategy can guarantee correctness in implementing abstraction, it also can cause performance problems. When we mention "performance", two things are related to it. They are time constraint and space constraint. Whenever this guideline is applied to performance-critical situations, balancing between completeness and performance is needed.

A03: Provide initialisation and finalisation procedures for every data structure that may contain dynamic data.

Any application that must control memory should use the initialisation and finalisation routines to guard against memory leakage, which is concerned with space aspects of performance [18]. Although it is said much space problem was solved thanks to hardware technology, but this guideline is still needed, for Ada is mainly used for real-time, embedded systems, where time and space constraints are severe.

A04: Make all dependent components reusable.

This guideline is about "inter-completeness". A component is not fully reusable unless all the components it includes through *with*[1] are reusable. If a component depends on components that are not reusable, then potentially there may be portability and tailorability problems [41].

Generality

Matsumoto argues that component is written with the right balance between generality and specificity [45]. With respect to code generality, we can think of two things. One is use of generic units. Sommerville et al call this kind of code characteristic "component genericity", specifically [44]. Another is use of general names of units and identifiers.

Guidelines A05–A09 are about the former, whereas the latter is recommended in guidelines A10 and A11.

A05: Use generic units to avoid code duplication.

A06: Parameterise generic units for maximum adaptability.

A07: Use generic units to encapsulate algorithms independently of data type.

[1]In Ada community, the word *with* is used to mean the situation when a component includes other component(s).

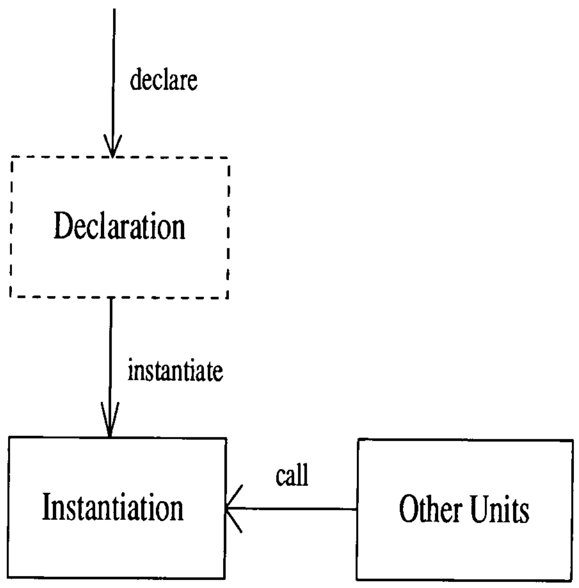


Figure 3.1: Three steps relevant to the use of a generic unit

Almost every author of each set of guidelines strongly recommends the use of the generic facility of Ada since using generics can improve adaptability of components dramatically. Another attractive point of using them is that the generic facilities in Ada have been specifically designed to support adaptability without run-time overhead [44]. As a matter of fact, generics are instantiated at compile time.

Using generic units, it is possible to produce as many objects as possible. Further, if they are parameterised, we can even get objects of different data types [43].

Most algorithms can be described independently of the data type. So Ada's generic facility is very useful for that kind of situations. As depicted in figure 3.1, normally, three steps — declaration, instantiation and call — are involved for users to use a generic unit practically.

A simple example is given below to show the mechanism of generic facility of Ada.

```
-----------------------------------------------------------------------

-- Declaration of a generic unit


-- Specification
generic
   type Element is limited private;
   type Data    is array (Positive range <>) of Element;
   with function ''<'' (Left  : in Element;
                        Right : in Element)
                        return    Boolean is <>;
   with procedure Swap (Left  : in out Element;
                                 Right : in out Element) is <>;
procedure Generic_Sort (Data_To_sort : in out Data);

-- Body
procedure Generic_Sort (Data_To_Sort : in out Data) is
begin
   ...
   for I in Data_To_Sort'range loop
      ...
         ...
         if Data_To_Sort(J) < Data_To_Sort(I) then
            Swap(Data_To_Sort(I), Data_To_Sort(J));
         end if;
         ...
      ...
   end loop;
   ...
end Generic_Sort;
-----------------------------------------------------------------------

-- Two possible instantiations

-- The first instantiation
type Integer_Array is array (Positive range <>) of Integer;
procedure Swap (Left  : in out Integer;
                Right : in out Integer);
procedure Sort is
       new Generic_Sort (Element => Integer,
                         Data    => Integer_Array);

-- The second instantiation
subtype String_80 is string (1 .. 80);
type    String_Array is array (Positive range <>) of string_80;
procedure Swap (Left  : in out String_80;
                Right : in out String_80);
procedure Sort is
```

```
   new Generic_Sort (Element => String_80,
                     Data    => String_Array);


-----------------------------------------------------------------------
-- Two possible callings


-- The first calling
Integer_Array_1 : Integer_Array (1 .. 100);
...
Sort (Integer_Array_1);


-- The second calling
String_Array_1 : String_Array (1 .. 100);
...
Sort (String_Array_1);
```

A08: Use abstract data types in preference to abstract data objects.

A09: Use generic units to implement abstract data types independently of their component data type.

Guidelines A08 and A09 appeared in the SPC's guidelines [43]. Associated with abstract data type or object, five different forms of implementation are possible. They are *abstract data object (ADO)*, *abstract data type (ADT)*, *generic abstract data object (GADO)*, *parameterised generic abstract data object (PGADO)*, and *generic abstract data type (GADT)*.

Figure 3.2, which was drawn by the author due to the SPC's guidelines describes the 4 kinds of data objects and type implementable in Ada. A diagram of PGADO was omitted for its difficulty in drawing it. Below, each examples of the five kinds are given with some explanation.

In Ada, five kinds of the abstract data object/type, "stack" can be implemented.

The first one is ADO. In the following example, only one stack of integers can be produced. So it is naturally lacking in genericity and powerfulness.

```
-- An ADO
package Bounded_Stack is
   subtype Element is Integer;
   Maximum_Stack_Size : constant := 100;
   procedure Push (New_Element : in     Element);
   procedure Pop  (Top_Element :    out Element);
```

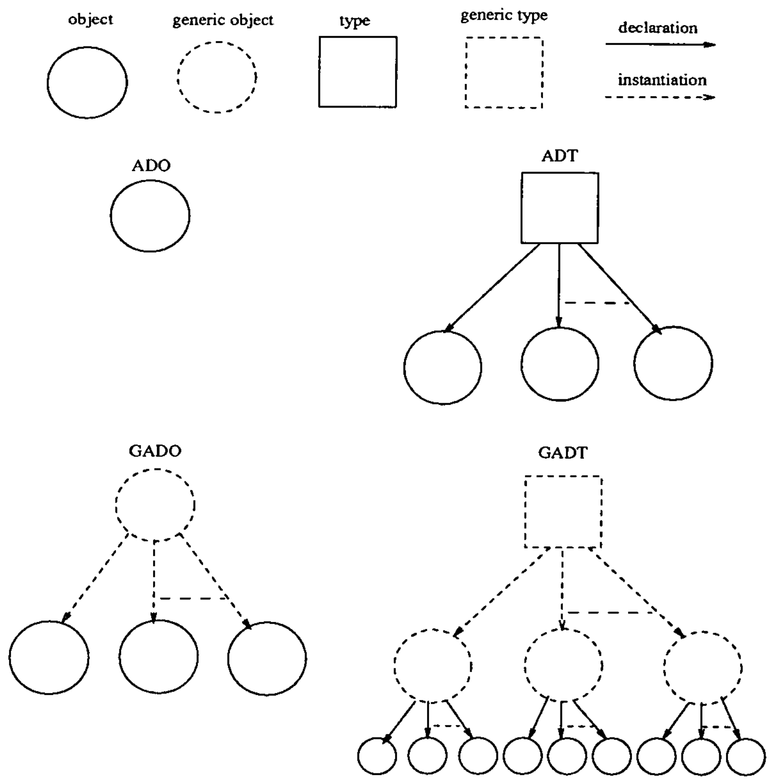


Figure 3.2: Kinds of Abstract Data Objects and Types available in Ada

```
   Overflow  : exception;
   Underflow : exception;
   ...
end Bounded_Stack;
```

The second one, ADT allows users to declare any number of stacks of integers by exporting the *Stack* type.

```
-- An ADT
package Bounded_Stack is
   subtype Element is Integer;
   type    Stack   is limited private;
   Maximum_Stack_Size : constant := 100;
   procedure Push (On_Top      : in out Stack;
                   New_Element : in     Element);
   procedure Pop  (From_Top    : in out Stack;
                   Top_Element :    out Element);
   Overflow  : exception;
   Underflow : exception;
   ...
private
   type Stack_Information;
   type Stack is access Stack_Information;
end Bounded_Stack;
```

The third one is a parameterless generic abstract data object (GADO). Since it is a generic unit, users can instantiate it multiple times to obtain multiple stacks of integers. It should, however, be noticed that only *integer* type of stacks can be obtained.

```
-- A GADO
generic
package Bounded_Stack is
   subtype Element is Integer;
   Maximum_Stack_Size : constant := 100;
   procedure Push (New_Element : in     Element);
   procedure Pop  (Top_Element :    out Element);
   Overflow  : exception;
   Underflow : exception;
   ...
end Bounded_Stack;
```

The fourth one is also a generic abstract data object but with parameters unlike the third one. Thus, stacks of data types other than "Integer" can be created.

```
-- A PGADO
generic
   type Element is limited private;
   with procedure Assign (From : in     Element;
                          To   : in out Element);
   Maximum_Stack_Size : in Natural := 100;

package Bounded_Stack is
   procedure Push (New_Element : in     Element);
   procedure Pop  (Top_Element :    out Element);
   Overflow  : exception;
   Underflow : exception;
   ...
end Bounded_Stack;
```

The last one, GADT is considered as the most powerful and flexible. That is because it allows users to produce virtually any number of stacks of any types. So whenever we implement an abstract data object/type, we should try to design as a GADT.

```
-- A GADT
generic
   type Element is limited private;
   with procedure Assign (From : in     Element;
                          To   : in out Element);
   Maximum_Stack_Size : in Natural := 100;

package Bounded_Stack is
   type Stack is limited private;
   procedure Push (On_Top      : in out Stack;
                   New_Element : in     Element);
   procedure Pop  (From_Top    : in out Stack;
                   Top_Element :    out Element);
   Overflow  : exception;
   Underflow : exception;
   ...
private
   type Stack_Information;
   type Stack is access Stack_Information;
end Bounded_Stack;
```

The biggest advantage of an ADT over an ADO (or a GADT over a GADO) is that the user of the package can declare as many objects as desired with an ADT. Another gain is that an ADT or a GADT provides more protection of the data structure than an ADO or a GADO since private types can be used in the formers.

Similarly, the biggest advantage of a GADT or GADO over an ADT or an ADO is that the formers can be parameterised with types, subprograms, and other configuration information since they are generic. So from the above facts, power and flexibility increase, approaching from an ADO to a GADT.

It is also observed that those advantages are not expensive in terms of complexity or development time. Therefore, wherever possible, a GADT is to be preferred to an ADO.

A10: Select the least restrictive names possible for reusable parts and their identifiers.

A11: Select the generic name to avoid conflicting with the naming conventions of instantiations of the generic.

Choosing a general or application-independent name for a reusable part encourages its wide reuse. When the part is used in a specific context, it can be instantiated (if generic) or renamed with a more specific name.

An example of applying the above two guidelines for general-purpose stack follows below.

```
generic
   type Item is limited private;
   ...
package Bounded_Stack is
   procedure Push (New_Item    : in Item);
   procedure Pop  (Newest_Item : in Item);
   ...
end Bounded_Stack;
```

The above general purpose stack abstraction can be renamed appropriately for use in current application as follows:

```
with Bounded_Stack;
package Cafeteria is
   type Tray is limited private;
   package Tray_Stack is new Bounded_Stack (Item => Tray, ...);
   ...
end Cafeteria;
```

## 3.5.2 Principle of Comprehensibility

These properties are also known as *readability*, *understandability* or *clarity.* As explained in chapter 2 of this thesis, program comprehension is the most expensive activity during maintaining software. Thus any attempt to improve understandability of software is warmly welcomed. According to the statistical data appearing in section 2.2, the implications are that if we want to improve software development, we should look at maintenance, and if we want to improve maintenance, we should facilitate the process of comprehending existing programs. Rugaber of Georgia Institute of Technology identified the following gaps we should bridge to comprehend a program [46]:

- Application domain/program domain — The gap between a problem from some application domain and a solution in some programming language.
- Concrete/abstract — The gap between the concrete world of physical machines and computer programs and the abstract world of high level descriptions.
- Coherency/disintegration — The gap between the desired coherent and highly structured description of the system and the actual system whose structure may have disintegrated over time.
- Hierarchical/associational — The gap between the hierarchical world of programs and the associational nature of human cognition.
- Bottom-up/top-down — The gap between the bottom-up analysis of the source code and the top-down synthesis of the description of the application.

There exist two approaches to understanding a program while bridging those gaps: bottom-up, starting with the source code and generating a description; and top-down, formulating hypotheses and confirming them by examining the program.

An example of the former is the approach taken by Soloway and Ehrlich. They propose a bottom-up model of analysis based on the recognition of *plans* in the source code [47].

The top-down approach is championed by Ruven Brooks. In his approach , the program understander attempts to create a series of mappings between the application domain and the program. Exploration is driven by expectations derived from the application description with the aid of "beacons" [48].

The following guidelines are suggested to support the top-down approach of Brooks. That is, applying them can help users find "beacons" more easily.

Cohesion

Cohesion is the degree to which the statements in a component from a coherent whole. The most coherent components do just one thing, whether it be manipulating an object or performing a function. [1].

According to Stevens [49], there are six layers of cohesion. They are listed below from lowest to highest:

1. Coincidental Cohesion: The module does tasks[2] that are related loosely or not at all.
2. Logical Cohesion: The tasks are related in some logical way.
3. Temporal Cohesion: The tasks are related in some way and must be done in the same time span.

[2]In this context, the term, task means a thing or a piece of work rather than a program unit of Ada language that executes concurrently with other program units.

4. Communicational Cohesion: All processing elements of a task refer to the same set of input or output data.
5. Sequential Cohesion: Output data from one element of the module is input for the next element.
6. Functional Cohesion: All elements of a module are related to performing a single function.

In this scheme, low levels of cohesion should be avoided as much as possible. Middle levels of cohesion are about as good as high levels. In practice, it is not necessary to improve the cohesion of a component once it is in the middle range.

Another way to measure cohesion is found in Embley and Woodfield's paper [50]. They classified the degree of cohesion in a module into four kinds — separable, multifaceted, non-delegation, and concealed. These are defined as follows for abstract data types (ADTs), but the ideas can be generalised to all reusable components. Details of the four kinds follow:

1. Separable Strength: An ADT part has separable strength if the part exports an operator (function or procedure) that does not use a domain of the ADT it exports; or the part has a logically exported domain of the ADT that no operator of the part uses; or the part has two or more logically exported domains whose operators do not share any of the domains of the ADT.
2. Multifaceted: An ADT part has multifaceted strength if it does not have separable strength, and it exports two or more domains of the ADT. Because it is not separable some operator must share two or more exported domains.
3. Non-delegation: An ADT part has non-delegation strength if it has neither separable nor multifaceted strength, and it has an operator that can be delegated to a more primitive ADT.
4. Concealed: An ADT part has concealed strength if it has neither separable, multifaceted, nor non-delegation strength and it has a logically hidden ADT.

C01: Make cohesion high within each component.

Although not essential for reuse, cohesion is a desirable attribute, because components with high cohesion are likely to be easier to understand and more tailorable, since related code will tend to concentrate in one place.

Commenting

The author believes that zero-commenting is the best. That means that if source code itself is perfectly self-documenting, then any comments or documents would not be necessary to understand the code. However, that goal is not accomplishable; thus a suitable commenting scheme is needed in real programming practice. Below, a general guideline C02 and 6 specific guidelines C03–C08 are given.

C02: Make each comment adequate, concise and precise.

C03: Put a file header on each source file.

C04: Put a header on the specification of each program unit.

C05: Place information required by the maintainer of the program unit in the body header.

C06: Comment on all data types, objects, and exceptions unless their names are self-explanatory.

C07: Minimise comments embedded among statements.

C08: Use pagination markers to mark program unit boundaries.

Comments of suitable quantity and good quality will obviously make the component more readable and thus easier to tailor [43]. As one thing notable, comments often fail to change in accordance with the change of the source code which contains the comments. Thus it is important to make source code as self-documenting as possible. And , if ever comments are needed, they must be made in a concise manner to facilitate users' understanding.

In the above 6 guidelines, C03–C08, it is recommended the use of *file header*, *program unit specification header*, *program unit body header*, *data comment*, minimal *statement comment* and *marker comments.*

Identifier Qualification

Ada's "use" clause permits us to utilise package identifiers such as New_Line without qualification. The advantage to using a use clause is that we do not have to type so many characteristics when entering Ada program. As Ada programs are often longer than BASIC, C, and Pascal programs, this is attractive to programmers familiar with these languages.

The advantage of avoiding "use" clause and qualifying all references to package identifiers is the additional documentation provided. Ada was designed for writing large programs. In a program that contains hundreds of thousands of statements and hundreds of packages, the information supplied by qualification is invaluable [33].

C09: Minimise the use of "use" clauses [42].

Two examples containing each unqualified identifiers and fully qualified identifiers are given below.

```
-- A program containing an unqualified identifier
Put (Item => Part);
```

```
--A program containing a fully qualified identifier
Integer_IO.Put (Item => Part);
```

In the first example, it is impossible to determine whether the identifier Part is type Integer, Float, or String from this unqualified call to procedure Put. But, in the second one, it is immediately clear that Part is type Integer.

Information Hiding

The rationale comes from the good software engineering practice of minimising the amount of information visible to the outside world. The principle of *information hiding* [51] suggests that modules be "characterised by design decisions that (each) hides from all others". In other words, modules should be specified and designed so that information (procedures and data) contained within a module are inaccessible to other modules that have no need for such information.

The use of information hiding as a design criterion can help produce comprehensible codes, for only needed codes are accessible to users.

In Ada, this concept can be easily implemented by using *package* concept and two kinds of *private types.*

The following five guidelines were made to promote information hiding principle in Ada.

C10: Only place in the specification section those declarations that must be seen externally.

C11: Only "with" those compilation units that are really needed.

In the Ada language, users only can access specifications of packages which they are using. Thus, only if the specification needs such visibility, the context clause should appear in the specification; otherwise it should appear in the body. And including unnecessary context clauses could make understanding the code more difficult [18]. As an note on automation, a tool could be written to catch unneeded "withs".

C12: Use private and limited private types to promote information hiding.

C13: Try to use limited private types.

Guidelines C12 and C13 were established on the basis of "Lovelace", an online Ada95 tutorial[3].

When declaring a type in a package declaration, we can declare the type as private, and then complete the definition in a section of the package declaration in a section called the "private part". If a type is declared as private, other packages can only use the operations that we provide and the default assignment (:=) and equality (=) operations. Let's suppose that we want to create a type called "Key", which uniquely identifies some resource; we only want people to be able to request a key and determine if one key was requested before another (let's call that operation "<"). Here's one way to implement this (this example is from the Ada LRM section 7.3.1):

[3]Lovelace Ada Tutor is situated at "http://lglwww.epfl.ch/Ada/Tutorials/Lovelace/lovelace.html". Like other http addresses, this is correct at the time of writing this thesis.

```
package Key_Manager is
  type Key is private;
  Null_Key is constant Key;  -- a deferred constant.
  procedure Get_Key(K : out Key);  -- Get a new Key value.
  function "<"(X, Y : Key) return Boolean; -- True if X requested before Y
private
  Max_Key : constant := 2 ** 16 - 1;
  type Key is 0 .. Max_Key;
  Null : constant Key := 0;
end Key_Manager;
```

In the above example, the type declaration in the package declaration is declared as "private". This is later followed by the word "private" introducing the "private part" of the package specification. Here the type can be defined, as well as any constants necessary to complete its definition. Although "Key" is actually a numeric type, other packages cannot use addition, multiplication, and other numeric operations because Key is declared as "private" — the only operations are those defined in the package (and := and =).

If we do not want the default assignment (:=) and equals-to (=) operations, we should declare the type to be "limited private". This means that not even assignment and equals-to operations are automatically defined. It is done by changing one sentence in the above example as follows:

```
type Key is limited private;
```

A limited private generic formal type prevents the generic unit from making any assumptions about the structure of objects of the type or about operations defined for such objects. But a non-limited private type generic formal type allows the assumptions that assignment and equality comparisons are defined for the type. Therefore, to be reusable in as many contexts as possible, limited private types should be used [42, 18].

C14: Use mode "in out" rather than "out".

In two situations, it is advised not to use the mode "out".

The first situation is where the parameters are of an imported limited type for parameters of a generic formal subprogram. According to the Ada Language Reference Manual [34, Section 7.4.4(4)], Ada allows an out mode parameter of a limited private type on a subprogram only when the subprogram is declared in the visible part of the package that declares the private type. On the other hand, there is no such restriction in parameters of mode in out. For instance, suppose we define a generic with a limited generic formal type and a generic formal subprogram with an out parameter of that type. Then, a potential user who wants to instantiate the generic with a limited type defined in another package would not be able to write a program to pass as the generic actual.

The second reason why we should avoid using the mode was explained by Sommerville et al [44]. According to their guidelines, the parameter passing mode should always be **in** (for read-only parameters) or **in out**. If initial values to parameters are assigned in a procedure, these initial values usually reflect the environment for which the procedure was originally defined. This is not advisable as far as reuse is concerned. A further reason for avoiding the **out** mode of parameter passing is that its semantics are nor well-defined.

## Nesting

Nesting can make changing a program more difficult owing to the limitations of human intelligence and perception. There are four kinds of situations where nesting happens. They are when we use "if" construct, "while" construct, "for" construct and, finally, "procedures". Below are a general guideline and a specific guideline given.

C15: Do not nest expressions, control structures or procedures to an excessive degree unnecessarily [43, 33].

In the case of "if" construct, "while" construct and "for" construct, SPC guidelines recommend not to nest the constructs beyond a nesting level of five. Meanwhile, as for the nesting within procedures, strict nesting, i.e. zero level of nesting, makes a procedure completely self-contained and thus easy to reuse in other programs. It can be compromised between strict nesting and straight-line declarations of procedures by nesting those procedures that are called by only one procedure, and globally declaring those that are widely used. The documentation for procedures that make nonlocal calls should note which procedures are needed for proper execution [33].

C16: Use "elsif" for nested "if" statements.

This reduces the nesting levels of the if statements, giving the code as clean, uncluttered appearance. It also emphasises the equal status of each if statement [52, page50]. Below are two examples about this guideline given.

```
-- A example to be avoided
if Order = Left then
   Turn_Left;
else
   if Order = Right then
      Turn_Right;
   else
      if Order = Back then
         Turn_Back;
      end if;
   end if;
end if;
```

```
--------------------------------------------------------------------

-- An example to be preferred
if Order = Left then
   Turn_Left;
elsif Order = Right then
   Turn_Right;
elsif Order = Back then
   Turn_Back;
end if;
```

Overloading

In Ada, the same variable name in different declarative regions can represent different locations in memory. The visibility of homographs is determined by name precedence [32].

In addition to it, Ada provides a way to use the same identifier name for different subprograms even if they are declared in the same region. This is known as *overloading* of subprograms. Unlike in the above case, overloaded procedures are distinguished with the aid of both the number of parameters and the types of the parameters [33].

Finally, Ada also allows the overloading of the predefined operators such as "+" or "-".

A guideline is suggested on the basis of the above things.

C17: Do not overload names from package "Standard" [53].

Rymer argues that Ada names predefined in package Standard should not be *redefined* or *overloaded* [53]. That is because this keeps the reader from confusing the overloaded names with the names predefined in package "Standard".

Self-descriptiveness

As explained in guidelines on commenting in detail, it is important to make code as self-documenting, in other words, self-descriptive as possible.

The following 9 guidelines are recommended to increase code readability.

C18: Make reserved words and other elements of the program visually distinct from each other.

This guideline is instantiated as follows, although slightly different style can be adopted for each organisations [43]:

- Use lower case for all reserved words (when used as reserved words).

- Use mixed case for all other identifiers, a capital letter beginning every word separated by underscores.
- Use upper case for abbreviations and acronyms.

These kinds of style were also used in the Ada 95 Reference Manual [54].

C19: Use descriptive identifier names.

C20: Do not use any abbreviations in identifier or unit names.

Using descriptive identifier names is needed for the sake of programmer himself and other users, for it promotes readability and self-documentation. Names should be as long as necessary to provide the needed information and to promote readability. They should be considered part of the documentation of the component.

Almost every set of guidelines mandates users not to use any abbreviations in identifiers. Even if some abbreviations are well defined and known in a certain domain, they could cause difficulties in understanding them. That is because reuse very often happens across domains other than where the software components were built, thus the meanings of abbreviations can be ambiguous to other users.

C21: Use names which indicate the behavioural characteristics of the reusable part, as well as its abstraction.

If this general guideline is applied to names for procedures or functions, it can be instantiated as follows [43]:

- Use action verbs for procedures and entries.
- Use predicate-clauses for boolean functions.
- Use nouns for nonboolean functions.

Some examples are given below.

```
-- Sample procedure names
procedure Get_Next_Token
procedure Create

-- Sample function names for boolean-valued functions
function Is_Last_Item
function Is_Empty

-- Sample function names for nonboolean-valued functions
```

```
function Successor
function Length
function Top
```

C22: Do not hard code array index designations [41].

Instead of hard coding array index designations as below, types or subtypes should be used.

```
type Table is array (1..50) of Element_Type;
```

The reason is that the additional declaration will make the code more self-documenting and thus more tailorable [41]. The upper or lower bound may be an index that will change at some time. The subtype or type declaration will allow the change to be made once instead of many times throughout the program.

C23: Use named constants for parameter defaults [38].

Using named constants as parameter defaults would help the reader to better understand the code as in the following examples:

*Example*

```
procedure Read (Value :    out Element_Type;
                Group : in Tag_Group_Type := Default_Group);
```

is easier to understand than this.

```
procedure Read (Value :    out Element_Type;
                Group : in     Tag_Group_Type := 0);
```

C24: Use named parameters association.

As in the case of Guideline C23, the added documentation from using named parameters association would make code more understandable.

C25: Use descriptive named constants as return values.

Layout

*Layout* of code can be achieved through suitable vertical and horizontal spacing. It could help perceive the semantics as well as the syntax of a certain code segment.

C26: Code program in a well-arranged manner horizontally and vertically.

Perhaps, current sophisticated pretty-printers can do this for programmers.

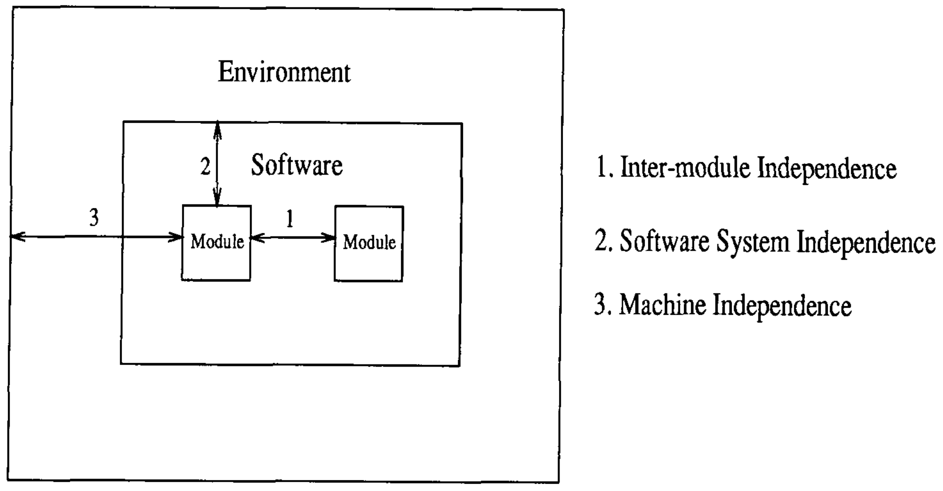


Figure 3.3: Three Kinds of Independence

### 3.5.3 Principle of Independence

*Independence* of a component means the degree to which a component is related to other components or environment surrounding it, either it is hardware or software environment. Thus, *independence* has been represented as terms such as *coupling* or *portability*. Here, the term *independence* is considered as a concept concept than *coupling*, *portability*, or *rehostability*.

There exist three kinds of *independence* as described in figure 3.3 [41]:

- inter-module independence
- software system independence
- machine independence

Below are guidelines given to improve the three kinds of independence.

Coupling

*Inter-module independence* mentioned above has also been known as *coupling* [1]. Pressman defines *coupling* as a measure of interconnection among modules in a software structure. In other words, coupling measures how much modules depend on one another. It depends on the interfaces between modules, the data that pass between them, and the control relationships.

I01: Make coupling low.

Coupling should be as low or loose as possible. This helps make dependencies both clear and isolated, thus making components easier to reuse.

According to Stevens [49], there are several levels of coupling, listed below from lowest to highest.

1. No Coupling: The modules are independent and do not communicate.
2. Data Coupling: Communication is limited to passing simple arguments.
3. Stamp Coupling: A variation of data coupling, where part of a data structure is passed, rather than simple arguments.
4. Control Coupling: Data of a control nature are passed. An example is the passing of a control flag.
5. External Coupling: Modules are tied to specific external environments. For some modules this may be unavoidable, but environment dependence should be isolated as much as possible.
6. Common Coupling: Modules share data in a global data area.
7. Content Coupling: One module uses the data within the boundary of another module.

In this scheme, coupling should be as low as possible, both for components and for modules making up components. For some modules it may not be possible to achieve the lower levels of coupling (no coupling, data coupling). An effort should be made, however, to build modules with coupling as low as possible in the above scale.

Another way of to measure coupling comes from Embley and Woodfield's research [50]. In this scheme two compilation units are *visibly coupled* if one directly accesses the data structures of the other. They are *surreptitiously coupled* if one uses undocumented information about the other's data structures. Finally, they are *loosely coupled* if they are neither visibly nor surreptitiously coupled. In this scheme, the goal is to make components loosely coupled.

I02: Minimise "with" clauses on reusable parts, especially on their specifications [42].

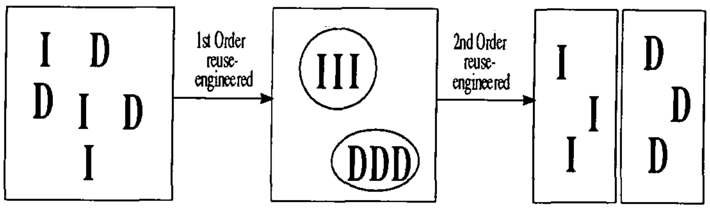
I03: Use generic parameters instead of "with" statements to reduce the number of context clauses on a reusable part, and to import portions of a package rather than the entire package [41].

It can assumed that the more "with" clauses a component has, the more difficult it would be reused in the future, since the component can not be reused without the components on which it depend.

## Machine Independence

*Machine Independence* can be used in the nearly same meaning as *retargetability* or *rehostability*, and, perhaps, *portability*. Code should be written to ignore details of underlying implementations. And components should be designed without reference to the surrounding environment. *Contact* between a component and its environment should occur through explicit parameters and explicitly invoked subprograms [55, page7].

The following 3 guidelines are concerned with the relationships between a component and its hardware environments with which it interacts.



I : Machine-Independent Features

D : Machine-Dependent Features

Figure 3.4: A method to improve independence of components

I04: Machine-dependent and low-level Ada features should be avoided except when absolutely necessary [38, page 187].

I05: Encapsulate input/output (I/O) uses into a separate I/O package [42, 41].

I06: Minimise the use of implementation dependent I/O procedures [56].

In Ada, heavy emphasis is given to the use of packages to increase the independence of a program by encapsulating machine dependencies.

SPC suggested the following 3 steps in order to increase machine independence of components [43]:

1. Don't use machine-dependent or environment-dependent features.
2. If 1) is impossible, isolate those features.
3. If 2) is also impossible, document them well for future users.

As depicted in figure 3.4 drawn by the author due to SPC guidelines, maintaining the source code at the left side case will be definitely more expensive than ones at the right side.

Software System Independence

*Software system independence* indicates the relationships in more inner areas rather than machine level.

I07: Use the predefined packages for string handling.

The predefined Ada language environment includes string handling packages to encourage portability. They support different categories of strings: fixed length, bounded length, and unbounded length. Subprograms for string construction, concatenation, copying, selection, ordering, searching, pattern matching, and string transformation. Thus, we no longer need to define our own string handling packages [43].

I08: Avoid predefined and implementation defined types [41].

Predefined types such as *integer* or *float* are not likely to be portable because their form, i.e. range and precision, can vary from Ada implementation to Ada implementation [40]. Therefore, predefined defined types should be avoided except for the case of *string.*

I09: Explicitly specify the precision required.

Each floating point or fixed point type should explicitly specify the precision, using the "Delta" or "Digits" accuracy definition. This will make clear any assumptions made about accuracy of calculations [41].

I10: Use "attributes" instead of explicit constraints.

An example from Nissen and Wallis' book well explains the reason why this guideline should be followed [38].

```
A: array (Discrete_Type) of F;
   ...
for I in Discrete_Type loop
   exit when A(I) < Sum * F'Epsilon;
   Sum := Sum + A(I);
end loop;
```

This example assumes that the series A(1) + A(2) + A(3) + ... converges when all terms are positive. Because the loop depends on F's model numbers and not on explicit constraints, all Ada implementations should have the same accuracy.

I11: Use explicitly declared types for integer ranges in the loop statement.

If no type name is specified, then *Integer* is used as the default, which can result in a discrete range being invalid under some Ada implementations. By using type designations, the logic can be more independent of the data [41, 43].

```
-- An example to be avoided
for I in 1..Max_Num_Apples loop
   ...
end loop;

-- An example to be preferred
type Apple_count_Type is range 1..Max_Num_Apples;
for I in Apple_Count_type loop
   ...
end loop;
```

I12: Avoid optional language features.

To make components portable, it is advised not to use optional language features and Ada implementation dependencies, where this cannot be done, they should be isolated, so users can plug in new versions easily [42]. For example, using "Unchecked_Deallocation" and "Unchecked_Conversion" can cause portability problem in the future reuse. That is because these two procedures are optional and implementation dependent. If ever they must be used, their use should be documented.

I13: Avoid using pragmas [41].

Pragmas are generally environment dependent. Therefore, their use is not recommended. However, sometimes, their use may be unavoidable. For instance, pragma "Interface" may be needed to specify interfaces with subprograms of other languages. Pragma "Elaborate" may be needed to insure that a program is correctly elaborated no matter what compiler is used, since elaboration order varies from compiler to compiler [43].

As in the above two instances, if pragmas are used, they must be isolated and thoroughly documented.

I14: Close files before a program completes.

Different Ada implementations handle unclosed files in different ways. The state of unclosed files after program termination is undefined. Therefore, to increase the independence of a component, it is recommended to close all files before a subprogram terminates normally or abnormally [41].

I15: Do not input or output "access" types.

The effect of I/O of access types is undefined. If used, it may lead to components that are not portable. To output an object pointed to, output the object. To output the address of an object pointed to, the address of the object using "System.Address" should be output [56].

### 3.5.4 Principle of Robustness

High *Robustness* of a component means that the component is of the highest possible quality. In other words, it is correct and reliable [43]. An error or weakness in a reusable part may have very expensive consequences, and it is important that other programmers can have a high degree of confidence in any parts offered for reuse. That is to say, robustness of components are closely related with the cultural obstacles to a successful reuse adoption which are represented as the term, "NIH syndrome".

Error Tolerance

"Error tolerance" has also been known as "defensive programming" [38]. Because Ada has traditionally been used in real-time, embedded systems where a small fault could cause a catastrophic results, it is important not only to prevent faults, but also to handle them, if even happen.

Ada provides facilities to deal with these real problems which make handling them much easier than in other programming languages. In Ada, an exception represents a kind of exceptional situation, usually a serious error. At run-time an exception can be raised, which calls attention to the fact that an exceptional situation has occurred.

The following guidelines are believed to improve error tolerance and hence software robustness.

R01: Put a modest amount of "exception" and "raise" statements in code.

Exception declarations and handlings should be used in the situations where they are definitely needed. Unnecessary "exception" and "raise" statements could do more harm than good to the principle of defensive programming.

Good exception handling is important to software reuse for several reasons [33]:

- Components with good error exception handling have safety built in.
- Errors are isolated and well documented.
- The way interfaces work is made clear. There are fewer hidden assumptions. Thus it is safer than not so.
- The users have the freedom to decide whether to propagate exceptions further, to retry the operation that raised the exception, to abandon the operation, or to continue regardless.
- Good exception handling makes components more tailorable and thus more reusable.

R02: Propagate exceptions out of reusable parts.

The rationale behind this guideline is as follows. An exception is raised because an undesired event has occurred. Such events often need to be dealt with entirely differently with different uses of a particular software segment. Also, it is very difficult to anticipate all the ways that users of the part may wish to have the exceptions handled. Passing the exception out of the part is the safest and best treatment [43].

R03: Never use the "when others" construct with the "null" statement [41].

Use of the null statement suggests that the exception is not used for an abnormal condition. Below is a typical of it shown.

```
begin
   loop
      ...
      raise Miscellaneous_Error;
      ...
      end;
   exception
      when others =>
         null;
      end;
      -- rest of normal program code
```

In the above example "raise" statement is used to exit the loop and to continue executing normal control flow. This implies that there never was an abnormal condition and thus it was unnecessary.

R04: Avoid pragma "suppress".

The Ada Language reference Manual [34, Section 11.7] does not require that pragma suppress be implemented. Program suppress does not guarantee that exceptions will not be propagated to a unit for which exception suppression is in effect. The execution of a program is erroneous if an exception occurs while pragma suppress is in effect.

R05: Do not propagate an exception beyond where its name is visible [40].

An exception should not be propagated beyond where its name is visible. Otherwise, it can only be handled by a "when others" handler.

R06: Do not propagate predefined exceptions without renaming them.

Predefined exceptions have no corresponding "raise" statement in the source code, so it is not always obvious that an exception can be propagated. Predefined exceptions can be raised by many operations, making them difficult to locate. Renaming predefined expressions makes it easier to pinpoint the exact cause of each exception [44, 41].

R07: Do not execute normal control statements from an exception.

Ada's exception handling should be only used for abnormal control flow, not for normal control [41]. For instance, the following code contains an unnecessary exception statement.

```
begin
   loop
      Text_IO.Get(Data_File, Data_Value);
      ...
   end loop;
exception
   when Text_IO.End_Error(Data_File) =>
      ...
end;
```

This can be changed as below, where the exception statement was substituted for a normal control statement "while".

```
while not Text_IO.End_Of_File(Data_File) loop;
   Text_IO.Get(Data_File, Data_Value) =>
   ...
end loop;
```

R08: Use range constraints on numeric types.

This causes the compiler to issue a message if the range cannot be supported. The range constraints should be meaningful to the application [52].

R09: Explicitly declare a type to use in defining discrete ranges [55][page 28].

Use explicitly declared types for discrete ranges. That is, use

```
type Discrete_Range is range 1..Table_Size;
type Table is array (Discrete_Range) of Element_Type;
```

instead of

```
type Table is array (1..Table_size) of element_Type;
```

This provides several benefits. There will be fewer logic errors when components are tailored, because the compiler will have already caught them when it checked for type inconsistencies. Also, the code will be more portable, since the compiler can select the best internal representation for the numeric type requested by the range declaration.

Unfortunately, using explicitly declared types for integer discrete ranges does not always lead to easy to read code. Type conversions may be needed to convert among explicitly declared types. The combination of long type names and required type conversions results in long multi-line Ada statements that are hard to read. Nevertheless, the advantages of using explicitly declared types for integer discrete ranges outweigh the disadvantages.

R10: Avoid using the "when others" clause of the "case" statement as a shorthand notation.

The *when others* clause of the case statement should not be used as a shorthand to handle all cases that have not been listed. Instead, each case should be explicitly handled and the *when others* clause must be omitted. If the component is later modified to add more values to the data type, this will call attention to the fact that the new values are not handled in the case statement. If the *when others* clause was used, the new data values would be handled by this clause and the operation on the data might be incorrect [42, 44].

## 3.6 Summary

In this chapter, firstly, the history of Ada and Ada's major features were reviewed. Through it, the specific situations where Ada is most suitable were identified, as well as the language's strong points and weak points especially in terms of *code reuse.*

After that, existing Ada code reuse guidelines were reviewed in section 3.2. Ways to improve their usability were investigated. On the basis of these considerations, guidelines were suggested on the basis of the above things to maximise the Ada's strong points while complementing and minimising its weak ones. The guidelines appearing in this thesis were grouped in accordance with the constituent properties, i.e. *adaptability*, *comprehensibility*, *independence* and *robustness*, which are believed to contribute towards *code reusability.*

# Chapter 4

# Reuse Metrics: Metrics regarding Software Reuse

"When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science". — Lord Kelvin

Software Metrics are necessary to know the properties of the software we are developing and predict the needed effort and development period. Moreover, they are needed when software is maintained for various reasons which allow us to classify maintenance into four kinds, i.e *corrective maintenance*, *adaptive maintenance*, *perfective maintenance* and *preventive maintenance* [57].

In this chapter, first of all, the general background of software metrics is studied. The background studies cover the objectives of software metrics, measurement theory applied to software engineering and five types of scales relevant to software product and process measurement. In addition to these studies, desirable characteristics of software metrics are defined.

Finally, software metrics especially relevant to reuse are investigated.

## 4.1 Introduction

The history of measurement is as old as human history. Among those measuring units, some such as *foot* still exist until now. It is believed that one of the most important concepts in engineering discipline is measurement [58], as is *reuse*. An engineer needs to know why to make measurements, what can be measured, how to measure, and what to do with the results.

As an engineering discipline, software engineering has faced the following questions while building or maintaining software.

> How good is a program? How reliable will a software system be once it is installed? How much more testing should I do? How many more bugs can I expect to find? How much will the testing cost? How difficult will it be to maintain a system? How much will it cost to build a new system similar to one we built five years ago? How long will it take?

In addition to the above things, especially in the view of software reuse, good measurement skills are crucial to promote it in software industry communities.

## 4.2 Software Metrics

Confusion in using terms such as *metrics* and *measurement* proves that the area is still a young discipline, and has been neglected by computer scientists.

Lorenz defines the terms as follows [59]. *Metrics* is a standard of measurement used to judge the attributes of something being measured, such as quality or complexity, in an objective manner. On the other hand, *Measurement* is the determination of the value of a metric for a particular object. Therefore, considered with those definitions, the term, *measurement* should be used, when mentioned about the activity itself to measure something. However, since the term, metrics is generally accepted and used in the discipline of software engineering, the distinction between two terms is not strictly made in this thesis.

### 4.2.1 Objectives of Reuse Metrics

Ford [58] suggested four reasons why engineers measure. For the sake of software, they can be said as follows:

- To understand the current state which software community is facing

  Every software measurement describes an aspect of the current situation of software communities and helps us discover patterns and trends. Thus valuable theories or laws on software could be drawn by the results of measurements. Also, additional measurements can be used to support or refute them, thereby leading to a better explanation for the current situation.

- To state software requirements quantitatively and demonstrate compliance

It is almost impossible to imagine an engineering project without quantitative requirements. For example, suppose a civil engineer is designing a highway bridge over a river. He should be concerned with the length of the bridge, the maximum traffic load, the height and flow of the river at the flood stage, the maximum wind load the bridge must withstand etc., which are likely to be expressed quantitatively. Otherwise, the bridge might collapse in the future.

Software engineers also have to work with quantitative requirements. As a result of these, engineers need to be able to demonstrate compliance with such requirements.

- To track progress and predict results of a software project

  In a large software project, periodic measurement of what has been accomplished or complete allows the project manager to track progress quantitatively. That kind of software measures can be specially useful in the identification of unusual trends, so the manager can foresee problems and try to solve them before they get out of hand. This can resolve not only technical problems, but also schedule or cost overruns. For instance, software engineers use defect counts during testing to calibrate reliability models, which in turn can predict when system testing will be complete and the desired level of system reliability achieved.

- To analyse costs and benefits

  In the real world, we are almost always not allowed to chase two rabbits. For example, if we want to get a good mark in an examination paper, we might have to sacrifice some entertaining times for study. Likewise, similar trade-offs happen when software is built. There are almost always many ways to design software products and many ways to design the components and subcomponents of those products. Each design offers advantages and disadvantages, and the software engineers must trade one thing against another. The classic trade-off in computer programming is *time* vs. *space*. The two aspects very often conflict. Therefore, if quantitative data on the costs and benefits are provided to software engineers, they would be able to make better decisions, as a result, leading to reduced costs and increased costs.

### 4.2.2 Measurement Theory in Software Engineering

Although originally *measurement theory* is from mathematics [58], for software engineering to become a real engineering discipline, it is needed for those principles to be adopted in a form of *software engineering measurement.*

Informally, as mentioned earlier, we can think of a measure as a way of associating a number, representing some attribute, with a physical object. Such association is usually called a mapping or a function in mathematical terms. Formally the association is defined as the following six formula [58].

Definition 1

Let $A$ be a set of physical or empirical objects. Let $B$ be a set of formal objects, such as numbers. A *measure* $\mu$ is defined to be a one-to-one mapping $\mu : A \rightarrow B$.

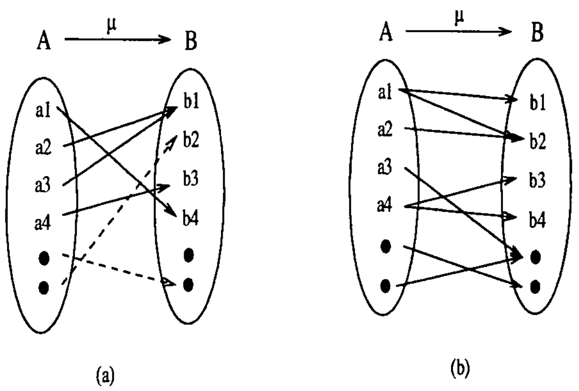


Figure 4.1: One-to-one mapping and one-to-many mapping

The requirement that the measure be a one-to-one mapping guarantees that every object has a measure, and every object has only one measure. It does not require that every number in set $B$ be the measure of some object in set $A$. As shown in figure 4.1, in the case of (a), since each measure is unique, we can use them to expect the benefits described above. On the other hand, if our measures belong to the second case, they would cause confusion to not only software engineers but also managers. Thus, we must try to get measures desirable like in the first case.

## Definition 2

Let $A$ be a set of objects, let $R$ be the set of real numbers, and let $m : A \rightarrow R$ be a measure. Then $m$ is a *metric* if and only if it satisfies these three properties:

$$m(x,y) = 0 \text{ for } x = y$$
$$m(x,y) = m(y,x) \text{ for all } x,y$$
$$m(x,z) \leq m(x,y) + m(y,z) \text{ for all } x,y,z$$

The above definition clearly shows that metric has a smaller scope than measure, and that our usage of the term, metrics is incorrect. Therefore "software measure" is a more precise term than "software metric".

This definition on measure is extensible to other sets, as long as the set includes zero and the addition and less-than-or-equal operations are defined on the set.

## Definition 3

A *relational system* is defined as an ordered tuple $(S, rel_1, ..., rel_n, op_1, ..., op_m)$, where:

$S$ is a nonempty set of objects;

$rel_1, ..., rel_n$ are $k_i$-ary relations on objects in $S$ (this means that the relation $rel_i$ defines a relationship among $k_i$ objects);

$op_1, ..., op_m$ are binary operations on objects in $S$ (this means that each operation operates on exactly two objects, producing a third object in $S$).

In the case of software engineering, we can draw an example of a relational system according to the above definition. The following relational system can be defined.

A *relational system* is defined as an ordered tuple $(S, rel_1, op_1, ..., op_m)$, where:

$S$ is a set of software components;

A binary relation "bigger than or smaller than or same size as", $rel_1$ exists.

Four binary operations exist. They are $+$,$-$,$\times$, and $\div$.

## Definition 4

Let $A = (S_A, relA_1, ..., relA_n, opA_1, ..., opA_m)$ be a relational system of physical or empirical objects, and let $B = (S_B, relB_1, ..., relB_n, opB_1, ..., opBm)$ be a relational system of formal objects(such as numbers). Let $\mu : S_A \rightarrow S_B$ be a measure. Then the triple $(A, B, \mu)$ is a *scale* if and only if $relA_i(a_{i_1}, ..., a_{i_k}) \Leftrightarrow relB_i(\mu(a_{i_1}), ..., \mu(a_{i_k}))$

and $\mu(a\ opA_j\ b) = \mu(a) opB_j \mu(b)$

for all values of $i$ and $j$, and for all $a, b, a_{i_1}, ..., a_{i_k} \in S_A$.

More informally, this definition says two things. First, every relation defined on the physical objects, there is a equivalent relation defined on the measures of those objects. By *equivalent*, we mean that if a statement about a relationship betwwn or among objects is true, then the corresponding relationship between or among their measures is also true. Second, for every operation defined on the physical objects, there is a corresponding operation defined on the measures, such that the result of measuring the combined objects is the same as performing the corresponding operation on the measures of the individual objects.

## Definition 5

Let $(A, B, \mu)$ be a scale, where the set of objects in $B$ in the set of real numbers. Let the notation $\mu(A)$ mean the set of all real numbers that are measures of some object in $A$. (In mathematics, we call this *range* of $\mu$.) Then a mapping $t : \mu(A) \rightarrow B$ is defined to be an *admissible transformation* if and only if the triple $(A, B, t \circ \mu)$ is a scale.

This definition can be interpreted as saying that if we have one scale of measure for a certain kind of object, we can invent other, equally good scales by applying admissible transformations to the original scale.

## Definition 6

Let $(A, B, \mu)$ be a scale, where the set of objects in $B$ is the set of real numbers. A statement about the measures $\mu(a)$ of objects in $A$ is meaningful if and only if the truth value space (whether it is true or false) of that statement is unchanged after applying any admissible transformation to $\mu$.

As two important characteristics of engineering measurement, *precision* and *repeatability* are considered. The former one means how much the measures collected approach to the correct values. The latter one indicates that the same results are achieved whenever measurements are carried out.

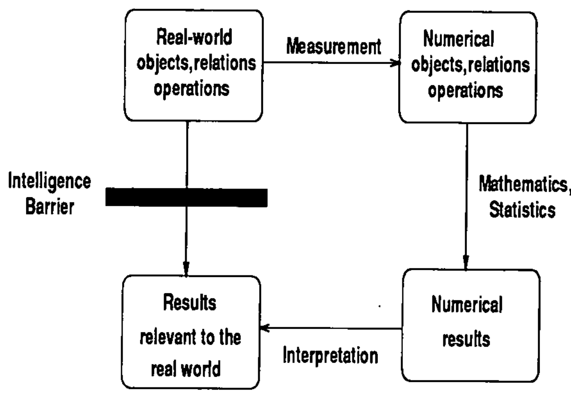

Real-world objects,relations operations
Measurement
Numerical objects,relations operations
Intelligence Barrier
Mathematics, Statistics
Results relevant to the real world
Interpretation
Numerical results

Figure 4.2: Measurement and the intelligent barrier

In engineering disciplines such as applied physics, chemical engineering or civil engineering, repositories of templates have been established over the centuries. So scientists and engineers involved in those disciplines actively use measurements to propose theories and validate them. But relatively new disciplines such as psychology or software engineering still have a long way to go before they reach a similar level of theory and practice. This kind of phenomena is well described as an intelligent barrier. This barrier is shown in the left side of figure 4.2.

Finally, relevant to measurement theory, five kinds of scales exist. Depending on each kind, admissible transformations and operations on them are applied accordingly. Detailed explanations are followed below.

Let $(A, B, \mu)$ be a scale, where $B$ is the set of real numbers, and transformations $t$.

## Nominal scales

These scales simply give numeric "names" to objects. Thus, any numbering is as good as any other, so any one-to-one function $t$ is an admissible transformation. As an example of nominal data, one can measure the type of program being produced by placing it into a category of some kind — database program, operating system, etc. For such data, we cannot perform arithmetic operations of any type or even rank the possible values in any "natural order". The only possible operation is to determine whether program **A** is of the same type as program **B**. Such data are said to have a nominal scale, and the particular example given can be an important parameter in a model of the software development process [60]. The data might be considered either subjective or objective, depending upon whether the rules for classification allow equally qualified observers to arrive at different classification for a given program.

## Ordinal scales

These scales assign numbers to objects in a particular order, but any numbers that maintain that order are equally good. Any strictly increasing function $t$ is an admissible transformation. For example, programmer experience level may be measured as *low*, *medium*, or *high*. In order for this to be an objective metric, one must assume that the criteria for placement in the various categories are well defined, so that different observers always assign the same value to any given programmer.

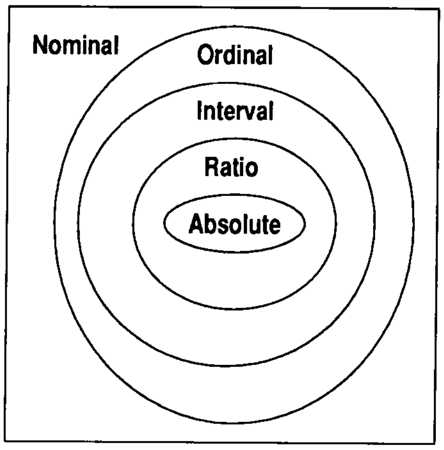


Figure 4.3: Relationships among classes of scales

Interval Scales

These scales assign numbers to objects in such a way that the interval between two measure values is meaningful throughout the range of values. Only positive linear functions $t(x) = ax + b$ are admissible transformations. A typical example of these is McCabe's complexity measure [61]. Differences appear to be meaningful; but there is no absolute zero, and ratios of values are not necessarily meaningful. For example, a program with complexity value of 6 is 4 units more complex than a program with complexity of 2, but it is probably nor meaningful to say that the first program is three times as complex as the second.

Ratio scales

These scales assign values in such a way that the ratio of two measures is meaningful. The only admissible transformations are positive linear functions of the form $t(x) = ax$. An example is program size, in lines of code(LOC). A program of 2,000 lines can reasonably be interpreted as being twice as large as a program of 1,000 lines, and programs can obviously have zero according to this measure.

Absolute scales

These scales have only one way of measuring objects, and so the only admissible transformation is the identity $t(x) = x$. For instance, the number of "with" clauses which might be needed to measure "inter-module independence" is got from counting it directly, and the value is uniquely fixed whenever measured.

It is noticed that this sequence of scales is increasingly restrictive as described in figure 4.3. And computational power and usefulness increase from nominal scale to absolute one.

### 4.2.3 Characteristics possessed by Ideal Metrics

Good metrics should facilitate the development of models that are capable of predicting process or product parameters, not just describing them. Thus, ideal metrics should be [60]:

- Simple, precisely definable—so that it is clear how the metric can be evaluated;
- Objective, to the greatest extent;
- Easily obtainable (i.e., at reasonable cost);
- Valid—the metric should measure what it is intended to measure; and
- Robust—relatively insensitive to (intuitively) insignificant changes in the process or product.

In addition, for maximum utility in analytic studies and statistical analyses, metrics should have data values that are on appropriate measurement scales [62, 63].

### 4.2.4 Reuse Metrics

To encourage use of reuse guidelines, we need *reuse metrics* to prove and validate the efficiency and benefits of using them. In terms of reuse, we can think of two kinds of metrics. One is *property metrics*, with which we can evaluate or predict the reusability of a certain component. Another group of metrics can be called *impact metrics* since we can be informed of the impact such as productivity increase or defect decrease of reusing software during development or maintenance. One main difference between the two groups of metrics is that property metrics are used during development whereas impact metrics can be collected after development of software. It can be safely said that the former can be collected in the shorter term than the latter one. Detailed explanations are given below.

#### Property Metrics

Although some metrics such as size metrics or complexity are now available, most valuable metrics for which the main purpose of software metrics is, are still unmeasurable. Software engineers have identified a number of other properties or qualities or attributes of software that seem to be desired but for which we currently have no way of measuring. Because of many of their names, these properties are often referred to as *ilities* (pronounced like "ill at ease", which describes our emotional state when asked to measure them) [58]. The most representative measures belonging to that kind are *accessibility*, *adaptability*, *comprehensibility*, *fault tolerance*, *integrity*, *interoperability*, *maintainability*, *portability*, *reusability*, *robustness*, and *testability*.

One of methods to tackle this problem is to decompose them into lower level until we can measure them. A model to decompose the property *reusability* already appeared in figure 2.1 The author of this thesis decomposes it into four lower-level properties, i.e. adaptability, comprehensibility, independence and robustness, as shown below:

$$\begin{aligned} \mathit{Reusability} &= k1 \times \mathit{Adaptability} + k2 \times \mathit{Comprehensibility} \\ &\quad + k3 \times \mathit{Independence} + k4 \times \mathit{Robustness} \end{aligned}$$

where $k1, k2, k3$ and $k4$ are propositional constants,

$0 \leq each\, Metrics \leq 1$, and

$k1 + k2 + k3 + k4 = 1$

When we think about adaptability of a component, we find most guidelines concerning the property are related to semantics, thus we can only get metrics by inspection and they are on ordinal scales. For instance we can say that one component implementing an abstract data type "stack" is complete, but another is not—Guidelines A01, A02 and A03. And it can be said that the adaptability of components increases from "abstract data object (ADO)" to "generics abstract data type (GADT)—Guidelines A08 and A09. In the above cases, however, it is totally meaningless to say that a component is twice complete as another one, or that the interval between "abstract data object (ADO)" and "abstract data type (ADT)" in terms of their adaptability is the same as the interval between "generic abstract data object (GADO)" and "generic abstract data type (GADT)".

With respect to comprehensibility, the current situation is better than in the case of adaptability. It is natural to assume that the smaller the size and the lower the complexity of components, the more comprehensible the components are as explained in the following simple equations:

$$Comprehensibility = k1 \frac{1}{Size}$$

$$Comprehensibility = k2 \frac{1}{Complexity}$$

$$Complexity = k3 \times Size$$

where $k1, k2, k3$ are propositional constants.

Fortunately, most research on software metrics has been concentrated in size and complexity until now. We have a number of useful metrics such as "lines of code (LOC)" [64, 65], "function points (FP)", McCabe's cyclomatic complexity measure [61] and Halstead's product metrics, etc.

To measure independence of a certain component, we can count the number of "withs", but like other properties, we have to depend on inspecting source code. Unfortunately the above situation also happens in the case of measuring robustness. Overall, we can say in summary that most metrics are lacking in computational power since they are on nominal scales or ordinal scales. A fortunate thing is that we have some sophisticated size and complexity metrics.

## Impact Metrics

As defined above, impact metrics are used to investigate the impact of reusing software components during software development or maintenance. An experiment conducted at the

University of Maryland [66] suggests some impact metrics also applicable to other projects. They were enhanced by the author of this thesis, and are explained below.

In terms of measuring impact of reuse, "reuse rate", "effort", "productivity" and "number of defects" are the most representative metrics. Before the four impact metrics are defined, firstly size metrics and reusability metrics must be defined.

Size

Suppose a system $S$ is a set consisting of components which are also sets consisting of source codes as their elements. Then the following can be defined.

The size of a system $S$ is function $Size(S)$ that is characterised by the properties:

Property Size 1. $Size(C) \geq 0$

Property Size 2. $Size(C_i + C_j) = Size(C_i) + Size(C_j)$ when $C_i \cap C_j = \emptyset$.

Let us assume an operator called $Components$ which, when applied to a system $S$, gives the distinct components of the system $S$ such that:

$Components(S) = C_1, ..., C_n$, such that if $C_i = C_j$ then $i = j$, where $i, j = 1, ..., n$.

The size of a system $S$ is given by the following function: $Size(S) = \sum_{C \in Components(S)} Size(C)$

where $Size(C)$ is equal to the number of lines of code in the component $C$.

Reusability

The "amount" of reused code in a system $S$ is a function $Reuse(S)$. Since it is a type of size metric, it inherits its basic properties from the properties Size 1 and 2. If we formalise them, they are as follows:

Let us assume an operator called $Reused\_Code$ which, when applied to a component $C$, gives the reused code in the component. Then,

Property Reuse 1. $Reuse(C) \geq 0$

Property Reuse 2. $Reuse(C_i + C_j) = Reuse(C_i) + Reuse(C_j)$

when $Reused\_Code(C_i) \cap Reused\_Code(C_j) = \emptyset$.

Here, we can think four kinds of reuse according to $Reuse(C)$ of each component $C$:

1. Verbatim Reuse: A component is reused without being modified
   $Reuse(C) = Size(C)$
2. Slightly modified: More than or equal to 75% of source code of a component is reused without being deleted or modified
   $Reuse(C) \geq 0.75 \times Size(C)$

3. Extensively Modified: Less than 75% of source code of a component is reused without being deleted or modified

   $Reuse(C) < 0.75 \times Size(C)$

4. New: Component $C$ is created from scratch

   $Reuse(C) = 0$

On the basis of the above assumptions, the reuse of a system $S$ is given by the following function:

$$Reuse(S) = \sum_{C \in Components(S)} Reuse(C)$$

Thus, reuse rate in a particular system is measured as follows:

$Reuse\_Rate(S) = Reuse(S)/Size(S)$

Effort

Two metrics are included in this group. One is "person-hours across development activities" during analysis, design and implementation. Another is "person-hours across errors (rework)" and this includes the number of hours spent on isolating an error and correcting it.

Productivity

Productivity during software development can be defined as shown below:

$Productivity(S) = Size(S)/DE(S)$

where $DE(S)$ means development effort and is defined the total number of hours spent on analysing, defining, implementing and repairing the system $S$.

Number of Defects

Finally, to inspect the impact of reuse on software quality, we need the number and density of defects found in each system/component. Defect density is simply defined as:

$Defect\_Density(S) = \#Defects(S)/Size(S)$

where $\#Defects(S)$ is the total number of defects detected in the system $S$ across the test phases.

An experiment conducted at the University of Maryland [66] shows us that the higher the reuse rate of a system, the higher the productivity is achieved, whereas effort and defect density decrease.

## 4.3 Summary

In this chapter, various aspects of software metrics were reviewed. Through it, the purposes of using metrics in software engineering were identified, and the reasons of the current immature state of software measurement were explained.

Having reviewed the above, reuse metrics were classified into two groups which are "property metrics" and "impact metrics". Associated with guidelines in chapter 3, attempts to extract useful metrics were performed.

As a conclusion, it was found that to increase the powerfulness and usefulness of use of metrics, efforts to find metrics which are on interval, ratio or absolute scales should be made. It is necessary to move from subjective metrics to more objective metrics to avoid confusion happening between users. Some metrics introduced in this chapter are used in chapter 5, Case Study.

# Chapter 5

# Case Study

In this chapter, experiments are carried out to validate and evaluate the proposed Ada code reuse guidelines. Through these experiments, the usefulness and limitations of the Ada reuse guidelines proposed here are shown.

## 5.1 Introduction

"Divide and conquer" strategy is well known as an approach to tackling complex tasks. "Quick sort" algorithm and top-down software development are typical examples. The strategy has also been adopted to measure various quality or characteristics of software. McCall and Boehm's approach to modelling software quality [12, pages 222–227], and Basili's GQM paradigm are on the basis of this kind of philosophy. Figure 2.1 appearing in the section 2.2 and figure 5.1 depict the approaches.

In the previous chapters, as approaches to producing software components of high reusability, Ada reuse guidelines and reuse metrics were discussed. Experiments are needed to see

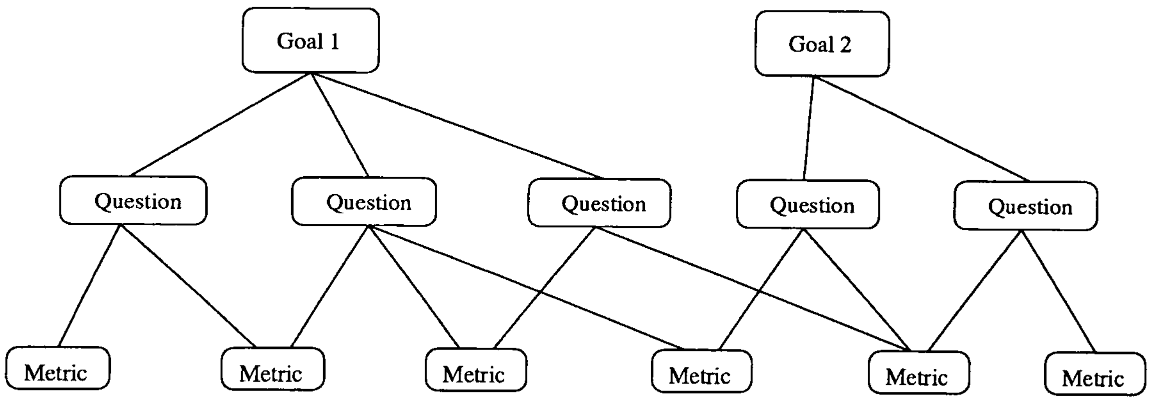


Figure 5.1: The GQM Model

whether the reuse guidelines and associated metrics are suitable.

In section 5.2, the experimental method is described, and it, first of all, discusses the experimental goals. This is followed by a description of the experimental materials and the experimental framework.

Experimental results are shown in the following section 5.3 in detail.

After that, an analysis of them is made in section 5.4.

## 5.2 Experimental Method

### 5.2.1 Experimental Goals

The main goal of the experiments is to investigate the usefulness of reuse guidelines in terms of producing software components of high reusability. As accompanying subgoals, inter-relationships between properties of software are inspected, and, finally, CASE tools necessary for automating the reuse-engineering processes are identified.

### 5.2.2 Experimental Materials

In this case study, 7 stacks were used from 3 different repositories. The table 5.1 shows the origins of the 7 stack components.

The reason why "stacks" are used here, is that abstract data types (ADTs) including "stacks" have been well defined during the last decades. Thus, manual checking of whether they are following the reuse guidelines is comparatively easier than other software components. This aspect is especially attractive considering that few CASE tools, if any, with basic functionality, exist in the area of software reuse.

### 5.2.3 Experimental Framework

The 7 stacks mentioned in the previous section are examined to get their values of "reusability". The metrics were named "property metrics" in chapter 4, "Reuse Metrics" of this thesis. All metrics originated from the guidelines. As explained earlier, the property, "reusability" is decomposed into lower-level factors until measurable ones appear. In those decomposition schemes, each constituents are equally valued. Since the correct weighting is not yet known, allotting equal weighting is the best feasible strategy for now. Therefore the "reusability formula" suggested in section 4.2.4 is instantiated as follows.

| Components | Author | Organisation | Date | Repository |
|---|---|---|---|---|
| Stack1 | David Blanchard | Science Applications International Corporation | 16 August 1989 | AdaBasis |
| Stack2 | Ron Kownacki | | | AdaBasis |
| Stack3 | Bill Toscano, Michael Gordon | Intermetrics, Inc. | 15 October 1985 | ELSA |
| Stack4 | Bill Toscano, Michael Gordon | Intermetrics, Inc. | 15 October 1985 | ELSA |
| Stack5 | Tom Duke | TI Ada Technology Branch | 16 April 1985 | ELSA |
| Stack6 | A Strohmeier | LGL at EPFL | 10 July 1987 | LGL |
| Stack7 | A Strohmeier | LGL at EPFL | 18 August 1987 | LGL |

Table 5.1: Materials used in this Case Study

$$Reusability = \frac{Adaptability + Comprehensibility + Independence + Robustness}{4}$$

Further steps of decomposition are omitted here, for they are explained in the next section in great detail.

Another important thing related to measuring reusability of components is how to assign suitable values to each empirical object. The author here uses the "normal distribution" to map each empirical object to formal objects. The normal distribution curve has been broadly used in schools and universities where a comparative evaluation of their students' academic records was adopted. Mapping between empirical and formal objects was explained in section 4.2.2. An example of this can be found in figure 5.2.

Suppose $y$ represents the normal variable, then the height of the probability distribution for a specific value of $y$ is represented by $f(y)$ [1] [67, chapter 3].

Finally many metrics were collected with three CASE tools. Detailed description on those tools can be found in Appendix 2.

[1]For the normal distribution

$$f(y) = (1/\sqrt{2\pi}\sigma)e^{-1/2[(y-\mu)/\sigma]^2}$$

where $\mu$ and $\sigma$ are the mean and standard deviation, respectively, of the population of $y$ values.

## 5.3 Experimental Results

### 5.3.1 Adaptability

#### (1) Completeness

Guidelines relating *completeness* are guidelines A01 and A02 appearing in chapter 3. And they were checked automatically with CASE tools.

| Component | No. of Units | Completeness |
|---|---|---|
| Stack1 | 27 | 0.75 |
| Stack2 | 22 | 0.50 |
| Stack3 | 10 | 0.25 |
| Stack4 | 18 | 0.50 |
| Stack5 | 12 | 0.50 |
| Stack6 | 38 | 1.00 |
| Stack7 | 44 | 1.00 |

Let $\mu$ be *mean*, $\sigma$ be *standard deviation*, $y$ be a *physical object*. Then $\mu = 24$ and $\sigma = \sqrt{\frac{\sum_{i=1}^{n}(y_i-\mu)^2}{n-1}} = 13$ are obtained. Through a mapping $g$, a physical object $y$ can be converted into a formal object on the basis of the "normal distribution curve" as follows:

$$g(y < 11) = 0.25,$$
$$g(11 \leq y < 24) = 0.50,$$
$$g(24 \leq y < 37) = 0.75, \text{ and}$$
$$g(y \geq 37) = 1.00$$

#### (2) Generality

Guidelines A05 to A09 are concerned with *generality.* Unfortunately CASE tools are available for checking them. But they were checked comparatively easily by code inspection.

In line with the powerfulness of generality, GADT (Generic Abstract Data Type), PGADO (Parameterised Generic Abstract Data Object), GADO (Generic Abstract Data object), ADT (Abstract Data Type) and ADO (Abstract Data Object) are assigned points of 1.00, 0.75, 0.50, 0.25 and 0.00 accordingly.

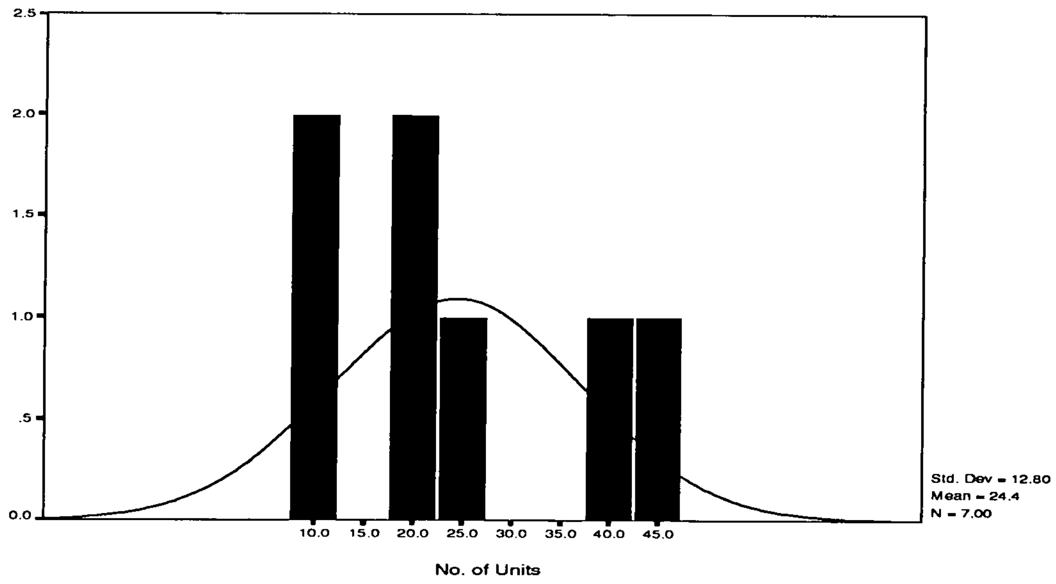


Figure 5.2: Graph showing No. of Units in 7 Stacks with their Normal Curve

| Component | Classification | Point | Generality |
|---|---|---|---|
| Stackl | GADT | 1.00 | 0.75 |
| Stack2 | GADT | 1.00 | 0.75 |
| Stack3 | ADO | 0.00 | 0.25 |
| Stack4 | ADO | 0.00 | 0.25 |
| Stack5 | GADT | 1.00 | 0.75 |
| Stack6 | GADT | 1.00 | 0.75 |
| Stack7 | GADT | 1.00 | 0.75 |

The same procedures as in the case of completeness were used to obtain component generality.

## (3) Adaptability

As a whole, the property *adaptability* can be calculated through composing the above two properties as shown below. Figure 5.3 shows relationships among them.

$$Adaptability = \frac{Completeness + Generality}{2}$$

| Components | Adaptability | Ranking |
|---|---|---|
| Stackl | 0.75 | 3 |
| Stack2 | 0.63 | 4 |
| Stack3 | 0.25 | 7 |
| Stack4 | 0.38 | 6 |
| Stack5 | 0.63 | 4 |
| Stack6 | 0.88 | 1 |
| Stack7 | 0.88 | 1 |

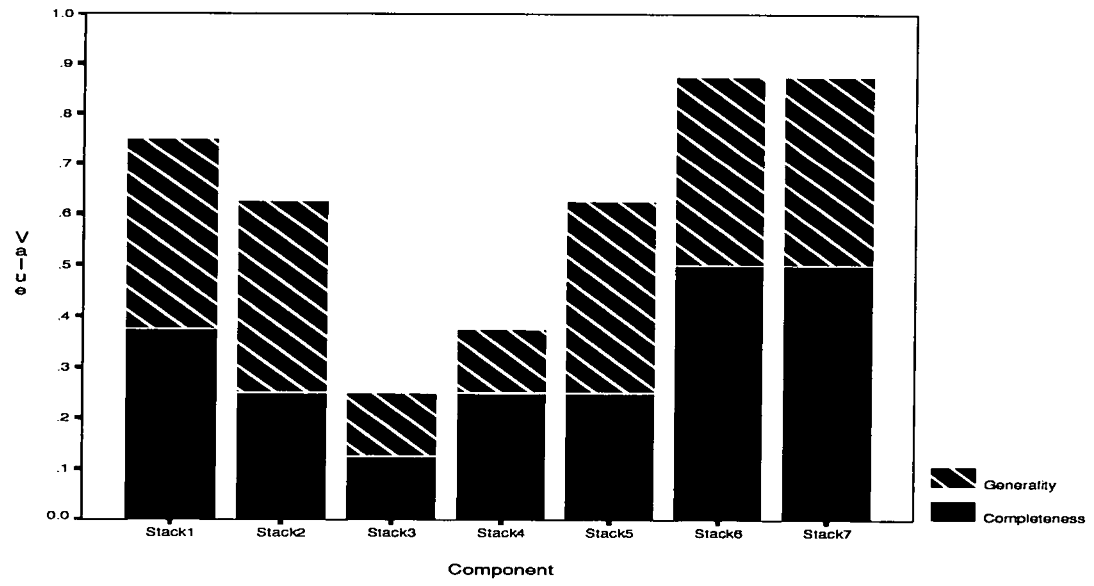


Figure 5.3: Adaptability of 7 Stacks

## 5.3.2 Comprehensibility

### (1) Conciseness and Communicativeness

These properties can be inspected by two things, i.e. the quantity and quality of commenting.

### (i) The Quantity of Commenting

Guideline C02 is applied in this aspect, and it is automatic-checkable.

$$\frac{Comments}{Ada\ Statements} = Comment\ Rate$$

| Components | Comment Rate | Metric |
|---|---|---|
| Stackl | 6.90 | 0.75 |
| Stack2 | 1.77 | 0.50 |
| Stack3 | 3.21 | 1.00 |
| Stack4 | 3.25 | 0.75 |
| Stack5 | 2.58 | 0.50 |
| Stack6 | 0.85 | 0.50 |
| Stack7 | 0.86 | 0.50 |

It is noteworthy that stackl has an excessive amount of comments, so a penalty was given to it. The author believes that excessive commenting is as bad as scarce commenting, for it means the poor expressiveness of the code itself. The metric of stackl was lowered one level from 1.00 to 0.75.

### (ii) The Quality of Commenting

The quality of comments inserted into code is as important as the quantity for its readability. Guidelines C03 to C08 recommend uses of "file header", "program unit specification header", "program unit body header", "data comments", "statement comments" and "marker comments". For now, they are only checked manually.

| Components | C03 | C04 | C05 | C06 | C07 | C08 | Points | metric |
|---|---|---|---|---|---|---|---|---|
| Stack1 | O | X | X | O | O | X | 3 | 0.25 |
| Stack2 | X | O | O | O | O | X | 4 | 0.75 |
| Stack3 | X | O | O | O | O | O | 5 | 1.00 |
| Stack4 | X | O | O | O | O | O | 5 | 1.00 |
| Stack5 | O | X | X | O | O | X | 3 | 0.25 |
| Stack6 | O | O | X | O | O | X | 4 | 0.75 |
| Stack7 | O | O | X | O | O | X | 4 | 0.75 |

### (iii) Conciseness and Communicativeness

Through the composition of the above two factors, *conciseness and communicativeness* are computed as shown in the following table.

| Components | Conciseness & Communicativeness |
|---|---|
| Stack1 | 0.50 |
| Stack2 | 0.63 |
| Stack3 | 1.00 |
| Stack4 | 0.88 |
| Stack5 | 0.38 |
| Stack6 | 0.63 |
| Stack7 | 0.63 |

## (2) Identifier Qualification

Here the usage statistics of "use" clauses are surveyed. Guideline C09, which is automatically checkable, is related to them. A detailed explanation of the harmfulness of using "use" clauses in Ada can be found in chapter 3, where the relevant Ada reuse guidelines were proposed.

Here, the more "use" clauses code contains, the worse the code is. Therefore the values are given in reverse below.

$\mu = 0.86$, $\sigma = 1.46$

$$g(y < -0.60) = 1.00,$$
$$g(-0.60 \leq y < 0.86) = 0.75,$$
$$g(0.86 \leq y < 2.32) = 0.50, \text{ and}$$
$$g(y \geq 2.32) = 0.25$$

| Components | No. of Use Clauses | Metric |
|---|---|---|
| Stack1 | 0 | 0.50 |
| Stack2 | 0 | 0.75 |
| Stack3 | 3 | 0.25 |
| Stack4 | 3 | 0.25 |
| Stack5 | 0 | 0.75 |
| Stack6 | 0 | 0.75 |
| Stack7 | 0 | 0.75 |

## (3) Nesting

Both guideline C15 and C16 were automatically checked to produce the needed data. The investigated data were obtained by multiplying the number of occurrences of nesting constructs by their nesting level.

| Components | Nesting | Metric |
|---|---|---|
| Stack1 | 0 | 0.75 |
| Stack2 | 1 | 0.75 |
| Stack3 | 3 | 0.75 |
| Stack4 | 5 | 0.75 |
| Stack5 | 0 | 0.75 |
| Stack6 | 17 | 0.25 |
| Stack7 | 17 | 0.25 |

As in the the above case, the reverse assignment of values was done since deeper nesting should be avoided.

## (4) Self-descriptiveness

This property is especially difficult to measure since it is sematic. Thus the relevant guidelines C18, C19, C20, and C22 are only checked whether they are well applied.

| Components | C18 | C19 & C20 | C22 | Points | Metric |
|---|---|---|---|---|---|
| Stack1 | O | O | O | 3 | 0.75 |
| Stack1 | X | X | O | 1 | 0.25 |
| Stack1 | O | X | O | 2 | 0.50 |
| Stack1 | O | X | O | 2 | 0.50 |
| Stack1 | O | O | O | 3 | 0.75 |
| Stack1 | O | O | O | 3 | 0.75 |
| Stack1 | O | O | O | 3 | 0.75 |

## (5) Layout

Relating to code layout, both horizontal spacing and vertical spacing can be said as one of important factors.

Guideline C26 can be checked automatically with tools.

$$\frac{Lines - Comments}{Statements}$$

| Components | Spacing | Metric |
|---|---|---|
| Stackl | 2.53 | 0.50 |
| Stack2 | 2.93 | 0.75 |
| Stack3 | 2.97 | 0.75 |
| Stack4 | 3.16 | 1.00 |
| Stack5 | 2.81 | 0.75 |
| Stack6 | 1.82 | 0.25 |
| Stack7 | 1.74 | 0.25 |

## (6) Volume

Although this was not included in the list of guidelines, we can easily guess that more readable code is of lower complexity. The representative methods were studied by McCabe and Halstead. Here, Halstead's complexity metrics are used. Further, the following relationships can be safely drawn:

$$Comprehensibility = k_1 \frac{1}{Complexity}$$

$$Complexity = k_2 \times Volume$$

where $k_1$ and $k_2$ are proportional constants, respectively.

| Components | Volume | Metric |
|---|---|---|
| Stackl | 0.00 | 1.00 |
| Stack2 | 851.60 | 0.75 |
| Stack3 | 363.90 | 0.75 |
| Stack4 | 550.00 | 0.75 |
| Stack5 | 444.00 | 0.75 |
| Stack6 | 3073.10 | 0.25 |
| Stack7 | 2924.20 | 0.25 |

As source code of lower complexity is better than ones of higher complexity, values are given reversely. Since stackl does not contain any operators and operands, its volume is 0.00.

## (7) Comprehensibility

On the basis of the above data, *comprehensibility* is calculated as below, and figure 5.4 is about this.

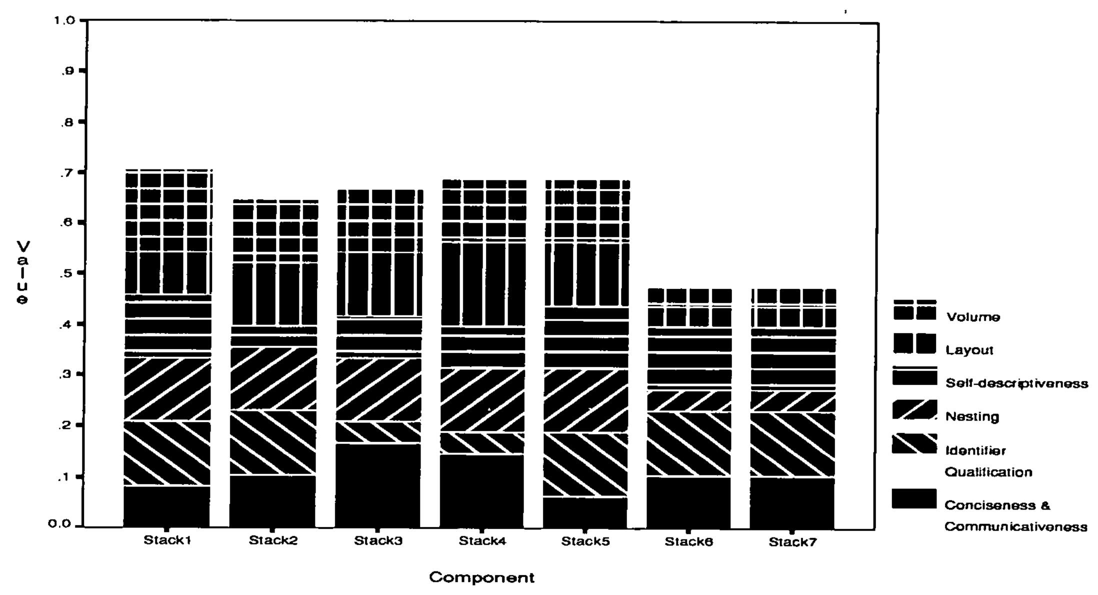


Figure 5.4: Comprehensibility of 7 Stacks

| Components | Comprehensibility | Ranking |
|---|---|---|
| Stack1 | 0.71 | 1 |
| Stack2 | 0.65 | 5 |
| Stack3 | 0.67 | 4 |
| Stack4 | 0.69 | 2 |
| Stack5 | 0.69 | 2 |
| Stack6 | 0.48 | 6 |
| Stack7 | 0.48 | 6 |

### 5.3.3 Independence

#### (1) Coupling

Coupling between modules can be detected through counting the number of "withs". This is originated from guidelines I02 and I03.

| Components | No. of withs | Metric |
|---|---|---|
| Stack1 | 4 | 0.50 |
| Stack2 | 2 | 0.75 |
| Stack3 | 3 | 0.50 |
| Stack4 | 3 | 0.50 |
| Stack5 | 0 | 1.00 |
| Stack6 | 3 | 0.50 |
| Stack7 | 5 | 0.25 |

Having more "with" clauses means higher coupling. So the reverse assignment is used to assign values as shown below:

$\mu = 2.86$, $\sigma = 1.57$

$$g(y < 1.29) = 1.00,$$
$$g(1.29 \leq y < 2.86) = 0.75,$$
$$g(2.86 \leq y < 4.43) = 0.50, \text{ and}$$
$$g(y \geq 0.25) = 0.25$$

## (2) Machine Independence

The relevant guidelines to machine independence are guidelines I04 and I06. The former is automatic-checkable, whereas the latter is checked manually by inspecting code itself.

Five items were investigated:

- number of machine code statements,
- interfaces to non-Ada routines,
- X11R4 interfaces,
- Motif 1.1 interfaces, and
- number of implement dependent IO packages.

An exhaustive list of implement dependent IO packages appears in chapter 3.

As a result of checking the above things, it has found no components have contain those features. Thus, equally the highest value 1s are given to each components.

## (3) Software System Independence

### (i) Ada predefined types

Why Ada predefined types such as "integer" or "real" can affect portability of code was explained in guidelines I08 of chapter 3. Here, the occurrences of their usage are inspected. These were obtained with CASE tools.

| Components | Integer | Natural | Positive | Total | Metric |
|---|---|---|---|---|---|
| Stack1 | 0 | 2 | 0 | 2 | 0.75 |
| Stack2 | 0 | 3 | 0 | 3 | 0.75 |
| Stack3 | 0 | 0 | 0 | 0 | 0.75 |
| Stack4 | 0 | 0 | 0 | 0 | 0.75 |
| Stack5 | 0 | 4 | 1 | 5 | 0.50 |
| Stack6 | 1 | 9 | 1 | 11 | 0.25 |
| Stack7 | 1 | 9 | 1 | 11 | 0.25 |

## (ii) Pragmas

The numbers of Ada predefined pragmas and compiler-specific pragmas were collected with respect to guideline I13. These were easily detected with tools.

| Components | Ada predefined pragmas | Compiler-specific pragmas | Total occurrences | Metric |
|---|---|---|---|---|
| Stack1 | 1 | 0 | 1 | 0.25 |
| Stack2 | 0 | 0 | 0 | 0.75 |
| Stack3 | 0 | 0 | 0 | 0.75 |
| Stack4 | 0 | 0 | 0 | 0.75 |
| Stack5 | 0 | 0 | 0 | 0.75 |
| Stack6 | 0 | 0 | 0 | 0.75 |
| Stack7 | 0 | 0 | 0 | 0.75 |

## (iii) Software System Independence

Software system independence of each 7 stacks was calculated with the above two groups (i) and (ii) of data.

| Components | Metric |
|---|---|
| Stack1 | 0.50 |
| Stack2 | 0.75 |
| Stack3 | 0.75 |
| Stack4 | 0.75 |
| Stack5 | 0.63 |
| Stack6 | 0.50 |
| Stack7 | 0.50 |

## (4) Independence

On the whole, independence of the 7 stacks was obtained as follows:

$$Independence = \frac{Coupling + Machine\ Independence + Software\ Independence}{3}$$

| Components | Independence | Ranking |
|---|---|---|
| Stack1 | 0.67 | 5 |
| Stack2 | 0.83 | 2 |
| Stack3 | 0.75 | 3 |
| Stack4 | 0.75 | 3 |
| Stack5 | 0.88 | 1 |
| Stack6 | 0.67 | 5 |
| Stack7 | 0.58 | 7 |

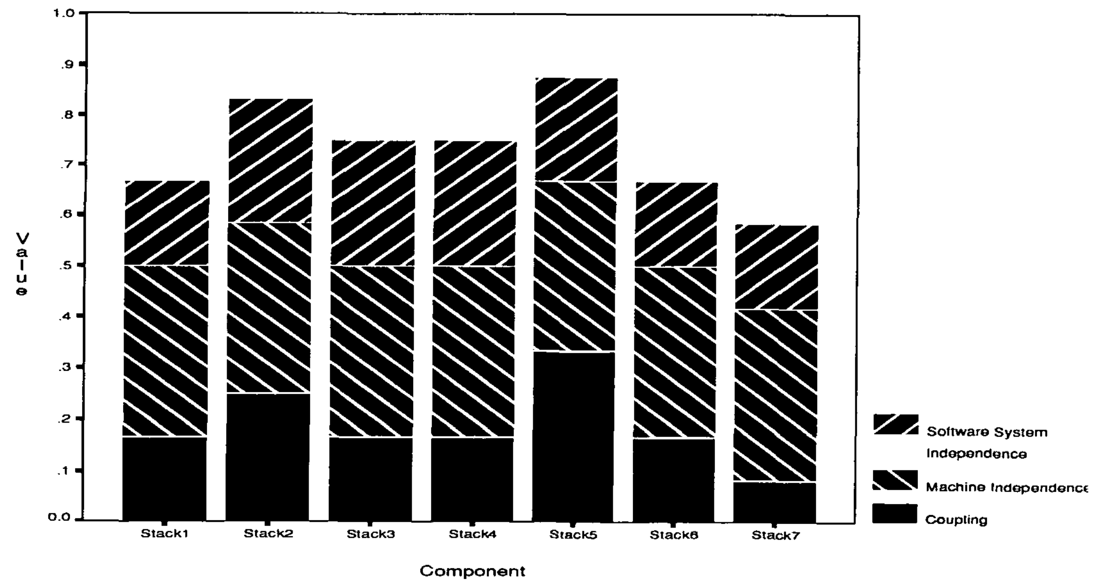


Figure 5.5: Independence of 7 Stacks

### 5.3.4 Robustness

Only few guidelines on robustness are checkable either manually or automatically. Guidelines R01 and R03 are the relevant guidelines. Here, the number of "exception declarations" and "raise" statements were counted with regard to measuring robustness of components, for they are believed to contribute to fault tolerance. The number of occurrences of "when others" following "null" statement were also checked.

#### (1) No. of exception declarations

| Components | Exception declarations | Metric |
|---|---|---|
| Stack1 | 3 | 1.00 |
| Stack2 | 2 | 0.75 |
| Stack3 | 2 | 0.75 |
| Stack4 | 2 | 0.75 |
| Stack5 | 3 | 1.00 |
| Stack6 | 1 | 0.25 |
| Stack7 | 1 | 0.25 |

### (2) No. of raise statements

| Components | Raise statements | Metric |
|---|---|---|
| Stack1 | 6 | 0.75 |
| Stack2 | 10 | 1.00 |
| Stack3 | 3 | 0.25 |
| Stack4 | 5 | 0.50 |
| Stack5 | 7 | 0.75 |
| Stack6 | 4 | 0.50 |
| Stack7 | 4 | 0.50 |

### (3) The number of when others + null statement

All stacks do not contain the constructs. So the values of 1s are given to each of them equally. Since both $\mu$ and $\sigma$ are 0, all metrics become 1.00 .

### (4) Robustness

Based upon the above 3 groups (1), (2) and (3) of data, robustness was computed as follows.

| Components | Robustness | Ranking |
|---|---|---|
| Stack1 | 0.92 | 1 |
| Stack2 | 0.92 | 1 |
| Stack3 | 0.67 | 5 |
| Stack4 | 0.75 | 4 |
| Stack5 | 0.92 | 1 |
| Stack6 | 0.58 | 6 |
| Stack7 | 0.58 | 6 |

## 5.3.5 Reusability

The following table shows each properties of the 6 stacks. Reusability was calculated on the basis of the 4 constituent properties. At the last column, the rankings of the stacks appear. The best stack, i.e. the highest reusable stack was identified as Stack5, whereas the worst one is Stack3.

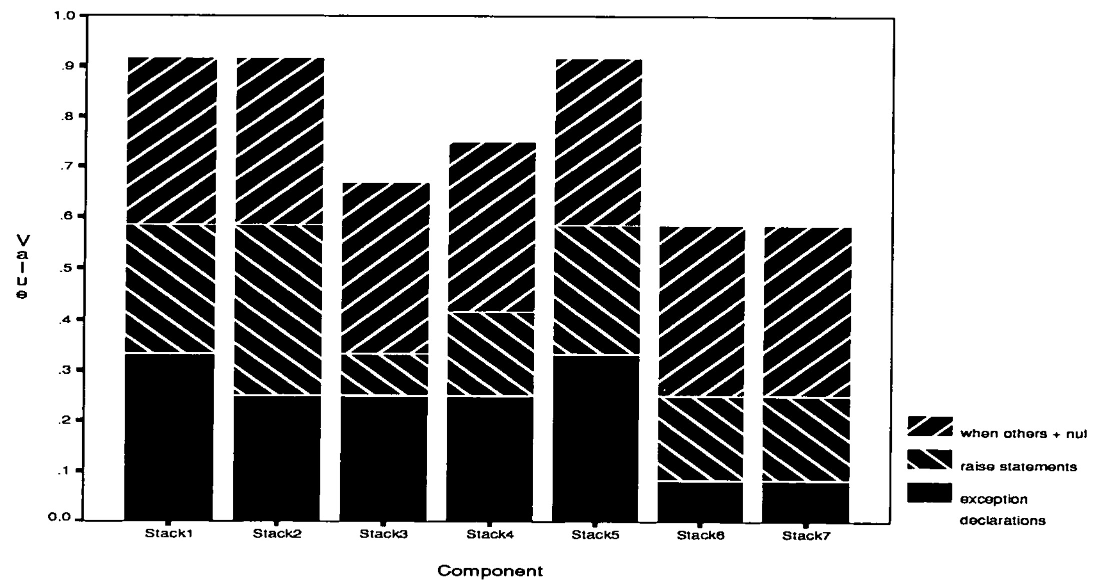


Figure 5.6: Robustness of 7 Stacks

| Components | Adaptability | Comprehensibility | Independence | Robustness | Reusability | Ranking |
|---|---|---|---|---|---|---|
| Stackl | 0.75 | 0.71 | 0.67 | 0.88 | 0.75 | 2 |
| Stack2 | 0.63 | 0.65 | 0.83 | 0.88 | 0.75 | 2 |
| Stack3 | 0.25 | 0.67 | 0.75 | 0.50 | 0.54 | 7 |
| Stack4 | 0.38 | 0.69 | 0.75 | 0.63 | 0.61 | 4 |
| Stack5 | 0.63 | 0.69 | 0.88 | 0.88 | 0.77 | 1 |
| Stack6 | 0.88 | 0.48 | 0.67 | 0.38 | 0.60 | 5 |
| Stack7 | 0.88 | 0.48 | 0.58 | 0.38 | 0.58 | 6 |

## 5.4 Experimental Analysis

Through the above experimental results, the following two facts were observed.

Firstly, when a component has the 4 constituents, i.e. adaptability, comprehensibility, independence and robustness, of a small disparity, it tends to be highly reusable. Figure 5.8 shows the relationship between disparity of 4 properties and reusability of components. Here, we can say that a strong relationship exists between them, and it forces us to develop components with high measures of all 4 properties.

Secondly, to investigate the usability of the proposed guidelines, guideline usage statistics were collected. The following 5 pie charts show the portions of automatic checkable guidelines, manually checkable guidelines and non-checkable guidelines to the total number of guidelines. Generally, guidelines, except for robustness, are well checked. The most verifiable guidelines are on independence. This means much research has been focussed on

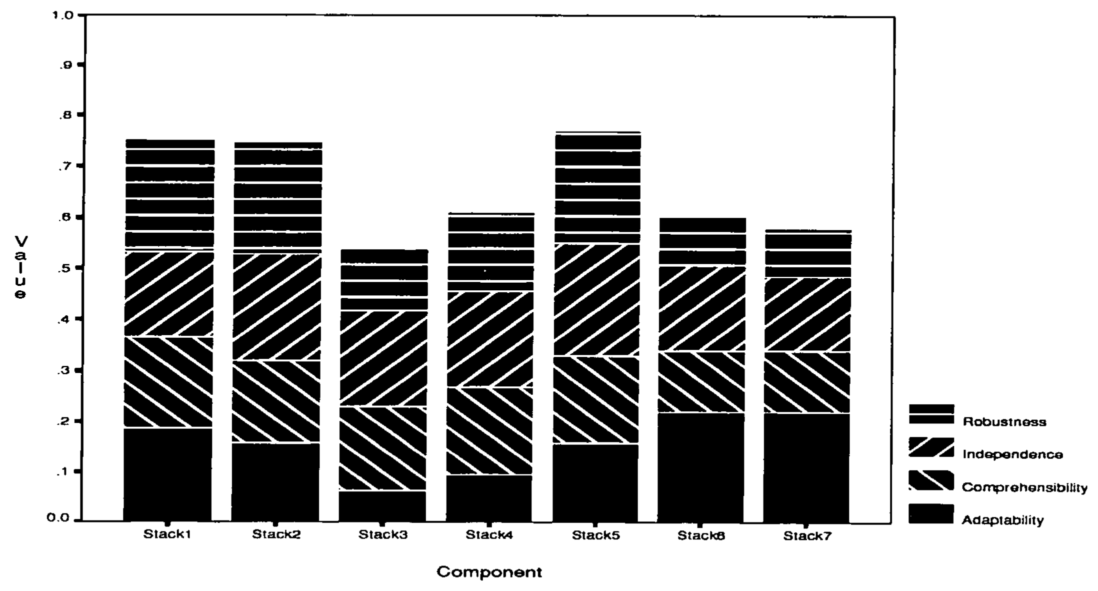


Figure 5.7: Reusability of 7 Stacks

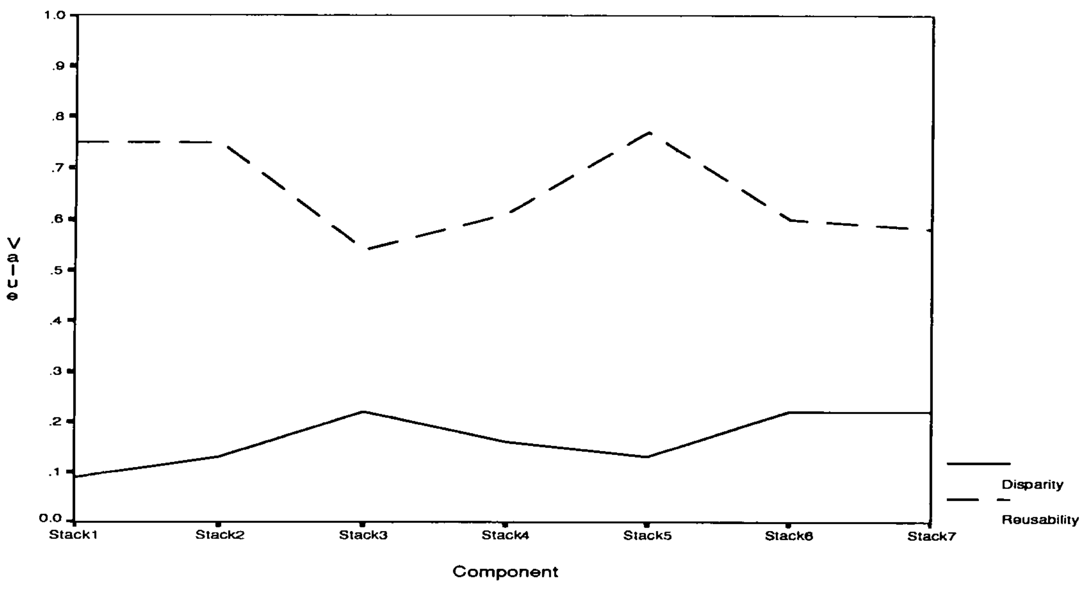


Figure 5.8: Relationship between disparity of properties and reusability of components

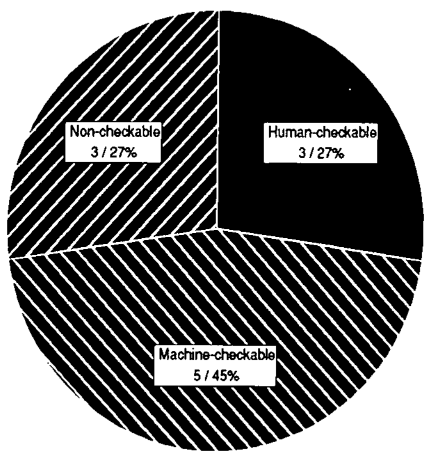


Figure 5.9: Guideline Usage Statistics on Adaptability

increasing independence. There has also been much research on increasing portability; theories on modular programming or object-oriented programming are representative ones of this research. However, a large portion of other guidelines can only be checked by manual inspection. This suggests that more effort needs to be spent in those areas, especially, in the case of robustness, where only two out of fourteen were checkable.

## 5.5 Summary

In this chapter, experiments were carried out to validate the proposed Ada reuse guidelines. Also reusability of components was measured through the "divide and conquer" strategy, in other words, following the "decomposition and recomposition" principle. Through these experiments, two conclusions were obtained. They were about property disparity and guideline usage.

This kind of experimentation can be done with other ADTs or components. If more comprehensive data are collected and a deeper knowledge on "reusability" is obtained, then more correct weighting and mapping strategies can be established. Subsequently, this would lead to the production of components that have high reusability which can be established through a reliable means of measurement.

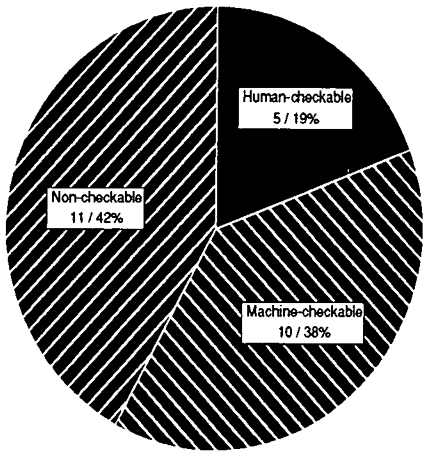


Figure 5.10: Guideline Usage Statistics on Comprehensibility

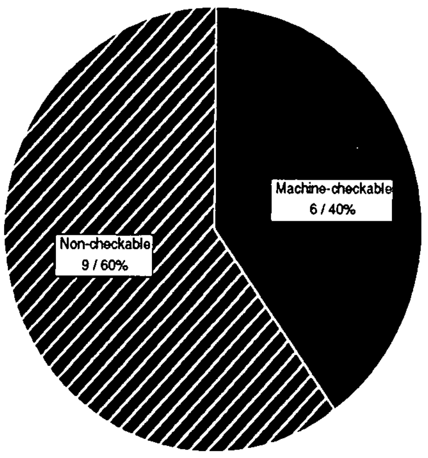


Figure 5.11: Guideline Usage Statistics on Independence

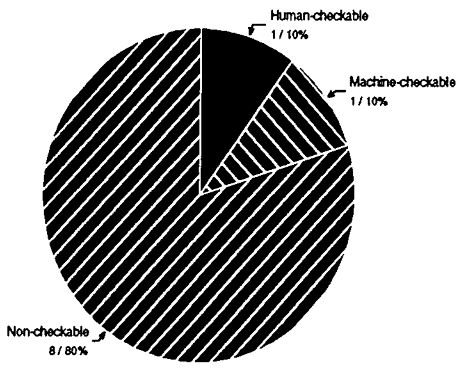


Figure 5.12: Guideline Usage Statistics on Robustness

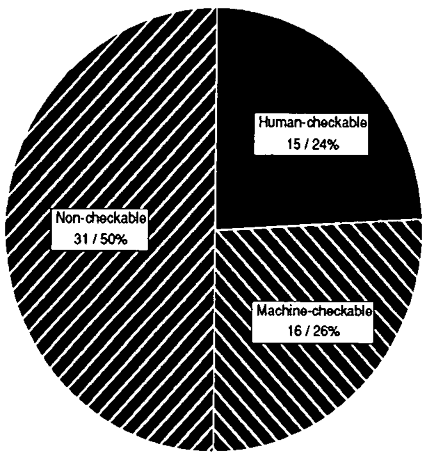


Figure 5.13: Guideline Usage Statistics on Reusability

# Chapter 6

# Evaluation

In this chapter the proposed Ada code reuse method is evaluated against the criteria for success, its strength and weaknesses are discussed.

## 6.1 Evaluation Against the Criteria for Success

Chapter 1 presented a list of the criteria against which this thesis can be evaluated. Each of these criteria is now addressed.

1. Identifying the differences between Ada code reuse and high level reuse or code reuse in other programming languages.

   This was addressed mainly in chapter 2 and 3. Although bigger benefits are expected from reusing higher-level components such as requirements, specifications or design, at the current phase, code reuse is more feasible than reuse of higher level components. The reasons were explained in section 2.2 in full. The advantages of implementing reuse with Ada against other languages were identified with respect to the language's distinctive features such as generics or packages in section 3.1.2. Through the study, it was learnt that choosing the most suitable language for a certain situation is more important than insisting a language's superiority.

2. Establishment of the exact meaning of the term *reusability.*

   Hooper's definition of reusability was adopted, and it was further decomposed with McCall and Boehm's theory. The decomposition strategy allowed reusability to be broken down into more measurable and visible constituents.

3. Suggesting guidelines as an approach to code reuse.

   Ada reuse guidelines were used as a means to improve code reusability. A total of 62 guidelines was collected either as-it-is or in a modified manner from the previous research work done in the last decade.

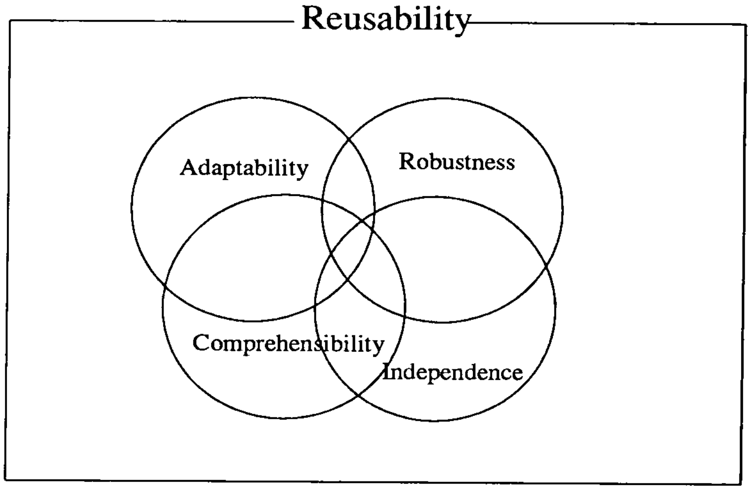


Figure 6.1: Reusability as a combination of overlapped properties

4. Validating the usefulness and usability of the guidelines with software measurement.

   Reuse metrics were used to validate the improvement of code reusability. In the case study, it was demonstrated that applying the guideline to Ada code improves the reusability of code. To promote the use of the proposed reuse guidelines, using them should be cost-effective. In the case study conducted, 22.39% of guidelines were known as machine-checkable. Thus it can be argued that increasing the rate of automatically-checkable guidelines is as crucial point for programming-in-the-large.

## 6.2 Strengths and Weaknesses

One of the strong points possessed by the proposed reuse guidelines is that they follow property-oriented approach, not language-oriented approach. That is, the guidelines were enumerated according to their constituent properties. Thus, users can get to know the weak aspects of components which they have produced, and subsequently, strengthen them. This enable users to attack various aspects of components to improve their reusability.

One weakness is depicted in figure 6.1. As shown in the figure, many unknown factors contributing to code reusability are still missing. Also, the four identified factors of reusability overlap. This was experienced when guidelines were divided into the four groups, i.e. adaptability, comprehensibility, independence and robustness. But it is fair to say that they adequately cover code reusability. The fact was demonstrated through experiments done in the chapter 5. In the future, when more abundant and correct information on code has been obtained, we may be able to know more exactly what contributes to code reusability than we do at present.

Another unsatisfactory feature of the guidelines is that general guidelines were mixed with very specific ones. It is believed that two-level scheme, i.e. general guidelines and their instantiations, is better for applying and checking the guidelines.

## 6.3 Summary

In this chapter, the proposed Ada code reuse method was evaluated. The criteria for the success of this research appearing in chapter 1 were discussed.

In the next section, its strong points and weak points were identified.

Through the above evaluation, it can be said that the soundness and usefulness of the guidelines have been demonstrated.

# Chapter 7

# Conclusions

## 7.1 The Main Achievements of the Research

The main achievements and results of this research are Ada code reuse guidelines for design-for-reuse, divided into 4 groups, i.e. adaptability, comprehensibility, independence and robustness.

Another useful result was succeeding in measuring component reusability with the aid of software metrics. Through this, usability of the guidelines and relationships between the constituents of reusability were shown.

## 7.2 General Conclusions of the Research

Through the study and experiments done, there are 4 conclusions which can be inferred:

- It is meaningless to insist that a certain language is better than others in terms of producing reusable components. But it is right to say that using Ada is more suitable for large projects which require a strong quality and team work than many other commonly used languages.
- Simply programming in Ada does not guarantee that highly reusable code will be produced. Therefore guidelines are needed for users. And the guidelines must possess good standards allowing users easily to adopt them into their programming habits or projects.
- As an illicit, i.e. an intangible concept, the property "reusability" has been long believed unmeasurable, thus not attackable. But in this thesis, it was observed that "reusability" can be addressed and measured by the "decomposition and recomposition" strategy.

- As a result of the case study conducted, it was uncovered that highly reusable components maintain a good balance of their properties. This means that if a component has adaptability of a low level, while having other properties of high levels, it would become a low reusable component.

## 7.3 The Limitations of the Approach

Firstly, in this research, each proposed reusability property was evaluated with respect to each components, not as part of software development. Studying the impacts of using reusable components in real industrial environments would be valuable.

Secondly, an equal weighting was adopted in the case study. Since we cannot say all factors of reusability have the same weight, more exact weighting methods must be established.

Finally, the proposed approach may not be suitable to be used in the programming-in-the-large situations. Processes for reuse engineering and CASE tools for producing reusable Ada components semi-automatically, if not automatically, should be studied and developed.

## 7.4 Suggestions for Future Research

What is missing in the research described here are guidelines related to "concurrency" and "OOP (Object-Oriented Programming)" concepts, which are both available in Ada. The reason why the former was not addressed here was its complexity. The latter was not handled since validated "Ada 95" compilers with which OOP is possible, do not exist yet. But two of the above things are desperately needed to achieve the goals, i.e. developing reliable real-time embedded systems, for which Ada was developed.

Finally, another field of further work would be to investigate the applicability of the proposed reuse method to other languages such as C++. That is because C++ is one of the most heavily used languages nowadays together with Ada. Benefits from reusing those components programmed in C++ would be enormous as in the case of Ada.

## 7.5 Summary

As mentioned earlier, software engineering, especially reuse engineering, is still at its immature phase. Therefore, we should learn by example of other mature engineering disciplines, where high-level reuse such as specification reuse or design reuse is a common practice, while trying to find the most feasible solution such as code reuse.

# Appendix A

# Collected Ada Code Reuse Guidelines

The following is a complete list of the Ada code reuse guidelines presented in this thesis.

## A.1 Principle of Adaptability

Completeness

A01: Make components as complete as possible.

A02: Provide complete functionality in a reusable part or set of parts.

A03: Provide initialisation and finalisation procedures for every data structure that may contain dynamic data.

A04: Make all dependent components reusable.

Generality

A05: Use generic units to avoid code duplication.

A06: Parameterise generic units for maximum adaptability.

A07: Use generic units to encapsulate algorithms independently of data type.

A08: Use abstract data types in preference to abstract data objects.

A09: Use generic units to implement abstract data types independently of their component data type.

A10: Select the least restrictive names possible for reusable parts and their identifiers.

A11: Select the generic name to avoid conflicting with the naming conventions of instantiations of the generic.

## A.2 Principle of Comprehensibility

Cohesion

C01: Make cohesion high within each component.

Conciseness and Communicativeness

C02: Make each comment adequate, concise and precise.

C03: Put a file header on each source file.

C04: Put a header on the specification of each program unit.

C05: Place information required by the maintainer of the program unit in the body header.

C06: Comment on all data types, objects, and exceptions unless their names are self-explanatory.

C07: Minimise comments embedded among statements.

C08: Use pagination markers to mark program unit boundaries.

Identifier Qualification

C09: Minimise the use of "use" clauses.

Information Hiding

C10: Only place in the specification section those declarations that must be seen externally.

C11: Only "with" those compilation units that are really needed.

C12: Use private and limited private types to promote information hiding.

C13: Try to use limited private types.

C14: Use mode "in out" rather than "out" for parameters of a generic formal subprogram, when the parameters are of an imported limited type.

Nesting

C15: Use "elsif" for nested "if" statements.

C16: Do not nest expressions or control structures beyond a nesting level of five.

Overloading

C17: Do not overload names from package "Standard".

Self-descriptiveness

C18: Make reserved words and other elements of the program visually distinct from each other.

C19: Use descriptive identifier names.

C20: Do not use any abbreviations in identifier or unit names.

C21: Use names which indicate the behavioural characteristics of the reusable part, as well as its abstraction.

C22: Do not hard code array index designations.

C23: Use named constants for parameter defaults.

C24: Use named parameters association.

C25: Use descriptive named constants as return values.

Layout

C26: Code program in a well-arranged manner horizontally and vertically.

## A.3 Principle of Independence

Coupling

I01: Make coupling low.

I02: Minimise "with" clauses on reusable parts, especially on their specifications.

I03: Use generic parameters instead of "with" statements to reduce the number of context clauses on a reusable part, and to import portions of a package rather than the entire package.

Machine Independence

I04: Machine-dependent and low-level Ada features should be avoided except when absolutely necessary.

I05: Encapsulate input/output (I/O) uses into a separate I/O package.

I06: Minimise the use of implementation dependent I/O procedures.

Software System Independence

I07: Use the predefined packages for string handling.

I08: Avoid predefined and implementation defined types.

I09: Explicitly specify the precision required.

I10: Use "attributes" instead of explicit constraints.

I11: Use explicitly declared types for integer ranges in the loop statement.

I12: Avoid optional language features.

I13: Avoid using pragmas.

I14: Close files before a program completes.

I15: Do not input or output "access" types.

## A.4 Principle of Robustness

Error Tolerance

R01: Put a modest amount of "exception" and "raise" statements in code.

R02: Propagate exceptions out of reusable parts.

R03: Never use the "when others" construct with the "null" statement.

R04: Avoid pragma "suppress".

R05: Do not propagate an exception beyond where its name is visible.

R06: Do not propagate predefined exceptions without renaming them.

R07: Do not execute normal control statements from an exception.

R08: Use range constraints on numeric types.

R09: Explicitly declare a type to use in defining discrete ranges.

R10: Avoid using the *when others* clause as a shorthand notation.

# Appendix B

# Materials used in the Case Study

## B.1 Introduction to 3 repositories used

Source codes and CASE tools used in this thesis were from the following 3 software repositories[1].

### B.1.1 AdaBasis

AdaBasis - an acronym for the German phrase "Bibliothek anwendungsbezogener Ada Software-Komponenten in Stuttgart" - is a repository of free Ada Software, presented in a way that is (hopefully) easy to use and allows flexible access and effective searching.

The software in this repository is based mainly on the "PAL (Public Ada Library)" and is still extending. It is presented in a hierarchical manner, separated in different application domains, and, for some domains, with an additional searching facility. The repository is found in

http://www.informatik.uni-stuttgart.de/ifi/ps/ada-software/ada-software.html

### B.1.2 ELSA

ELSA (Electronic Library Services & Applications) is a NASA funded service provided by MountainNet, providing access to a large selection of high quality software examined for integrity and compatibility. ELSA project is the operational part of the Repository Based Software Engineering (RBSE) program. RBSE is a National Aeronautics and Space Administration (NASA) sponsored program dedicated to introducing and supporting common,

[1]The HTTP (Hyper-Text Transfer Protocol) addressed of the 3 repositories are correct at the time of writing of the thesis.

effective approaches to designing, building, and maintaining software systems by using existing software assets stored in a specialised library or repository.

In addition to operating a software life-cycle repository, RBSE promotes software engineering technology transfer, academic and instructional support for reuse programs, the use of common software engineering standards and practices, software reuse technology research, and interoperability between reuse libraries/repositories. During its life cycle, the ELSA project responded to emerging technologies, the growing sophistication of its client base, and industry trends by advancing the capabilities of its management software. Thus, ELSA stands as a customer-driven environment employing an advanced library management mechanism, MORE (Multimedia Oriented Repository Environment). The HTTP address is "http://rbse.mountain.net/ELSA/elsa_lob.html".

### B.1.3 LGL

The Software Engineering Lab (DI-LGL), at the Swiss Federal Institute of Technology in Lausanne (EPFL), distributes reusable Ada software components by FTP.

These components have been used (and re-used :-) since the mid-80's both by students (in a software engineering course) and in industrial applications. All of the package specifications as well as the documentation are now available through WWW. This repository is at "http://lglwww.epfl.ch/Components/".

## B.2 Introduction to 3 CASE Tools used

### B.2.1 Ada System Dependency Analyzer 2.1

The authors of this tool are Richard Conn (Design / Code), Grace Baratta-Perez (Code / Test / Document), Charles Finnell (Consultant), and Thomas Walsh (Group Leader) of the MITRE Corporation. It was developed in 1994. A short abstract from ELSA repository, where the tool is stored, follows below.

The Ada System Dependency Analyzer (SDA) is a software architecture analysis tool that generates a quantitative snapshot of an Ada application's software architecture. It can process thousands of Ada source files (at rates as high as 24,000 lines of code per CPU minute on some platforms) during a single run and report on them as a group of files comprising a single Ada system. It identifies Ada source code dependencies on Commercial Off-The-Shelf (COTS) products such as operating systems, compilers, the X Window System, and on routines written in other languages. With this analysis, it aids in predicting software portability and reliability problems. Finally, it presents statistics on the files analysed (number of lines of code, program units in each file, etc.), compilation order information, exception declaration and usage information, details on withing relationships between program units, and other useful items of information on the system analysed.

It has been released through the PAL in binary only. Release of the source code is on a case-by-case basis; contact the authors to obtain the source code.

A user's manual is included in the distribution. Additional information can be found in the February 1994 issue of IEEE Computer magazine, Volume 27, Number 2, pages 49 to 55 in the article entitled "Ada System Dependency Analyzer Tool."

This tool was developed by employees of The MITRE Corporation. Funding was provided by PM Common Hardware/Software of the U.S. Army.

### B.2.2 Certifier_1

This tool was developed by Richard Conn, Manager of Public Ada Library in 1994. A short abstract from him follows below.

The second stage in the development of the Public Ada Library (PAL) has begun with the introduction of the concept of certification to the Ada source code in the library. A program, Certifier_1, has been created that will be initially used to evaluate all Ada source code submitted to the PAL. Certifier_1 has the ability to analyse thousands of files in a single pass, checking on their interdependencies. It ranks the files it is asked to analyse as OK or NOT OK and assigns a letter grade to the system (A, B, or C is OK, D and F are NOT OK).

Certifier_1 contains a lexical analyser and a parser for the Ada83 language. A grade of F is assigned to the system if syntax or lexical errors are encountered. Certifier_1 also builds an internal data structure describing the interdependencies of the library units and subunits. If stubs (subunit bodies) are missing and there are no syntax or lexical errors, a grade of D is assigned to the system indicating that major parts of it are missing. This is not necessarily bad; the Abstractions library from Intermetrics, for instance, received a letter grade of D because of missing subunits, but, when the Intermetrics Standards Checker was evaluated with Abstractions, the Standards Checker code filled in the missing subunits, giving the combined Standards Checker and Abstractions system a grade of A.

Certifier_1 also checks on compiler-specific pragmas, the use of machine code, and the withing of library units that are not a part of the analysed code. It awards lower grades (B and C) if all else is OK and one of these issues comes up. A grade of a B or a C may or may not mean there is a problem. Compiler-specific library units may be employed, causing the lower grade, for example. Also, it may be possible to raise the grade by including another components library, like CS Parts or New Abstractions, in the evaluation to fill in the missing library units. However, a B or a C may also mean that code has been omitted.

Certifier_1 generates two reports: a report for inclusion in the PAL database entry on the item and a log file which describes details on the problems encountered, including line numbers and file names on or near which the problems can be found. Log reports can be found for each item in the PAL by checking in the directory languages/ada/userdocs/catalog/c1_rpts.

Reports are named after the items on which they report; ada_sda.cl, for instance, is the report associated with the Software ID file ada_sda.sid.

Certifier_1 is by no means a final solution to the problem of certification of reusable software in a library. However, it is a start. It does not beat a compiler by any means, but it does provide a quick, first-look solution. It does not determine logical errors or problems with completeness. Many things can slip through Certifier_1, but, likewise, many things do not. It is a first step.

### B.2.3 Met_Pars

MET_PARS is a program that measures the complexity of Ada source code. It was developed in June 1991 by "Source Translation & Optimization (STO)". Researchers have found that complexity metrics are correlated with the understandability of the source code, development and maintenance costs, and error rates. STO (Source Translation & Optimization) sells a library of reusable Ada software for $450 from which MET_PARS and other tools are built.

This version of MET_PARS measures the Halstead metric, and displays a wide variety of information about components in the Ada program, as well as values for Halstead's metric. The parser does not handle the PRAGMA statement of Ada.

## B.3 Source Codes used

The following are source codes of two stacks which are each of highest and lowest reusability.

stack5.ada

```
-------- SIMTEL20 Ada Software Repository Prologue ------------
--                                                            -*
-- Unit name    : stack_package
-- Version      : 1.0
-- Author       : Tom Duke
--              : TI Ada Technology Branch
--              : PO Box 801, MS 8007
--              : McKinney, TX  75069
-- DDN Address  : DUKE%TI-EG at CSNET-RELAY
-- Copyright    : (c) N/A
-- Date created : 16 Apr 85
-- Release date : 16 Apr 85
-- Last update  : 16 Apr 85
-- Machine/System Compiled/Run on :DG MV 10000, ROLM ADE
--                                                           -*
-----------------------------------------------------------------
--                                                           -*
-- Keywords     : stack, generic stack
----------------:
--
-- Abstract     : This is a generic package that provides the types,
----------------: procedures, and exceptions to define an abstract stack
----------------: and its corresponding operations.  Using an
----------------: instantiation of this generic package, one can declare
----------------: multiple versions of a stack of type GENERIC_STACK.
----------------: The stack operations provided include:
----------------: 1. clear the stack,
----------------: 2. pop the stack,
----------------: 3. push an element onto the stack, and
----------------: 4. access the top element on the stack.
----------------:
--                                                           -*
------------------ Revision history -------------------------
--                                                           -*
-- DATE        VERSION AUTHOR                  HISTORY
-- 4/16/85     1.0     Tom Duke                Initial Release
--                                                           -*
------------------ Distribution and Copyright ----------------
--                                                           -*
-- This prologue must be included in all copies of this software.
--
-- This software is released to the Ada community.
-- This software is released to the Public Domain (note:
--   software released to the Public Domain is not subject
--   to copyright protection).
-- Restrictions on use or distribution:  NONE
--                                                           -*
------------------ Disclaimer -------------------------------
```

```
--                                                               -*
-- This software and its documentation are provided "AS IS" and
-- without any expressed or implied warranties whatsoever.
-- No warranties as to performance, merchantability, or fitness
-- for a particular purpose exist.
--
-- Because of the diversity of conditions and hardware under
-- which this software may be used, no warranty of fitness for
-- a particular purpose is offered.  The user is advised to
-- test the software thoroughly before relying on it.  The user
-- must assume the entire risk and liability of using this
-- software.
--
-- In no event shall any person or organization of people be
-- held responsible for any direct, indirect, consequential
-- or inconsequential damages or lost profits.
--                                                               -*
-------------------END-PROLOGUE--------------------------------

generic

  type ELEMENTS is private;
  SIZE : POSITIVE;

package STACK_PACKAGE is

  type GENERIC_STACK is  private;


  function TOP_ELEMENT( STACK  : in  GENERIC_STACK )
    return ELEMENTS;

  function STACK_IS_EMPTY( STACK : in GENERIC_STACK )
    return BOOLEAN;

  procedure CLEAR_STACK( STACK : in out GENERIC_STACK );
```

```
   procedure PUSH       ( FRAME : in ELEMENTS;
                          STACK : in out GENERIC_STACK );

   procedure POP        ( FRAME : out ELEMENTS;
                          STACK : in out GENERIC_STACK );

   NULL_STACK      : exception;
   STACK_OVERFLOW  : exception;
   STACK_UNDERFLOW : exception;


private

  type STACK_LIST is array ( 1 .. SIZE ) of ELEMENTS;

  type GENERIC_STACK  is
     record
      CONTENTS       :  STACK_LIST;
      TOP            :  NATURAL range NATURAL'FIRST .. SIZE := NATURAL'FIRST;
     end record;

end STACK_PACKAGE;


-----------------------------------------------------------------------


package body STACK_PACKAGE is

---------------
--  function TOP_ELEMENT  --  This function returns the value of the top
--                            element on the stack.  It does not return a
--  pointer to the top element.  If the stack is empty, a constraint error
--  occurs.  The exception handler will then raise the NULL_STACK
--  exception and pass it to the calling procedure.
---------------
  function TOP_ELEMENT( STACK : in  GENERIC_STACK ) return ELEMENTS is
  begin
   return STACK.CONTENTS(STACK.TOP);
   exception
      when CONSTRAINT_ERROR =>
         raise NULL_STACK;
      when others =>
         raise;
  end TOP_ELEMENT;


  ----------
  --  Is stack empty?
  ----------
  function STACK_IS_EMPTY( STACK : in GENERIC_STACK )
    return BOOLEAN is
  begin
    return (STACK.TOP = NATURAL'FIRST);
  exception
    when OTHERS =>
         raise;
  end STACK_IS_EMPTY;



---------------
--  procedure CLEAR_STACK  --  This procedure resets the stack pointer, TOP,
--                             to a value representing an empty stack.
---------------
  procedure CLEAR_STACK( STACK : in out GENERIC_STACK ) is
  begin
   STACK.TOP := NATURAL'FIRST;
  end CLEAR_STACK;



---------------
```

```
--  procedure PUSH  --  This procedure attempts to push another element onto
--                      the stack.  If the stack is full, a constraint error
--  occurs.  The exception handler will then raise the STACK_OVERFLOW
--  exception and pass it to the calling procedure.
---------------
  procedure PUSH       ( FRAME : in ELEMENTS;
                         STACK : in out GENERIC_STACK ) is
  begin
   STACK.TOP := STACK.TOP + 1;
   STACK.CONTENTS(STACK.TOP) := FRAME;
   exception
      when CONSTRAINT_ERROR =>
         raise STACK_OVERFLOW;
      when others =>
         raise;
  end PUSH;


---------------
--  procedure POP  --  This procedure attempts to pop an element from
--                     the stack.  If the stack is empty, a constraint error
--  occurs.  The exception handler will then raise the STACK_UNDERFLOW
--  exception and pass it to the calling procedure.
---------------
  procedure POP        ( FRAME : out ELEMENTS;
                         STACK : in out GENERIC_STACK ) is
  begin
   FRAME := STACK.CONTENTS(STACK.TOP);
   STACK.TOP := STACK.TOP - 1;
   exception
      when CONSTRAINT_ERROR =>
         raise STACK_UNDERFLOW;
      when others =>
         raise;
  end POP;
end STACK_PACKAGE;
```

<u>stack3.ada</u>

```
-- $Source: /nosc/work/parser/RCS/ParseStk.spc,v $
-- $Revision: 4.0 $ -- $Date: 85/02/19 11:33:03 $ -- $Author: carol $

------------------------------------------------------------------------

with ParserDeclarations;        -- declarations for the Parser
use ParserDeclarations;

package ParseStack is           --| Elements awaiting parsing

--| Overview
--|
--| The ParseStack used by the parser.
--|
--| This data structure has the following sets of operations:
--|
--| 1) A set that add and delete elements.  This set can
--| raise the exceptions: UnderFlow and OverFlow.
--| The set includes:
--|
--|     Pop
--|     Push
--|     Reduce
--|
--| 2) A function that returns the number of elements in the
--| data structure. This set raises no exceptions.
--| The set includes:
--|
--|     Length

--|
--| Notes
--|
--|     Under some implementations the exception
--| ParserDeclarations.MemoryOverflow could be raised.
--|

    package PD renames ParserDeclarations;

    -----------------------------------------------------------------
    -- Declarations Global to Package ParseStack
    -----------------------------------------------------------------

    OverFlow  : exception;
        --| raised if no more space in stack.
    UnderFlow : exception;
        --| raised if no more elements in stack.

    -----------------------------------------------------------------

    procedure Push(         --| Adds new top element to stack
        Element: in PD.ParseStackElement); --| element to add

    --| Raises
    --|
    --| OverFlow - no more space in stack.

    --| Effects
    --|
    --| This subprogram adds an element to the top of the stack.
    --|

    -----------------------------------------------------------------

    function Pop                --| Removes top element in stack
        return PD.ParseStackElement;

    --| Raises
    --|
--| UnderFlow - no more elements in stack.

--| Effects
--|
--| This subprogram obtains the element at the top of the stack.
--|

-----------------------------------------------------------------

function Length                 --| Returns the number of
                                --| elements in the stack
    return PD.StateParseStacksIndex;

--| Effects
--|
--| This subprogram returns the number of elements in the stack.
--|

---------------------------------------------------------------------

procedure Reduce(           --| Pops and discards top n elements on
                            --| the stack.
    TopN : in PD.StateParseStacksIndex);
    --| Number of elements to pop.

--| Raises
--|
--| Underflow - no more elements in stack.

--| Effects
--|
--| Pops and discards top N elements on the stack.
--| If TopN is greater than the number of elements in the stack,
--| Underflow is raised.
--| This subprogram is used by the parser to reduce the stack during
--| a reduce action.
```

```
    --| This stack reduction could be done with a for loop and
    --| the Pop subprogram at a considerable cost in execution time.
    --|

    ----------------------------------------------------------------------

end ParseStack;

----------------------------------------------------------------------

-- $Source: /nosc/work/parser/RCS/ParseStk.bdy,v $
-- $Revision: 4.0 $ -- $Date: 85/02/19 11:34:13 $ -- $Author: carol $

----------------------------------------------------------------------

with ParseTables;                -- state tables generated by parser
                                 --     generator
use ParseTables;

with Grammar_Constants;
use Grammar_Constants;           -- to have visibility on operations
                                 -- on type ParserInteger declared there.
package body ParseStack is

--| Overview
--|
--| The data structure is implemented as an array.
--|

    ------------------------------------------------------------------
    -- Declarations Global to Package Body ParseStack
    ------------------------------------------------------------------

    Index       : PD.StateParseStacksIndex := 0;
        --| top element in stack.
```

```
Space : array (PD.StateParseStacksRange) of PD.ParseStackElement;
    --| Storage used to hold stack elements

----------------------------------------------------------------------
-- Subprogram Bodies Global to Package ParseStack
    -- (declared in package specification).
----------------------------------------------------------------------

procedure Push(Element : in PD.ParseStackElement) is

begin

    if (Index >= PD.StateParseStacksRange'Last) then
        raise OverFlow;
    end if;

    Index := Index + 1;
    Space (Index) := Element;

end Push;

----------------------------------------------------------------------

function Pop return PD.ParseStackElement is

begin

    if (Index < PD.StateParseStacksRange'First) then
        raise UnderFlow;
    end if;

    Index := Index - 1;
    return Space (Index + 1);

end Pop;
```

```
    -----------------------------------------------------------------

    function Length return PD.StateParseStacksIndex is

    begin

        return Index;

    end Length;

    -----------------------------------------------------------------

    procedure Reduce(TopN : in PD.StateParseStacksIndex) is

    begin
        if (TopN > Index) then
            raise UnderFlow;
        end if;

        Index := Index - TopN;

    end Reduce; -- procedure

    -----------------------------------------------------------------

end ParseStack;

---------------------------------------------------------------------
```

# Bibliography

[1] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, Inc., third edition, 1992.

[2] J. A. Siegel et al. National software capacity: Near-term study. Technical Report CMU/SEI-90-TR-12, Carnegie Mellon University/Software Engineering Institute, May 1990.

[3] Ada Joint Program Office. A strategy for a software initiative. Technical report, Department of Defense, 1985.

[4] Research in progress. University of Durham/The Centre for Software Maintenance, January 1995.

[5] Joseph Eugene Hollingsworth. *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada.* PhD thesis, Department of Computer and Information Science, The Ohio State University, 1992.

[6] M. C. Paulk et al. Capability maturity model for software. Technical Report CMU/SEI-91-TR-24, Carnegie Mellon University/Software Engineering Institute, November 1991.

[7] M. V. Wilkes, D. J. Wheeler, and S. Gill. *Programming for the Digital Computer.* Addison-Wesley, 1953.

[8] M. D. McIlroy. Mass produced software components. In P. Naur, B. Randell, and J. N. Buxton, editors, *Proceedings of NATO Conference on Software Engineering*, pages 88–98, New York, 1969. Petrocelli/Charter.

[9] Edward Stewart Garnett. *Software Reclamation: Upgrading Code for Reusability.* PhD thesis, Lancaster University, September 1990.

[10] Ted Biggerstaff. *Software Reusability, Concepts and Models*, volume I, page xv. ACM Press, 1989.

[11] James W. Hooper and Rowena O. Chester. *Software Reuse Guidelines and Methods.* Plenum Press, 1991.

[12] Norman E. Fenton. *Software Metrics: A rigorous approach.* Chapman & Hall, second edition, 1991.

[13] T. C. Jones. Reusability in programming: A survey of the state of the art. *IEEE Transactions on Software Engineering*, 10(5):488–494, September 1984.

[14] R. G. Lanergan and C. A. Grasso. Software engineering with reusable designs and code. *IEEE Transactions on Software Engineering*, 10(5):498–501, September 1984.

[15] B. W. Boehm. Improving software productivity. *IEEE Computer*, pages 43–58, September 1987.

[16] E. Horowitz and J. B. Munson. An expansive view of reusable software. *IEEE Transactions on Software Engineering*, 10(5):477–487, September 1984.

[17] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, September 1984.

[18] Grady Booch. *Software Engineering with Ada.* The Benjamin/Cummings Series in Ada and Software Engineering. The Benjamin/Cummings Publishing Company, Inc., second edition, 1987.

[19] Ricky W. Butler and Sally C. Johnson. Formal methods for life-critical software. In *The AIAA Computing in Aerospace 9 Conference*, pages 319–329, San Diego, Ca., October 1993. American Institute of Aeronautics and Astronautics Inc.

[20] P. A. V. Hall. Software reuse, reverse engineering, and re-engineering. In P. A. V. Hall, editor, *Software Reuse and Reverse Engineering in Practice*, chapter 1, pages 3–31. Chapman & Hall, 1992. First edition.

[21] Sholom Cohen. Process and products for software reuse and domain analysis. Software Engineering Institute.

[22] Ian Sommerville. Software reuse courses. Software Reuse Course Slides, 1994.

[23] I. Sommerville. *Software Engineering.* Addison-Wesley, third edition, 1989.

[24] Y. Matsumoto et al. SWB system: A software factory. *Software Engineering Environments*, pages 305–314, 1981.

[25] N. Akima and F. Ooi. Industrializing software development: A Japanese approach. *IEEE Software*, pages 13–22, March 1989.

[26] K. Geary. Practical problems in introducing software reuse, May 1987. IEE Colloquium on Reusable Software Components.

[27] Congress of the United States, Office of Technology Assessment. Intellectual property rights in an age of electronics and information. Technical report, Washington, D.C.: U.S. Government Printing Office, 1986.

[28] Pamela Samuelson and Kevein Deasy. Intellectual property protection for software. SEI Curriculum Module SEI-CM-14-2.1, Carnegie Mellon University, Software Engineering Institute, July 1989.

[29] Jody Armour and Watts S. Humphrey. Software product liability. Technical Report CMU/SEI-93-TR-13, ESC-TR-93-190, Carnegie Mellon University/Software Engineering Institute, August 1993.

[30] International Bible Society. *The Holy Bible, New International Version.* Hodder & Stoughton, 1994.

[31] Narain Gehani. *Ada, an advanced introduction.* Prentice-Hall Software Series. Prentice-Hall, Inc., 1984.

[32] Peter Wegner. *Programming with Ada.* Prentice-Hall Software Series. Prentice-Hall, Inc., 1980.

[33] Nell Dale, Chip Weems, and John McCormick. *Programming and Problem Solving with Ada.* D. C. Heath and Company, 1994.

[34] The Ada Joint Program Office (AJPO). *Reference Manual For The Ada Programming Language.* United States Department of Defense, 1983. ANSI/MIL-STD-1815A.

[35] Lloyd K. Mosemann. Ada and C++:a business case analysis. Pre-delevery draft, US Air Force, Air Force, SAF/AQK, Washington, June 1991. Press Conference Remarks by Lloyd K. Mosemann, II Deputy Assistant Secretary of the Air Force (Communications, Computers, and Logistics) on July 9, 1991.

[36] Brian Tooby. A brief introduction to Ada. In Chris Loftus, editor, *Ada Yearbook 1993*, chapter 1, pages 3–8. IOS Press, 1993. Studies in Computer and Communications Systems Volume 5.

[37] M. Ramachandran and I. Sommerville. Software reuse guidelines. Department of Computing, Lancaster University, Lancaster.

[38] John Nissen and Peter Wallis, editors. *Portability and style in Ada.* The Ada Companion Series. Cambridge University Press, 1984.

[39] R. J. St. Dennis. Reusable Ada software guidelines. In *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, pages 513–520, January 1987.

[40] The Ada-Europe Software Reuse Working Group. Ada reusability guidelines. In R. J. Gautier and P. J. L. Wallis, editors, *Software Reuse with Ada*, pages 99–173. Peter Peregrinus Ltd., 1990.

[41] IBM Systems Integration Division. STARS reusability guidelines. Technical report, Electronic Systems Division, Air Force Systems Command, USAF, Hanscomb AFB, Massachusetts, April 1990. Contract No. F19628-88-D-0032, Task IR40: Repository Integration, Delivered as part of: CDRL Sequence No. 1550.

[42] M. Ramachandran. *An Investigation into Tool Support for the Development of Reusable Software.* PhD thesis, Lancaster University, 1992.

[43] Software Productivity Consortium Services Corporation. *Ada 95 Quality and Style: Guidelines for Professional Programmers.* Draft Baseline Version, SPC-94093-CMC. Software Productivity Consortium Corporation, February 1995.

[44] I. Sommerville, L. Masera, and C. Demaria. Practical guidelines for ada reuse in an industrial environment. APPRAISAL Project Public Results, Lancaster University, 1995.

[45] Y. Matsumoto. Some experiences in promoting reusable software presentation in higher abstract levels. *IEEE Transactions on Software Engineering*, SE-10(5), September 1981.

[46] Spencer Rugaber. Program comprehension for reverse engineering. College of Computing, Georgia Institute of Technology, Atlanta, Georgia.

[47] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984.

[48] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[49] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, (2), 1974.

[50] David W. Embley and Scott N. Woodfield. Cohesion and coupling for abstract data types. In *Proceedings of Sixth Phoenix Conference on Computers and Communications*, Phoenix, Arizona, 1987.

[51] D. L. Parnas. On criteria to be used in decomposing systems into modules. *Communications of ACM*, 14(1):221–227, April 1972.

[52] J. G. P. Barnes. *Programming in Ada.* Addison-Wesley Publishers Limited, second edition, 1984.

[53] John Rymer and Tom McKeever. The FSD Ada style guide, 1986.

[54] Intermetrics, Inc. *Information Technology – Programming Languages – Ada.* International Organization for Standardization, International Electrotechnical Commission, 1995. INTERNATIONAL STANDARD ISO/IEC 8652:1995(E).

[55] Frank Pappas. Ada portability guidelines. Technical report, SofTech Inc., March 1985.

[56] E. R. Matthews. Observations on the portability of Ada I/O. *ACM SIGAda Letters*, VII(5), September 1987.

[57] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the Second International Conference on Software Engineering*, pages 492–497. IEEE, October 1976.

[58] Gary Ford. Lecture notes on engineering measurement for software engineers. SEI educational materials package CMU/SEI-93-EM-9, Carnegie Mellon University, Software Engineering Institute, 1993.

[59] Mark Lorenz. *Object-oriented software metrics: a practical guide.* PTR Prentice Hall, 1994.

[60] Everald E. Mills. Software metrics. SEI Curriculum Module SEI-CM-12-1.1, Carnegie Mellon University, Software Engineering Institute, December 1988.

[61] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[62] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models.* BenjaminCummings, Menlo Park, Calif., 1986.

[63] V. R. Basili and D. M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, November 1984.

[64] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, N. J., 1981.

[65] T. C. Jones. *Programming Productivity.* McGraw-Hill, New York, 1986.

[66] Walcélio L. Melo, Lionel C. Briand, and Victor R. Basili. Measuring the impact of reuse on quality and productivity in object-oriented systems. Technical Report CS-TR-3395, University of Maryland, Department of Computer Science, January 1995.

[67] Lyman Ott. *An introduction to statistical methods and data analysis.* PWS-KENT Publishing Company, Boston, Massachusetts, third edition, 1988.